UNIVERSIDADE ESTADUAL DE CAMPINAS
Faculdade de Engenharia Elétrica e de Computação

Rodrigo Alexander de Andrade Pierini

Design, implementação e avaliação de atestação remota distribuída em plano de dados programável

Campinas/SP

2025

Rodrigo Alexander de Andrade Pierini

Design, implementação e avaliação de atestação remota distribuída em plano de dados programável

Dissertação de mestrado apresentado à Faculdade de Engenharia Elétrica e de Computação da Universidade Estadual de Campinas como parte dos requisitos exigidos para a obtenção do título de Mestre em Engenharia Elétrica, na Área de Engenharia de Computação.

Orientador: Prof. Dr. Marco Aurélio Amaral Henriques

Co-orientador Prof. Dr. Christian Rodolfo Esteve Rothenberg

Este exemplar corresponde à dissertação de mestrado entregue à banca e defendida pelo aluno Rodrigo Alexander de Andrade Pierini, orientado pelo Prof. Dr. Marco Aurélio Amaral Henriques

---

Campinas/SP

2025

Ficha catalográfica
Universidade Estadual de Campinas (UNICAMP)
Biblioteca da Área de Engenharia e Arquitetura
Rose Meire da Silva - CRB 8/5974

Pierini, Rodrigo Alexander de Andrade, 1993-
An24d Design, implementação e avaliação de atestação remota distribuída em plano de dados programável / Rodrigo Alexander de Andrade Pierini. – Campinas, SP : [s.n.], 2025.

Orientador: Marco Aurélio Amaral Henriques.
Coorientador: Christian Rodolfo Esteve Rothenberg.
Dissertação (mestrado) – Universidade Estadual de Campinas (UNICAMP), Faculdade de Engenharia Elétrica e de Computação.

1. Redes definidas por software. 2. Criptografia. 3. Funções Hash. 4. Avaliação remota. I. Henriques, Marco Aurélio Amaral, 1963-. II. Rothenberg, Christian Rodolfo Esteve, 1982-. III. Universidade Estadual de Campinas (UNICAMP). Faculdade de Engenharia Elétrica e de Computação. IV. Título.

Informações complementares

Título em outro idioma: Design, implementation and evaluation of distributed remote attestation in programmable data plane
Palavras-chave em inglês:
Software-defined networking
Cryptography
Hash functions
Remote attestation
Área de concentração: Engenharia de Computação
Titulação: Mestre em Engenharia Elétrica
Banca examinadora:
Marco Aurélio Amaral Henriques [Orientador]
Ramon dos Reis Fontes
Alberto Egon Schaeffer Filho
Data de defesa: 17-01-2025
Programa de Pós-Graduação: Engenharia Elétrica

Objetivos de Desenvolvimento Sustentável (ODS)
ODS: 9. Inovação e infraestrutura

Identificação e informações acadêmicas do(a) aluno(a)
- ORCID do autor: https://orcid.org/0009-0009-7707-5194
- Currículo Lattes do autor: http://lattes.cnpq.br/8669256248614537

COMISSÃO JULGADORA - DISSERTAÇÃO DE MESTRADO

Candidato: Rodrigo Alexander de Andrade Pierini RA: 263748

Data da Defesa: 17 de janeiro de 2025

Título da Tese: "Design, implementação e avaliação de atestação remota distribuída em plano de dados programável”

Prof. Dr. Marco Aurélio Amaral Henriques (Presidente)

Prof. Dr. Alberto Egon Schaeffer Filho

Prof. Dr. Ramon dos Reis Fontes

A ata de defesa, com as respectivas assinaturas dos membros da Comissão Julgadora, encontra-se no SIGA (Sistema de Fluxo de Dissertação/Tese) e na Secretaria de PósGraduação da Faculdade de Engenharia Elétrica e de Computação.

Resumo

Um plano de dados programável é uma tecnologia essencial para vários paradigmas existentes e emergentes em redes de computadores que trazem consigo diversos desafios em segurança. Este trabalho visa aprimorar a eficiência e escalabilidade da verificação de conformidade de sistemas de computação, empregando a Atestação Remota (*Remote Attestation*, RA) como um estudo de caso, com foco no contexto de data centers. A abordagem proposta envolve a delegação de funções (*offloading*) para switches programáveis. Com essa abordagem, é possível tornar o processo de atestação de dispositivos em média 153,5 vezes mais rápido em comparação a uma atestação sem uso dos switches, permitindo verificações de conformidade mais frequentes e reduzindo a janela de tempo disponível para ataques entre verificações.

Palavras-chaves: Atestação remota; Redes definidas por software; Redes programáveis; Criptografia simétrica; Funções de hash.

Abstract

A programmable data plane is an essential technology for various existing and emerging paradigms in computer networks, bringing with it numerous security challenges. This work aims to improve the efficiency and scalability of compliance verification in computing systems by using Remote Attestation (RA) as a case study, focusing on data center environments. The proposed approach involves delegating functions (offloading) to programmable switches. With this approach, the attestation process for devices can be made, on average, 153.5 times faster compared to attestation without switches, enabling more frequent compliance checks and reducing the time window available for attacks between verifications.

Keywords: Remote attestation; Software-defined networking; Programmable networks; Symmetric cryptography; Hash functions.

Lista de ilustrações

Lista de tabelas

Sumário

1 Introdução

Garantir a segurança de um sistema computacional é uma empreitada desafiadora. Com a sociedade cada vez mais dependente da tecnologia, a demanda por capacidade de processamento aumenta, resultando na necessidade de mais recursos computacionais em várias situações. Isso, por sua vez, aumenta a vulnerabilidade a ataques que exploram a complexidade adicional relacionada à expansão de recursos.

Dentro do contexto das redes de computadores, surge uma questão fundamental: "Como podemos assegurar a confiabilidade dos dispositivos que fazem parte dessa infraestrutura computacional?". Esse desafio é extensivamente explorado em ambientes como servidores e internet das coisas (*Internet of Things*, IoT). Uma técnica frequentemente empregada para abordar essa questão é conhecida como "Atestação Remota" (*Remote Attestation*, RA).

A RA é uma técnica que verifica a integridade de um dispositivo de forma remota. Nessa técnica, um **provador** deve apresentar a um **verificador** uma prova de que está em um estado esperado. O estado esperado pode ser definido de diversas formas através de uma política que indica quais arquivos, parâmetros e configurações devem compor a prova. Além disso, é preciso que o verificador possa confirmar que a prova apresentada seja autêntica, confiável e atual.

A RA utiliza várias ferramentas criptográficas para garantir a confiabilidade, autenticidade e validade de uma prova de conformidade apresentada. No entanto, ela concentra uma carga computacional significativa no equipamento que verifica a conformidade dos demais, o que dificulta sua escalabilidade. Para abordar esse problema, no contexto de IoT, é empregada uma variante conhecida como "Atestação Remota Coletiva" (*Collective Remote Attestation*, CRA). Nessa variante, as evidências de conformidade fornecidas por um conjunto de dispositivos são agregadas para determinar o estado coletivo da rede, ou seja, se todos os dispositivos estão em conformidade ou se algum não está.

Este estudo tem como objetivo explorar a aplicação de RA em ambientes de data centers, que são particularmente desafiadores devido à densidade de dispositivos a serem atestados e ao custo dos recursos, como poder de processamento, capacidade de armazenamento e capacidade de transmissão de dados, que são alugados aos clientes (*tenants*) do data center. Portanto, busca-se maneiras de garantir a segurança das plataformas computacionais disponíveis com o mínimo de impacto nestes recursos.

Uma abordagem adotada para minimizar esse impacto é a descarga de funções (*offloading*) que são normalmente realizadas por processadores centrais (CPUs) dos

servidores para os equipamentos de rede programáveis disponíveis, como *switches* programáveis e smartNICs (*Smart Network Interface Card*). Assim, a pergunta central deste trabalho é: *Como reduzir o impacto da atestação remota nos recursos de um data center por meio dos recursos de redes programáveis disponíveis em sua infraestrutura?*

1.1 Definição do problema

A matéria-prima de prestação de serviço de um data center de núvem pública são seus recursos computacionais, como CPUs, memória RAM, sistemas de armazenamento, GPUs e infraestrutura de rede. Esses recursos são disponibilizados aos seus clientes em diversas formas de cargas de trabalho (*workloads*) e faturadas de acordo com sua alocação.

Nesse contexto, o uso de recursos para outras finalidades além dos serviços prestados pelo data center é considerado custo operacional. Portanto, operações de gerenciamento e segurança representam um custo para a operação do data center.

Para reduzir a utilização destes recursos, um paradigma chamado "*In-Network Computing*" surgiu com a proposta de realizar a descarga de funções antes realizadas por CPUs para hardwares programáveis, como os ASICs (*Application-Specific Integrated Circuit*) encontrados em *switches* programáveis e smartNICs. Já existem pesquisas nesse campo, abrangendo a aceleração de funções em sistemas distribuídos (DANG *et al.*, 2020), aprendizado de máquina (XIONG; ZILBERMAN, 2019) e algumas operações criptográficas (OLIVEIRA *et al.*, 2021).

Realizar a verificação de conformidade dos servidores de um data center por meio do uso dos recursos alocáveis a clientes implica em um custo operacional de gerenciamento e segurança. Acentuando esse cenário, há ataques que se aproveitam do tempo entre verificações para se explorar vulnerabilidades e esconder seus rastros, criando a necessidade da redução da janela entre verificações para proteger a infraestrutura contra esses ataques.

Como será descrito na Seção 2.2, o serviço de nuvem da Google faz uma verificação sob-demanda dos seus servidores, de forma que um servidor pode passar períodos consideráveis sem verificação. Ao nosso conhecimento, não há uma abordagem de verificação de conformidade em data centers que busque reduzir essa janela de ataque entre verificações.

A Figura 1.1 ilustra uma comparação de uma operação íntegra de um servidor de data center ao longo do tempo e o cenário de um ataque do tipo TOCCTOU (*Time-of-check-to-time-of-use*), onde um atacante aproveita os períodos entre verificações para

comprometer o funcionamento ou roubar dados durante a operação do servidor e oculta sua presença antes da próxima verificação realizada. Uma forma de se evitar ataques do tipo TOCTTOU é através do aumento da frequência de atestações realizadas, reduzindo a janela de tempo que um atacante possui para obter resultados e ocultar sua presença.

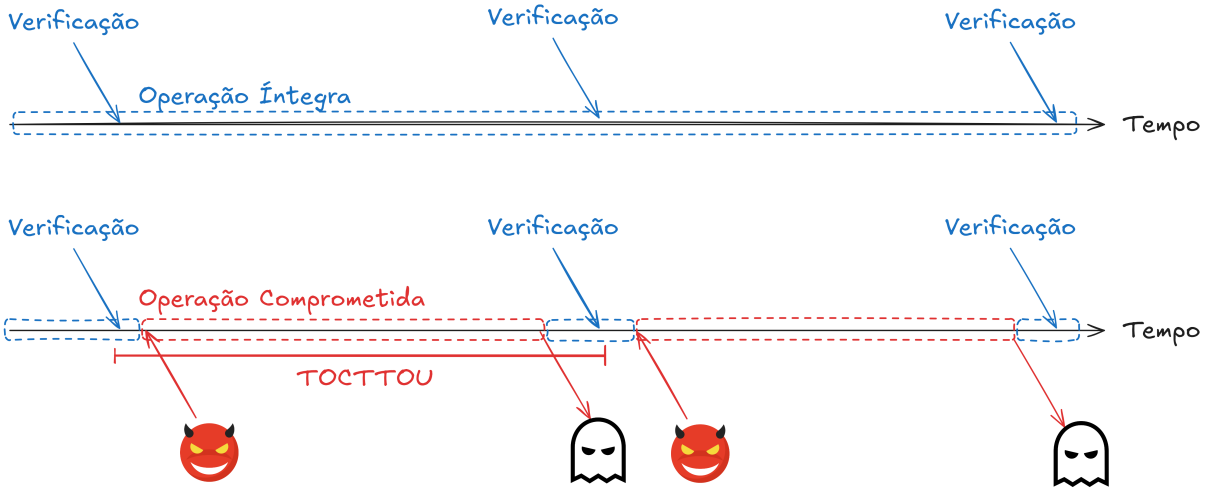

Figura 1.1 – Exemplo de um cenário de ataque TOCTTOU que ocorre entre verificações de integridade de um dispositivo remoto

Essa operação de atestação poderia ser descarregada para a rede programável presente em data centers para liberar recursos alocáveis a clientes e reduzir os custos operacionais. Portanto, o problema a ser investigado é: como aumentar a frequência de atestações em um ambiente de data center com redução dos seus custos operacionais neste processo.

1.2 Objetivos

O propósito deste trabalho é aplicar os princípios de plano de dados programável para aprimorar o processo de RA em servidores de data centers. Isso envolve encontrar maneiras de executar esse processo em um ambiente criptograficamente restrito, como o plano de dados de *switches* programáveis. Esse ambiente apresenta desafios, uma vez que algumas primitivas básicas, como estruturas de repetição, não estão disponíveis nessa plataforma computacional.

Além disso, é crucial que a solução proposta seja validada em um modelo de ataque previamente definido, alinhada com o ambiente em que será implementada. Não é suficiente demonstrar o funcionamento da solução: ela deve ser resistente a possíveis ataques de atacantes com capacidades e motivações específicas.

O desejo é ir além do que é proporcionado pelo método de CRA. No contexto de data centers, é fundamental identificar rapidamente qualquer equipamento que

não esteja em conformidade para que sejam tomadas ações corretivas e mitigar possíveis danos. Portanto, é essencial realizar uma validação individual de cada dispositivo, em vez de apenas identificar um estado coletivo da rede, como obtido na CRA. Acredita-se que a topologia de uma rede de data centers (organizada em setores) possa facilitar a escalabilidade na verificação dos dispositivos de forma individualizada.

Com essa abordagem, o objetivo geral é reduzir a janela entre verificações, de forma que o tempo disponível para um atacante realizar alterações nos dispositivos seja inviável para ataques que visam passar despercebidos pela verificação. Isto é, aumentar a frequência de atestações sem causar prejuízos à operação dos dispositivos e do verificador. Para alcançar esse objetivo geral, alguns objetivos específicos devem ser alcançados:

- definir uma abordagem que seja eficiente para ser implementada em planos de dados programáveis (PDP) de forma a permitir a verificação de integridade, autenticidade e validade de uma prova de atestação;
- definir um protocolo de atestação que viabilize o processo de atestação utilizando PDP como verificadores;
- implementar e analisar o protocolo e a abordagem proposta de forma a verificar sua viabilidade em comparação a uma técnica de atestação remota executada em um verificador baseado na arquitetura x86.

O trabalho apresenta uma solução que visa testar a viabilidade de usar o plano de dados programável como um verificador e comparar seu desempenho com uma solução baseada em servidores comerciais prontos para uso (*off-the-shelf*). Essa análise envolve avaliar os impactos na alocação de recursos computacionais ao utilizar o plano de dados.

1.3 Metodologia

O processo inclui diversas etapas, começando pelo levantamento da literatura relevante para embasar a proposta e compreender o problema em questão. Além disso, foi realizada uma caracterização do modelo de ataque, considerando as capacidades e motivações de possíveis atacantes, a fim de realizar uma análise de segurança da solução proposta.

No estágio final, foram aplicadas métricas identificadas na literatura para validar a viabilidade da solução e compará-la com outras alternativas já existentes. Os resultados obtidos nessa fase são discutidos considerando o cenário de experimentação com os protótipos desenvolvidos.

1.4 Contribuições e organização da dissertação

Este trabalho traz contribuições para o estado da arte por meio do projeto e avaliação de uma técnica aberta de atestação remota para cenários de data center utilizando plano de dados programáveis. Essa técnica reduz consideravelmente a janela de frequência de atestações em relação à quantidade de dispositivos disponíveis em uma rede, visto que torna o processo de verificação de provas de atestação muito mais rápido.

Um protocolo é definido de forma a generalizar o processo de atestação remota por meio do uso de *switches* programáveis como um verificador intermediário, reduzindo o gargalo do processo de atestação em um verificador central para uma grande quantidade de equipamentos. O projeto também contribui propondo um algoritmo de cálculo da prova e da verificação baseado em algoritmos de cifra de fluxo, de forma a tornar o processo mais adequado para processadores de pacotes programáveis realizarem o cálculo da verificação da prova.

Por fim, os experimentos realizados nesta dissertação validam o desempenho da solução proposta, de forma a torná-la comparativa com outras abordagens possíveis. Este trabalho fez a comparação entre a verificação baseada em CPU e a descarga da verificação para um *switch* programável, de forma a verificar a desoneração dos recursos de um verificador central em comparação a vários verificadores espalhados pela rede programável. Os códigos-fonte das implementações realizadas durante essa pesquisa estão disponíveis no repositório do Github e podem ser livremente utilizados e modificados.[1]

Durante a elaboração desta dissertação, um artigo foi publicado no Simpósio Brasileiro de Segurança da Informação e Sistemas Computacionais (SBSeg) de 2024, onde avalia-se a implementação de algoritmos de cifra de fluxo que poderiam ser utilizados no processo de atestação remota aplicado a PDP, comparando a implementação do algoritmo ChaCha20 disponível na literatura com uma nova implementação do algoritmo Forro14.[2]

Nesse capítulo foram apresentados de forma breve o contexto em que a pesquisa está inserida, o problema explorado, o objetivo deste trabalho, a metodologia empregada e as contribuições para o estado da arte. A solução implementada e essas contribuições serão melhor detalhadas nos próximos capítulos da dissertação.

No Capítulo 2 são apresentados os conceitos para contextualização do problema e trabalhos relacionados. Adiante, descreve-se no Capítulo 3 o protocolo proposto, elencando seus requisitos conforme a literatura e discorrendo sobre os detalhes técnicos da implementação realizada para a avaliação em prova de conceito. No capítulo 4, a implementação da solução é detalhada, ficando a demonstração e discussão dos dados coletados

[1] <https://github.com/regras/p4dra>
[2] <https://github.com/regras/p4-forro>

por meio de experimentos para o capítulo 5. Por fim, no Capítulo 6 é apresentada a conclusão do trabalho e são descritos possíveis trabalhos futuros para continuidade dessa pesquisa, de forma a expandir o estado da arte.

2 Fundamentos

Para contextualizar o trabalho realizado, um levantamento bibliográfico foi conduzido abrangendo artigos científicos de especialistas da área, a fim de identificar técnicas relacionadas a RA e CRA, entender como estas são aplicadas e destacar os conceitos essenciais associados a elas.

Com base nesse levantamento bibliográfico, serão apresentados conceitos ligados à RA, aos planos de dados programáveis, às cifras de fluxo e às suas aplicações em data centers. O propósito é proporcionar uma compreensão dos fundamentos necessários para a pesquisa e seu contexto, justificando a relevância do projeto.

2.1 Atestação remota

A técnica de RA é utilizada para verificar a integridade ou a conformidade de um dispositivo de forma remota. Essa verificação de integridade é garantida por meio de algumas técnicas e ferramentas criptográficas que geram uma prova autêntica e segura do estado atual do dispositivo.

Para que o dispositivo a ser verificado (chamado “provador”) possa gerar essa prova, é necessário que haja uma base computacional confiável (*Trusted Computing Base*, TCB) que garanta que até mesmo seu sistema operacional não tenha influência direta sobre a prova gerada. Um caso de uso comum da atestação remota é para verificar se o dispositivo passou por uma inicialização conforme e chegou a um estado esperado corretamente, sem alterações dos componentes da inicialização por um agente malicioso. Esse processo de inicialização verificada é chamado de "*boot* seguro" (*Secure Boot*, SB). (LING *et al.*, 2021)

Para garantir que o SB é confiável, é necessário que exista uma raiz de confiança (*Root of Trust*, RoT), considerada confiável pelo operador do sistema, para verificar o primeiro componente do processo de inicialização. Com o primeiro componente considerado confiável, este pode fazer a verificação do próximo componente da inicialização antes de sua execução e, assim, sucessivamente até que o sistema operacional tenha sido inicializado. Esse processo de verificação encadeada é chamada de cadeia de confiança (*Chain of Trust*, CoT) (TCG, 2024a). A Figura 2.1 ilustra este conceito.

A RoT pode ser implementada em hardware, software ou de maneira híbrida. As implementações de software geralmente são utilizadas em cenários com dispositivos com hardware minimalista ou atacantes com capacidades limitadas. Por isso, estão sendo

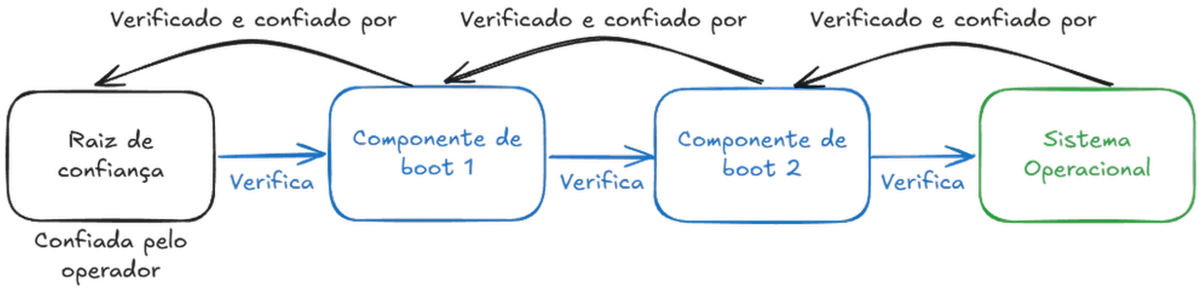

Figura 2.1 – Processo de cadeia de confiança em Boot Seguro

substituídas por abordagens híbridas ou baseadas em hardware (AMBROSIN *et al.*, 2020).

Uma abordagem comum para a raiz de confiança em hardware é o uso de uma Unidade de Proteção de Memória (*Memory Protection Unit*, MPU), que permite limitar o acesso a uma área de memória a recursos ou processos autorizados. Essa operação é isolada do sistema operacional pelo próprio processador, criando o que é chamado de "Ambiente de Execução Confiável" (*Trusted Execution Environment*, TEE). Exemplos dessas soluções incluem o "ARM TrustZone" (ARM, 2023), "AMD Secure Encrypted Virtualization" (SEV) (AMD, 2023) e o "Intel Secure Guard eXtensions" (SGX) (INTEL, 2023a).

Para que o *provador* forneça garantias de segurança adequadas, certas restrições devem ser cumpridas para impedir a falsificação da prova por um atacante. Isso pode envolver o uso de componentes dedicados de segurança, como um "Módulo de Plataforma Confiável" (*Trusted Platform Module*, TPM) (TCG, 2024b), ou a criação de um ambiente isolado no TEE.

O dispositivo que recebe a prova emitida pelo provador (chamado "verificador") deve conhecer o estado íntegro do dispositivo antes de solicitar a prova de atestação. Para garantir a integridade de um dispositivo, é essencial definir o que se considera íntegro por meio de uma política que descreve o que e como deve ser medido, determinando um resultado esperado para essa medição. O processo de atestação pode ser comparado a um "Teste de Resposta Conhecida" (*Known Answer Test*, KAT): o *provador* deve apresentar um resultado do algoritmo de atestação de acordo com a política estabelecida, e o *verificador* compara esse resultado com o valor considerado como correto. A Figura 2.2 ilustra esse processo.

2.2 Atestação remota no contexto de data centers

Data centers são infraestruturas complexas que abrigam equipamentos de processamento, armazenamento e transmissão de dados para otimizar a operação destes em condições ambientais, operacionais e de segurança adequadas. Considerando este cenário,

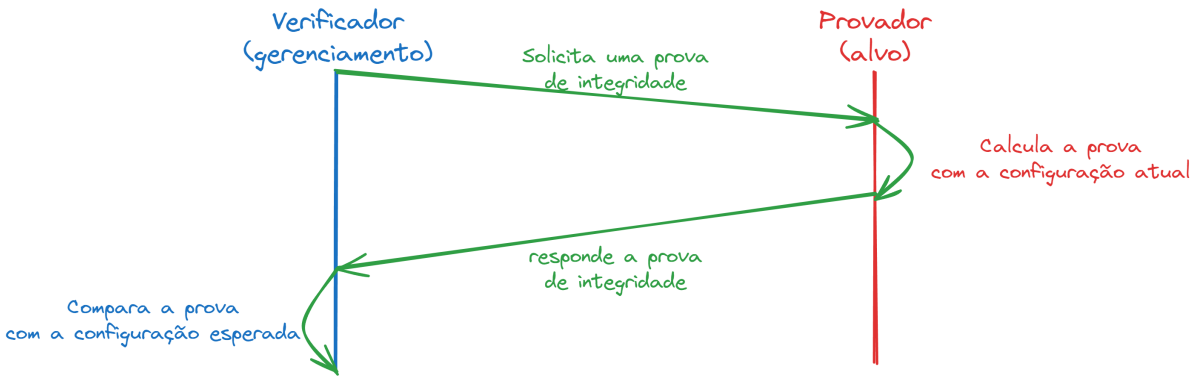

Figura 2.2 – Processo de atestação remota entre verificador e provador

o gerenciamento de um data center é uma atividade complexa e custosa e, para auxiliar em seu gerenciamento, diversas tecnologias são empregadas. Uma destas tecnologias é um componente embarcado em servidores para permitir seu ciclo de vida e de operação de forma remota e centralizada, chamado *Baseboard Management Console* (BMC). Este BMC é comumente um processador embarcado na placa principal e baseado na arquitetura ARM, com recursos de vídeo e lógica de controle, podendo ser acessado por uma rede dedicada ou compartilhada. (Gigabyte, 2024)

Há também protocolos que visam garantir a interoperabilidade de diversos fabricantes dentro de um mesmo data center, permitindo um gerenciamento centralizado de toda a infraestrutura. Esses protocolos são definidos em padrões emitidos pela *Distributed Management Task Force* (DMTF) e são adotados por diversos provedores, fabricantes e bibliotecas de software. Um dos padrões emitidos pela DMTF define o protocolo *Redfish*, responsável por realizar o inventário e gerenciamento de componentes e ciclo de vida dos equipamentos alocados no data center. (DMTF, 2024a)

Além do desafio do gerenciamento da infraestrutura, data centers de hiperescala enfrentam o desafio de garantir a integridade de seus recursos computacionais antes de alocar cargas de trabalho para operação. Essa integridade pode ser comprometida por diversos motivos, incluindo erros de software, reparos inadequados ou atividades maliciosas. Atualmente, três empresas se destacam como provedoras de serviços em nuvem (*Cloud Service Providers*, CSP): *Google Cloud Platform* (GCP), *Microsoft Azure* e *Amazon Web Services* (AWS). (SLINGERLAND, 2024)

A Google realiza a RA antes da alocação de uma carga de trabalho com a ajuda de um escalonador chamado *Job Scheduler*, presente em seu plano de controle. A atestação define um padrão válido de operação da máquina, que é usado para comparar com a prova de atestação fornecida pela máquina selecionada. A estrutura da máquina envolve uma arquitetura desagregada com vários TPMs agrupados pelo BMC. Esses controladores agregam e assinam as provas, que são posteriormente entregues ao verificador

para conferência. A documentação não menciona a plataforma computacional do verificador nem os algoritmos utilizados para assinatura e resumo criptográfico (GOOGLE, 2023).

A literatura indica que a catalogação dos recursos a serem atestados é realizada por meio do protocolo *Redfish* e a comunicação entre o BMC e os TPMs é feita por meio do protocolo de segurança "SPDM" (*Security Protocol and Data Model*), padronizado pela DMTF. (DMTF, 2024b)

A Azure oferece um serviço chamado "*Azure Attestation*", que permite aos clientes solicitarem uma prova de atestação para suas cargas de trabalho na nuvem, por meio de um TPM virtualizado em sua própria máquina virtual. Esse cenário difere um pouco do utilizado pela Google, visto que busca possibilitar ao cliente verificar a conformidade de sua própria carga de trabalho por meio de um serviço fornecido pelo CSP (MICROSOFT, 2023). Em contraste, a Google realiza a atestação do estado de sua própria infraestrutura antes de alocar a carga de trabalho. O serviço da Azure se concentra na verificação de enclaves de isolamento nas cargas de trabalho do cliente.

A AWS, por sua vez, oferece o "AWS Nitro" com a capacidade "AWS Nitro Enclave Attestation". A abordagem é semelhante à Azure e visa fornecer um serviço de atestação para os enclaves de memória que garantem o isolamento das cargas de trabalho hospedadas pelo CSP (AWS, 2023).

2.3 Paradigmas de redes de computadores

Nos últimos anos, novos paradigmas surgiram no contexto de redes de computadores, atribuindo uma maior importância financeira e maior desempenho à gestão e operação da rede, sobretudo em redes metropolitanas e redes de data centers. A Figura 2.3 ilustra o funcionamento de uma rede de computadores tradicional.

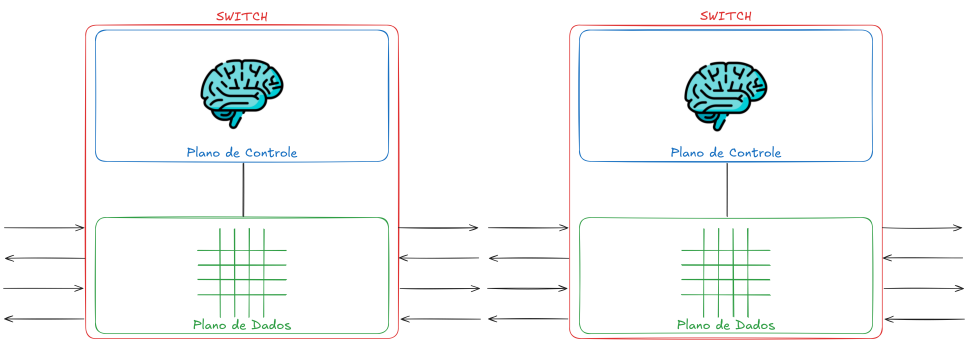

Figura 2.3 – Rede de computadores tradicional

A principal mudança nesses paradigmas é a separação entre o encaminhamento de pacotes (denominado "plano de dados") e a inteligência que define a forma de fazer esse

encaminhamento (denominado "plano de controle") dos *switches* de uma rede, movendo a inteligência para um controlador central com visão de toda a rede. Esse é o conceito de redes definidas por software (*Software-Defined Networking*, SDN) (KREUTZ *et al.*, 2014) que realiza a configuração do plano de dados por meio do protocolo OpenFlow (FERNANDES; ROTHENBERG, 2014). Esse paradigma é ilustrado na Figura 2.4.

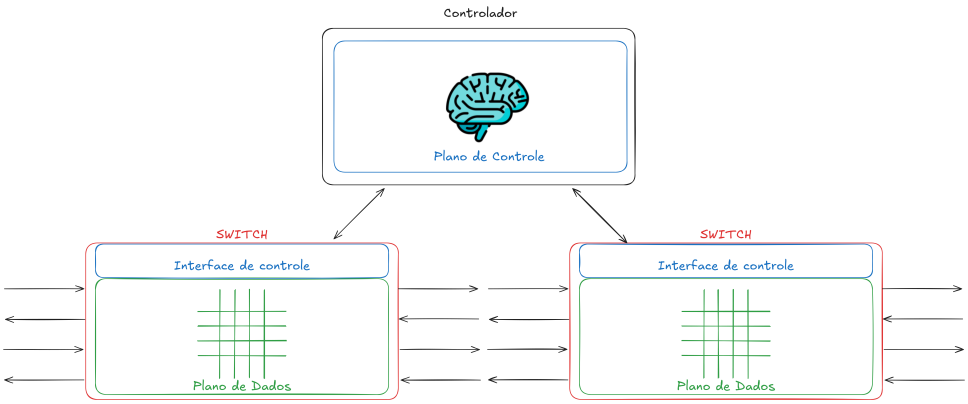

Figura 2.4 – Paradigma de rede de computadores definida por software

Posteriormente, surgiu a possibilidade de definir de forma programática como o plano de dados realiza a interpretação dos cabeçalhos de pacotes e seu processamento e encaminhamento. Esse é o paradigma de "plano de dados programável" (*Programmable dataplanes*, PDP), onde circuitos integrados de aplicação específica em *switches* são projetados para realizar o processamento de pacotes de forma dinâmica e independente de protocolos. Um exemplo desses ASICs é a série *Intel Barefoot Tofino*, conforme detalhado no site do fabricante (INTEL, 2023b).

A programação desses ASICs é realizada por meio de uma linguagem de domínio específico denominada P4 (*Programming Protocol-Independent Packet Processors*). A linguagem P4 permite a análise, processamento e síntese de pacotes de dados por meio de tabelas preenchidas pelo plano de controle e ações definidas pelo desenvolvedor, complementada por uma API gerada para a gestão do "*pipeline*" do plano de dados desenvolvido (BOSSHART *et al.*, 2014). Esse paradigma de redes programáveis é ilustrado na Figura 2.5

Com a possibilidade de realizar processamento de dados nos dispositivos que tradicionalmente realizavam apenas o encaminhamento de pacotes, surge a possibilidade de um processamento desagregado, repassando funções antes realizadas por processadores de propósito geral para hardwares mais especializados, inclusive enquanto é feito o encaminhamento de dados entre dois dispositivos processadores. Esse paradigma de realizar o processamento de dados durante o encaminhamento de pacotes é chamado de "*In-Network Computing*" (INC) e é ilustrado na Figura 2.6.

Para realizar a implementação do papel de verificador da atestação remota,

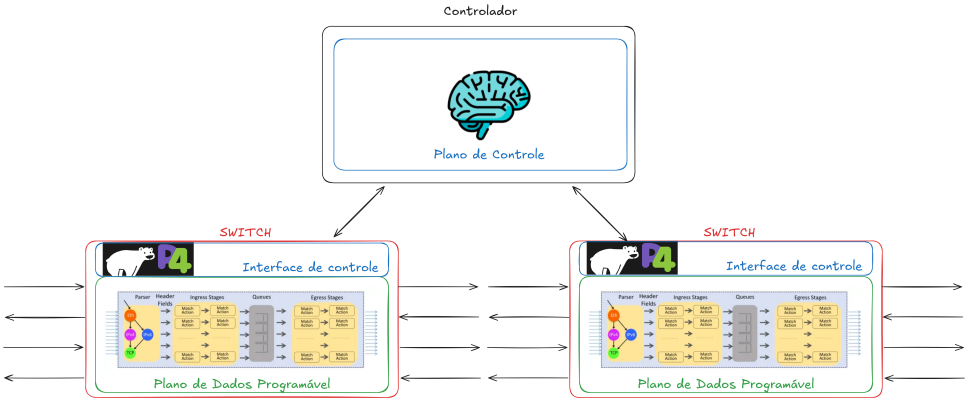

Figura 2.5 – Rede de computadores no paradigma de plano de dados programável

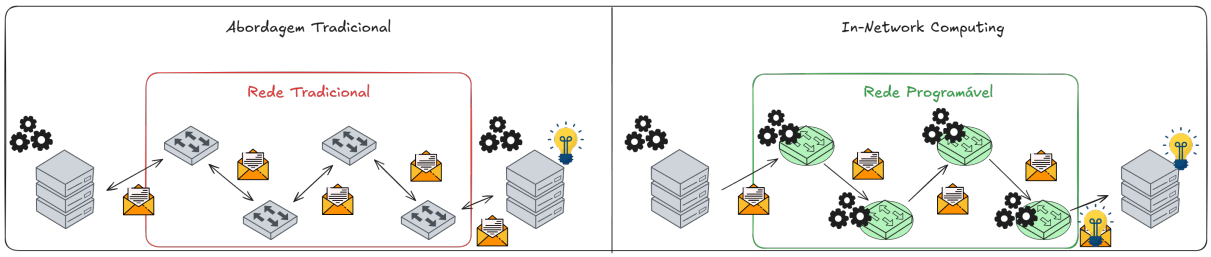

Figura 2.6 – Exemplo de computação em rede. Nesse cenário, os *switches* programáveis também participam do processamento de dados, acelerando o processamento de dados compartilhados entre dispositivos.

foi utilizada a arquitetura Intel Tofino devido à sua ampla adoção em data centers. Essa arquitetura utiliza a arquitetura de referência *Protocol-Independent Switch Architecture* (PISA), que prevê o uso de dois *pipelines* para a manipulação de dados de cabeçalhos extraídos de um pacote e seus metadados. A Figura 2.7 ilustra esta arquitetura de forma sucinta.

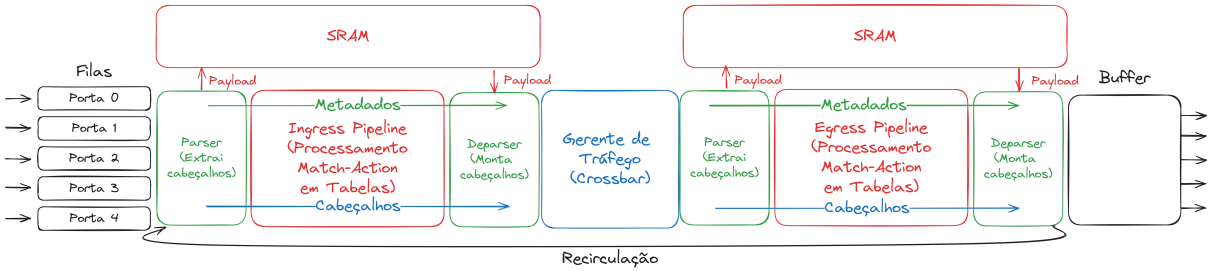

Figura 2.7 – Ilustração da arquitetura PISA.

A ASIC Tofino utiliza uma arquitetura derivada da PISA, chamada *Tofino Native Architecture* (TNA). Nessa arquitetura, os *pipelines* de entrada e saída possuem 12 estágios de processamento. Esses estágios são compostos por unidades de busca em tabela (LT) e unidades de lógica e aritmética (ALU) que utilizam os dados coletados dos cabeçalhos e metadados para realizar os processamentos definidos pelo operador. Operações independentes podem ser realizadas em ALUs do mesmo estágio de forma paralela, mas operações dependentes entre si devem ser executadas em estágios diferentes.

Esses *pipelines* de processamento são definidos em alto nível por meio da linguagem P4 para serem compilados em um formato que a ferramenta de programação da ASIC possa entender. No caso da arquitetura TNA, o compilador transforma o código de alto-nível P4 em um *bitstream* que é intepretado pelo driver da ASIC, fazendo sua programação.

Devido ao domínio da linguagem ser referente a processadores de pacotes, onde espera-se que o processamento seja linear, o P4 não tem suporte para estruturas de repetição. Para se obter um comportamento de repetição no processamento, é necessário utilizar o artifício de recirculação de pacotes, de forma que o pacote é resubmetido ao *switch* para passar novamente pelo processamento sob novas condições.

Uma limitação importante da arquitetura TNA é referente à quantidade de bits que podem ser analisados no *parser* que faz a varredura nos *pipelines* de entrada e de saída. O *pipeline* de entrada possui uma limitação de extração de até 4096b do pacote, sendo que 192b são utilizados como metadados da arquitetura TNA. Já o *pipeline* de saída possui uma limitação de 1480b para extração do cabeçalho, também utilizando 192b por metadados da TNA.

Para processamento das informações pelo protocolo proposto, os dados devem ser inseridos como cabeçalhos dos pacotes. Isso implica que a utilização de cabeçalhos no protocolo deve ser otimizada, para que o *switch* tenha capacidade de extrair os 512b do vetor de estado que está sendo calculado no algoritmo de cifra de fluxo, juntamente com outros metadados. Portanto, utiliza-se um cabeçalho inserido após o cabeçalho *ethernet*, identificado pelo campo *ethertype* de valor 0x1234.

2.4 Descarga de processamento

Os data centers costumam ser percebidos como estruturas maciças e altamente otimizadas, nas quais servidores são interconectados por uma rede de alta velocidade para encaminhar dados com a menor latência possível. No entanto, esse paradigma está passando por uma transformação em direção ao conceito de "desagregação computacional," que torna o data center em algo análogo a um único computador com recursos de armazenamento, processamento e interconexão (OKADA *et al.*, 2023).

As tecnologias de programação de dispositivos de rede estão possibilitando a descarga de funções anteriormente executadas por CPUs de propósito geral para componentes de aceleração de processamento especializados encontrados em placas de expansão, como SmartNICs, GPUs e arranjos de porta programáveis em campo (*Field-Programmable Gate Array*, FPGAs), ou em dispositivos intermediários conhecidos como "*middleboxes*",

como *switches* programáveis ou outros equipamentos de rede especializados. Um exemplo clássico dessa aplicação é a descarga da interconexão virtualizada de máquinas virtuais para smartNICs, o que pode reduzir a utilização da CPU para funções de rede em até 70%, permitindo a alocação de mais cargas de trabalho para a mesma CPU (YAN *et al.*, 2020). Surge, então, a necessidade de determinar quais funções são mais eficientes em placas de aceleração e quais são mais adequadas para as CPUs.

2.5 Algoritmos de *hash* e cifras de fluxo

Algoritmos de resumo (*hash*) são funções que criam um resumo de tamanho fixo que representa um conjunto maior de dados. Uma propriedade esperada de um algoritmo de hash é a mesma entrada criar uma mesma saída de tamanho pré-definido. Hashes são comumente utilizados para verificar a integridade de grande massas de dados, visto que proveem uma alta probabilidade de os dados não terem sido alterados caso o resultado da função seja igual a um hash destes dados anteriormente calculado.

Mais especificamente, algoritmos de *hash* criptográfico são uma classe de *hashes* que possui algumas propriedades que são úteis para cenários de segurança: é necessário que seja computacionalmente inviável encontrar uma outra entrada que gere a mesma saída (resistência à segunda pré-imagem), que não seja possível recuperar o conjunto de dados original dada a saída da função (resistência à pré-imagem) e que não seja viável encontrar dois valores que produzam o mesmo resumo (resistência à colisão). (ROGAWAY; SHRIMPTON, 2004)

Diversos algoritmos de *hash* foram criados utilizando construções diferentes, sendo que alguns não são mais considerados seguros pelo *National Institute of Standards and Technology* (NIST) dos Estados Unidos (NIST, 2022), como a família *Message Digest* (MD2, MD4 e MD5) (RIVEST, 1992) e o algoritmo SHA-1 (3RD; JONES, 2001) da família *Secure Hash Algorithm* (SHA-1, SHA-2 e SHA-3). Outros algoritmos de resumo criptográfico também foram criados, como a família BLAKE (BLAKE, BLAKE2 e BLAKE3) (AUMASSON *et al.*, 2008) e a família SHAKE (SHAKE128 e SHAKE256) (NIST, 2015).

Os algoritmos das famílias MD e os algoritmos SHA-1 e SHA-2 são baseados em uma construção chamada Merkle-Damgård (DAMGÅRD, 1989), onde um conjunto de vetores de inicialização (*Initialization Vectors*, IVs) são operados iterativamente por diversas rodadas com partes dos dados a serem resumidos, chegando a um valor de saída que é igual à soma do tamanho dos vetores iniciais. Já os algoritmos da família BLAKE são baseados em uma construção chamada *HAsh Interative FrAmework* (HAIFA) (BIHAM; DUNKELMAN, 2007), onde um número aleatório público (*salt*) e um contador de bits é adicionado às rodadas de processamento dos IVs junto aos dados de entrada, aumentando

a resistência a certos tipos de ataques. Por fim, a família SHA-3 (NIST, 2015) e SHAKE são baseados em uma construção chamada Keccak (BERTONI *et al.*, 2013), onde os dados são acumulados e posteriormente liberados em uma estrutura chamada de *Sponge Structure*. Tanto o HAIFA quanto o Keccak permitem a geração de um número arbitrário de bits de saída, sendo chamados de *eXtended Output Functions* (XOF).

Para os processos de SB (inicialização segura) e RA (atestação remota) são utilizados algoritmos de hash criptográfico. O processo de SB utiliza hashes criptográficos para realizar a medição de cada componente do processo de inicialização, garantindo que não houve modificações do próximo componente da cadeia de inicialização. Já o processo de RA utiliza o hash criptográfico para reduzir o tamanho da prova a ser apresentada para o verificador, permitindo a confirmação de toda a configuração esperada por meio da comparação do hash provido com o hash esperado.

As propriedades de algoritmos de hash criptográfico são interessantes para outro tipo de algoritmo: os algoritmos de cifra de fluxo. Em um algoritmo de cifra de fluxo, uma mensagem é cifrada por meio de uma operação de ou-exclusivo (XOR) com um material de chave (*Keystream*) gerado por meio de uma chave secreta, um nonce público e um contador público. Caso esse material de chave seja produzido com uma função com resistência à pré-imagem, não se repita e seja baseado em um valor secreto como uma chave, a cifra de fluxo é considerada segura.

Com isso, alguns algoritmos de cifra de fluxo usam uma construção similar a algoritmos de hash para gerar o keystream que será utilizado, visto que esses algoritmos produzem uma saída pseudoaleatória. (AUMASSON *et al.*, 2016)

Esse keystream pode ser descoberto em um ataque de texto escolhido, visto que um atacante pode recuperar um keystream válido se calcular o XOR da mensagem original com a mensagem cifrada. Com o keystream, o atacante pode forjar a autenticidade de um texto cifrado, aplicando a operação XOR da mensagem forjada com o keystream recuperado. Portanto, é necessário que um algoritmo de autenticação de mensagem (como o algoritmo Poly-1305) seja utilizado em conjunto à cifra de fluxo para garantir a autenticidade da mensagem enviada. (BERNSTEIN, 2005)

Alguns algoritmos de cifra de fluxo notórios foram propostos por Bernstein, como os algoritmos Salsa20 (BERNSTEIN, 2008b) e ChaCha20 (BERNSTEIN, 2008a). Esses algoritmos se caracterizam por produzir um keystream de 512 bits por meio da aplicação de operações de XOR, soma módulo 2^{32} e rotação circular de bits. Devido a essas operações, esses algoritmos são chamados do tipo "ARX" (*Add, Rotate and XOR*)

Recentemente, um outro algoritmo de cifra de fluxo foi proposto por Coutinho *et al.* chamado Forro14. Esse algoritmo possui o mesmo nível de segurança que o

algoritmo ChaCha20, mas com menos rodadas necessárias. Além disso, o Forro14 apresenta uma maior vazão de dados cifrados em cenários restritivos à paralelização de operações. (COUTINHO *et al.*, 2023b)

O leitor interessado pode consultar as referências para detalhes de funcionamento dos algoritmos citados e suas provas de segurança. Os detalhes de funcionamento relevantes para a compreensão da solução proposta serão apresentados em momento oportuno no Capítulo 3.

2.6 Trabalhos relacionados

No levantamento bibliográfico realizado, não foram encontrados trabalhos que utilizem a mesma abordagem desta pesquisa, isto é, utilizar o plano de dados programável como um verificador de dispositivos. No entanto, há propostas que realizam a RA de equipamentos de rede programáveis, bem como propostas que aumentam a escalabilidade do processo de RA utilizando, ou não, a CRA (atestação remota coletiva).

Dentre as propostas que buscam realizar a atestação de equipamentos de rede programáveis, podemos destacar o trabalho de Jacquin *et al.*, que traz uma proposta de atestação do boot e das regras inseridas em um *switch* SDN baseado no protocolo OpenFlow para verificação de integridade. Nesse artigo, o *switch* possui um módulo TPM embarcado que realiza o processo de medição de boot e a assinatura do estado das regras SDN instaladas para um "verificador SDN" avaliar a conformidade. (JACQUIN *et al.*, 2015)

O artigo de Sultana *et al.* propõe uma expansão da arquitetura PISA (*Protocol-Independent Switch Architeture*) para que suporte primitivas de RA, como gerar, assinar e verificar uma prova. A proposta cria uma cadeia de confiança baseada no caminho de travessia de um pacote, atestando cada equipamento de rede programável para garantir integridade de operação da rede. Esse artigo também propõe uma linguagem de definição de política para atestação. (SULTANA *et al.*, 2022)

Já o artigo de Conti *et al.* propõe uma arquitetura de alto-nível onde equipamentos SDN seriam capazes de atestar dispositivos IoT conectados a ele (CONTI *et al.*, 2019). O artigo aborda equipamentos com ASICs fixos configuráveis pelo protocolo OpenFlow e não faz menção a um método prático de se viabilizar essa atestação.

O artigo de Diop *et al.* traz uma forma de se realizar uma atestação escalável de diversos dispositivos utilizando agregação de assinaturas (DIOP *et al.*, 2020). Um problema em aberto do artigo é a escalabilidade relacionada ao verificador, visto que este é centralizado para toda a atestação realizada. Embora o processo de verificação das

assinaturas agregadas seja mais rápido do que o processo de cada assinatura individual, essa carga ainda é centralizada em um único verificador.

Os artigos de Banks *et al.* e Ambrosin *et al.* trazem uma análise de diversos métodos de RA e CRA, com premissas do modelo que servem como base para a elaboração da proposta deste trabalho (BANKS *et al.*, 2021) (AMBROSIN *et al.*, 2020). Já os artigos de Leão *et al.*, Heideker *et al.*, Madureira e Gao e Wang trazem exemplos de *offloading* para o plano de dados programável baseado em diversas plataformas (LEAO *et al.*, 2022) (HEIDEKER *et al.*, 2023) (MADUREIRA, 2021) (GAO; WANG, 2021).

Por fim, é importante citar artigos que trazem discussões de tecnologias habilitadoras para a RA. O artigo de Yoo e Chen traz uma implementação em PDP (plano de dados programável) do algoritmo "Half SipHash" para resumo criptográfico, que pode ser explorado para viabilizar atestações mais eficientes em PDP (YOO; CHEN, 2021). O artigo de Yoshinaka *et al.* traz uma discussão de implementação do algoritmo de cifra de fluxo "ChaCha" em plano de dados programável, que também é uma opção para a viabilização da atestação (YOSHINAKA *et al.*, 2022).

2.7 Considerações do capítulo

Neste capítulo foram apresentados os fundamentos necessários para a compreensão deste trabalho por meio de um levantamento realizado na literatura. Foi apresentado o conceito de atestação remota e como é feita sua utilização em grandes provedores de serviço de nuvem como AWS, Microsoft Azure e Google Cloud Platform.

Além disso, foram mostradas as mudanças de paradigmas de redes de computadores que ocorreram nos últimos anos e como esses novos paradigmas podem ser explorados para a solução do problema identificado, como o uso da descarga de processamento para a rede programável.

Foram mostrados também conceitos de algoritmos de resumo criptográfico e cifras de fluxo, que podem ser aplicados na técnica de atestação remota para a geração da prova de conformidade e sua verificação.

Por fim, foram mostrados trabalhos relacionados no estado da arte: tanto na parte de atestação remota, quanto na parte de offloading de operações para a rede. Foi discutido que essa abordagem de atestação remota utilizando plano de dados programáveis ainda não foi explorada na literatura e que pode ser compreendido como uma viabilização da proposta conceitual de atestação baseada em redes SDN (CONTI *et al.*, 2019).

No próximo capítulo será apresentada uma proposta de nova forma de realizar atestação remota que procura aproveitar os pontos fortes desta técnica, ao mesmo tempo

em que tenta minimizar os problemas que foram levantados em trabalhos anteriores. Também serão detalhados o esquema proposto e o modelo adversarial.

3 P4DRA: *P4-based Distributed Remote Attestation*

Neste capítulo serão apresentados os detalhes da proposta P4DRA (*P4-based Distributed Remote Attestation* ou Atestação Remota Distribuída baseada em P4), incluindo seu protocolo, algoritmo, requisitos técnicos fundamentados na literatura e o modelo adversarial considerado para o esquema de atestação remota neste cenário. A partir dessas informações, será detalhada a implementação realizada e os resultados coletados nos capítulos posteriores.

3.1 Esquema proposto

Será apresentado o esquema proposto para a atestação remota em data centers, para que possa ser descrito posteriormente à discussão sobre o atendimento dos requisitos levantados e da forma de mitigação do modelo de ataque considerado. Nesse cenário, será realizada a atestação de todos os dispositivos da rede de forma periódica, em contraste à realização de uma atestação sob demanda previamente à alocação de uma carga de trabalho, como realizado pela Google e descrito na Seção 2.2.

Essa abordagem busca garantir a conformidade de toda a infraestrutura com rápida identificação de nós inconsistentes para que ações corretivas possam ser iniciadas o mais breve possível. A atestação sob demanda identifica um dispositivo comprometido ou inconsistente somente antes da alocação de uma carga de trabalho, o que pode permitir a existência de um intervalo de tempo suficiente para um ataque.

O esquema é composto por um protocolo de comunicação, um conjunto de algoritmos e alguns atores principais dos contextos de RA e plano de dados programável. Podemos considerar como atores (i) o *switch* programável que fará o papel de verificador, (ii) o "mecanismo de atestação" existente em um dispositivo (como um servidor do data center) que será o provador e (iii) o controlador SDN que será a interface do operador da rede para configuração do contexto de atestação com políticas e parâmetros, distribuição de chaves criptográficas e solicitações de provas de atestação. Serão utilizados 4 algoritmos para o procedimento: *setup*, solicitação, verificação e reação. A Figura 3.1 ilustra estes algoritmos e as definições formais serão apresentadas na seção 3.3.

O Algoritmo de *Setup* é iniciado pelo operador da rede para fazer a configuração do contexto de atestação. Assume-se a existência de um canal seguro inicial

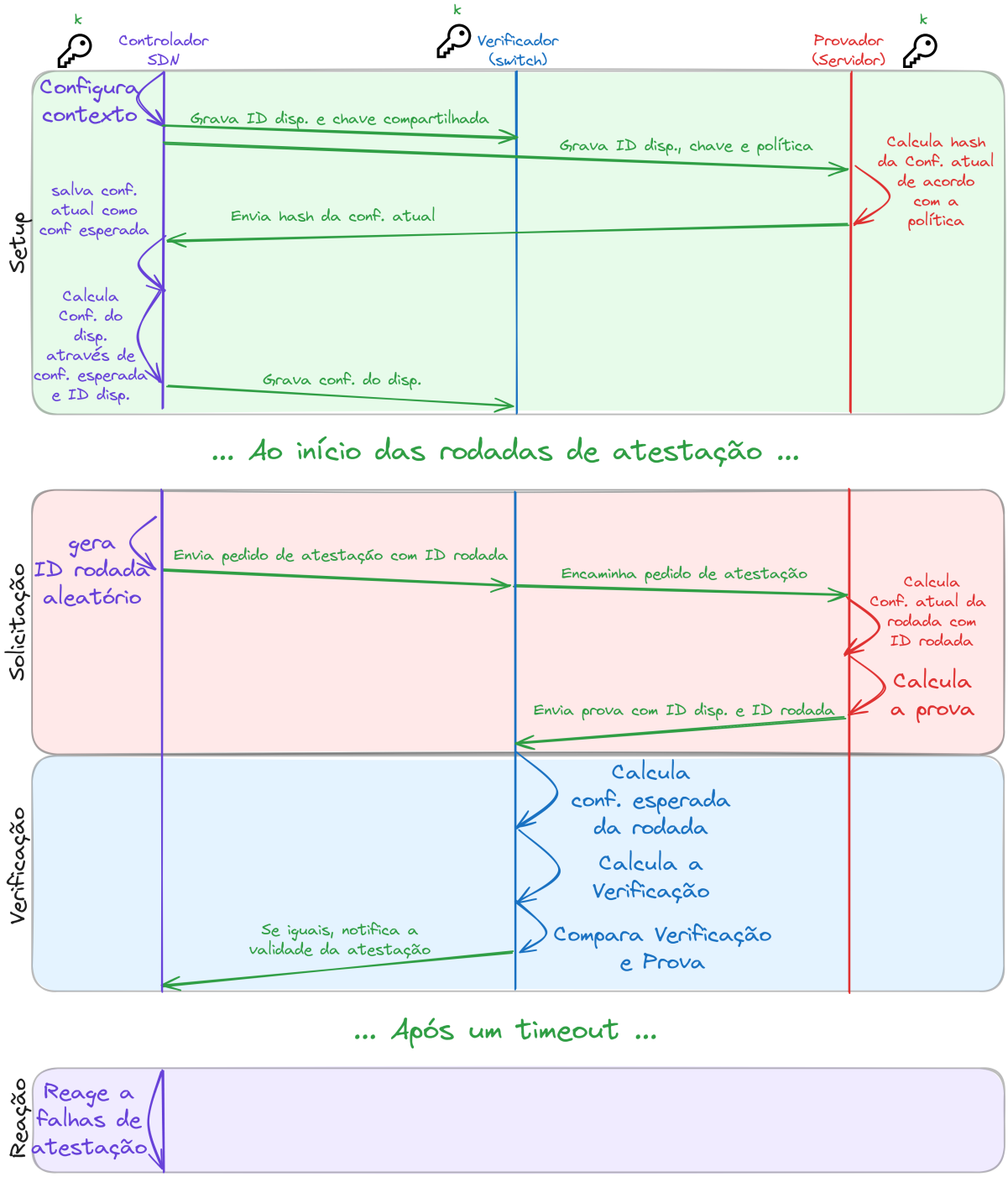

Figura 3.1 – Algoritmos de atestação da técnica proposta.

(em tempo de configuração) entre o controlador e cada mecanismo de atestação dos dispositivos listados, para a distribuição de chaves secretas a serem gravadas na memória segura de cada dispositivo. Esse canal seguro pode ser configurado utilizando chaves pré-compartilhadas no momento da instalação do servidor na infraestrutura, ou através de algoritmos de acordo de chaves como Diffie-Hellman e de um processo de provisionamento automático como *Secure Zero Touch Provisioning* (SZTP) (WATSEN *et al.*, 2019).

O controlador guarda o *hash* da configuração esperada do dispositivo a ser

verificado a cada rodada. O controlador grava no *switch* o hash da configuração do dispositivo calculado utilizando um identificador único do dispositivo (Id_dispositivo) e associa a esse Id a chave secreta compartilhada com o provador.

O Algoritmo de Solicitação é iniciado pelo controlador com um número aleatório (*nonce*) chamado "Id_rodada". Utilizando o protocolo de comunicação, o controlador envia a cada *switch* o pedido de atestação informando o nonce da rodada, que deve ser repassado aos mecanismos de atestação conectados ao *switch*, para que possam calcular a prova de atestação e devolvê-la ao *switch*.

O Algoritmo de Verificação é iniciado quando o *switch* recebe a prova de um mecanismo. O *switch* calcula uma prova utilizando os mesmos parâmetros da rodada, mas utilizando a configuração esperada. Caso as provas calculadas sejam iguais, o *switch* encaminha ambas as provas para apreciação do controlador. Caso sejam diferentes, o *switch* descarta o pacote, fazendo com que o controlador tome uma ação de reação.

O Algoritmo de Reação é iniciado pelo controlador ao não receber uma atestação válida vinda de um *switch* dentro de um *timeout*. Neste caso, a ação tomada pelo *controlador* é definida pelo operador da rede dentro de um conjunto pré-definido de ações possíveis programadas no controlador. Por exemplo, o controlador pode bloquear conexões daquele dispositivo ao *switch*, notificar o operador da inconsistência e marcar o dispositivo como comprometido para o plano de controle.

Há algumas considerações quanto à proposta: como a linguagem P4 não possui estruturas de repetição, a prova de verificação pelo *switch* deve ser calculada utilizando recirculação de pacotes. Há algoritmos de *hash* que são baseados em operações simples, como *BLAKE3* (O'CONNOR *et al.*, 2020) e *Half SipHash* (YOO; CHEN, 2021), que podem ser utilizados no lugar de algoritmos como os da família SHA em planos de dados programáveis. O mecanismo de atestação deve ser construído dentro de diversas restrições que serão elencadas nos requisitos e no modelo de ataque.

No caso de componentes desagregados de um único dispositivo que possuam mecanismos de atestação próprios, é possível fazer a agregação da configuração de todos os componentes do dispositivo em uma única prova. Nesse caso, o controlador deve definir a política de atestação do dispositivo de forma que sejam consideradas informações de todos os componentes. Este mesmo cenário de agregação pode ser utilizado para dispositivos coletivos, isto é, onde não há a necessidade de se verificar individualmente cada dispositivo que compõe a agregação, de forma a reduzir a quantidade de provas a serem verificadas pelo *switch*.

3.2 Requisitos técnicos

Os requisitos para o protocolo proposto foram adaptados das premissas elencadas no trabalho de Banks *et al.* (BANKS *et al.*, 2021). Esses requisitos foram elaborados para um protocolo de CRA, mas podem ser aplicados como uma verificação escalável para muitos equipamentos, mesmo que essa seja feita de forma granular ou distribuída. Nesta seção são apresentados os requisitos e a abordagem para cumpri-los é detalhada no Capítulo 4.

Informação atualizada: a atestação deve refletir a informação referente à última verificação. Isto é, não deve ser possível responder a uma atestação com informações desatualizadas do estado do dispositivo.

Informação compreensiva: a informação contida na atestação deve permitir o verificador atestar o estado do dispositivo.

Mecanismo confiável: o verificador deve ser capaz de confiar na informação de um provador mesmo que haja um adversário ativo na rede.

Acesso exclusivo: apenas os agentes envolvidos na atestação devem possuir acesso à chave secreta.

Sem vazamentos: o processo de atestação não pode vazar nenhuma informação sobre a chave secreta para o atacante.

Imutabilidade: o mecanismo de atestação não pode ser modificado por um atacante ativo ou passivo.

Execução atômica: a execução do mecanismo de atestação não pode ser interrompida por nenhuma ação no dispositivo até sua conclusão. No caso de interrupção, ela deve ser retomada desde o início, como se nunca tivesse sido iniciada.

Chamada controlada: o mecanismo de atestação só pode ser chamado pelo *entrypoint* esperado.

Eficiência: o protocolo de atestação deve ser mais eficiente do que atestar todos os dispositivos, um de cada vez.

Agregação de resultados: deve existir um mecanismo para agregar os resultados de cada verificação individual.

Topologia de Atestação: deve haver uma estrutura eficaz para atravessar a topologia de dispositivos (Spanning-Tree, Mesh ou Pub/Sub, por exemplo).

Tolerância a falhas: deve ser definido se o protocolo é resistente a modificações de topologia e se dispositivos podem sair e entrar do sistema de atestação sem afetar seu

funcionamento correto.

Atestação de controle de fluxo: deve haver indicação se o protocolo suporta controle de fluxo, isto é, se protocolo pode mudar seu comportamento em tempo de execução, sem alterar os binários presentes. Portanto, é preciso atestar também a ordem de instruções presentes na memória para garantir integridade.

3.3 Detalhamento do protocolo proposto

O protocolo P4DRA (*P4-based Distributed Remote Attestation*) é responsável por viabilizar o processo de atestação remota de forma distribuída utilizando o plano de dados programável como verificador intermediário entre um plano de controle e os provadores sob domínio do operador. Para um melhor entendimento do protocolo, os quatro algoritmos descritos na Seção 3.1, bem como as três funções básicas usadas pelos mesmos, são detalhados a seguir.

- $L(x)$: retorna o comprimento em bits de x;
- $A(c_i, d_i, r, n)$: retorna uma variação ($c_{i,r}$) do *hash* da configuração (c_i) do provador (i) para a rodada atual de atestação (r), dada por:

$$c_{i,r} = A(c_i, d_i, r, n) = (c_i[3] \oplus r) \| (c_i[2] \oplus d_i[1]) \| (c_i[1] \oplus n) \| (c_i[0] \oplus d_i[0]), \quad (3.1)$$

  onde:

  - c_i: *hash* de l bits da configuração do provador dividido em 4 partes de tamanhos iguais $c_i[j], j = 0, 1, 2, 3$, onde $c_i[0]$ representa os bits menos significativos e $c_i[3]$, os mais significativos do *hash*;
  - d_i: ID de $l/2$ bits do provador, definido pelo controlador e dividido em 2 partes de tamanhos iguais $d_i[j], j = 0, 1$, onde $d_i[0]$ representa os bits menos significativos e $d_i[1]$, os mais significativos;
  - r: contador de $l/4$ bits que identifica a rodada de atestação atual;
  - n: nonce aleatório de $l/4$ bits utilizado para adicionar imprevisibilidade aos parâmetros da rodada atual de atestação. Este nonce aleatório pode ser gerado utilizando um gerador de números pseudoaleatórios (*Pseudorandom Number Generator*, PRNG). Para o caso desta implementação, o nonce é gerado utilizando um carimbo de tempo de precisão de nanossegundos, visto que este apresenta uma quantidade de aproximadamente 2^30 valores possíveis a cada segundo, sendo suficiente para o cenário de atestação. r e n podem ser definidos com tamanhos diferentes, desde que a soma de seus comprimentos em bits resulte em $l/2$;

- $O(o, \hat{h}, m)$: retorna um *hash* da configuração do provador (c_i), onde:
  - o: define a política de composição da entrada m sobre a qual deverá ser calculado o hash;
  - $\hat{h}$: função de hash a ser utilizada sobre a entrada m;
  - m: texto de entrada para a função de hash, podendo o mesmo ser composto por *hashes* de arquivos e parâmetros de configuração do provador a serem verificados;

Os algoritmos apresentados na Seção 3.1 são descritos a seguir juntamente com seus parâmetros utilizados.

- Algoritmo de Setup:
  - operador da rede define:
    - período em segundos entre rodadas de atestação (T);
    - função de *hash* (h) responsável por calcular a prova de atestação;
    - função de *hash* $\hat{h}$ responsável por calcular o *hash* da configuração (c_i) do provador. A mesma função de *hash* pode ser utilizada para h e $\hat{h}$, mas h será executada em planos de dados programáveis e $\hat{h}$ será executada no provador. Portanto, recomenda-se que h e $\hat{h}$ sejam funções eficientes para seus respectivos ambientes de execução;
    - conjunto de dispositivos a serem atestados (I);
    - conjunto de chaves ($\mathbf{K}$), tal que $|\mathbf{K}| = |\mathbf{I}|$, ou seja, cada dispositivo possui uma chave, sendo k_i a chave compartilhada com o dispositivo i;
    - conjunto de políticas de configuração ($\mathbf{O}$), podendo uma política ser utilizada por mais de um dispositivo e o_i indicando a política definida para o dispositivo i. A política define quais arquivos e parâmetros devem ser verificados na atestação;
    - conjunto de *hashes* de configurações dos provadores ($\mathbf{C}$), tal que:
      - $|\mathbf{C}| = |\mathbf{I}|$, ou seja, haja um hash de configuração para cada dispositivo;
      - c_i seja um *hash* da configuração do dispositivo i calculado a partir de o_i, sua política definida; e
      - $L(c_i) = L(k_i)$, ou seja, o *hash* da configuração e chave do dispositivo tenham o mesmo comprimento em bits;
  - controlador envia a cada dispositivo (i):
    - ID dispositivo (d_i), tal que $L(d_i) = L(c_i)/2$;

  - ∗ Chave do dispositivo (k_i);
  - ∗ Política de configuração do dispositivo (o_i);
- controlador grava no *switch* cada chave k_i indexada pelo respectivo identificador d_i de maneira a viabilizar uma busca futura.

O Algoritmo de Setup é ilustrado na Figura 3.2.

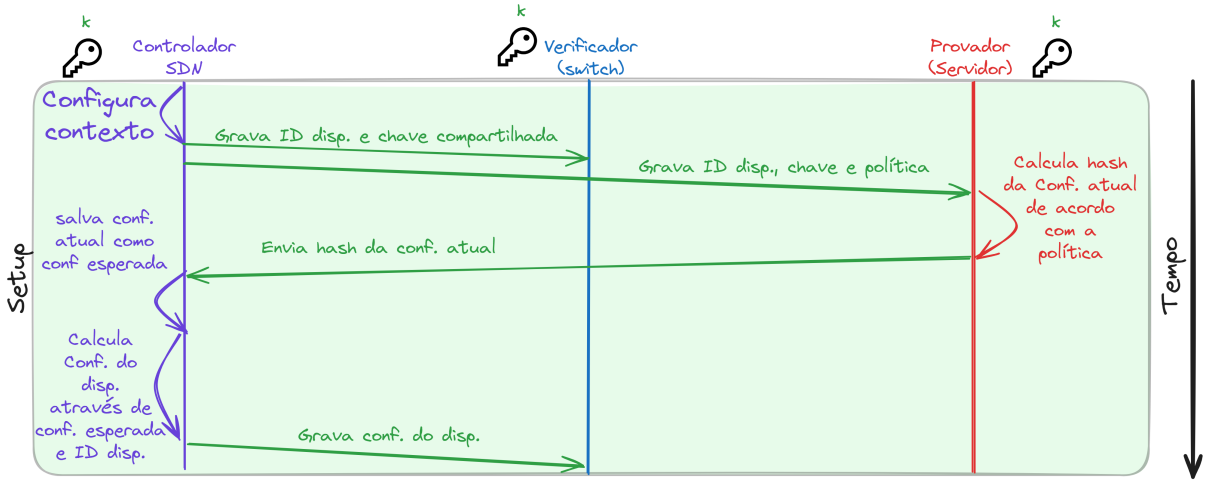

Figura 3.2 – Algoritmo de *Setup* da técnica proposta.

- Algoritmo de Solicitação:
  - controlador gera:
    - ∗ ID da Rodada (ρ), um valor monotônico, tal que $L(\rho) = L(c_i)/2$. ρ é a concatenação de um contador (r) e um nonce (n), tal que $L(n) = L(r) = L(c_i)/4$;
  - controlador envia um pacote de solicitação de atestação ao *switch* com o parâmetro ρ a cada T segundos;
  - *switch* encaminha aos dispositivos conectados a ele um pacote de solicitação de prova com o valor ρ;
  - mecanismo de atestação, residente na memória segura do provador, calcula:
    - ∗ $c_{i,r}$ dada pela função $A(\hat{c}_i, d_i, r, n)$ onde $\hat{c}_i$ representa o hash da configuração atual obtida de $O(o_i, \hat{h}, m_i)$;
    - ∗ $p_{i,r} = h(c_{i,r}, k_i)$, onde $p_{i,r}$ é a prova do dispositivo i na rodada r;
  - O mecanismo envia para o *switch* um pacote de resposta com d_i, $c_{i,r}$ e $p_{i,r}$.

O algoritmo de Solicitação é ilustrado na Figura 3.3.

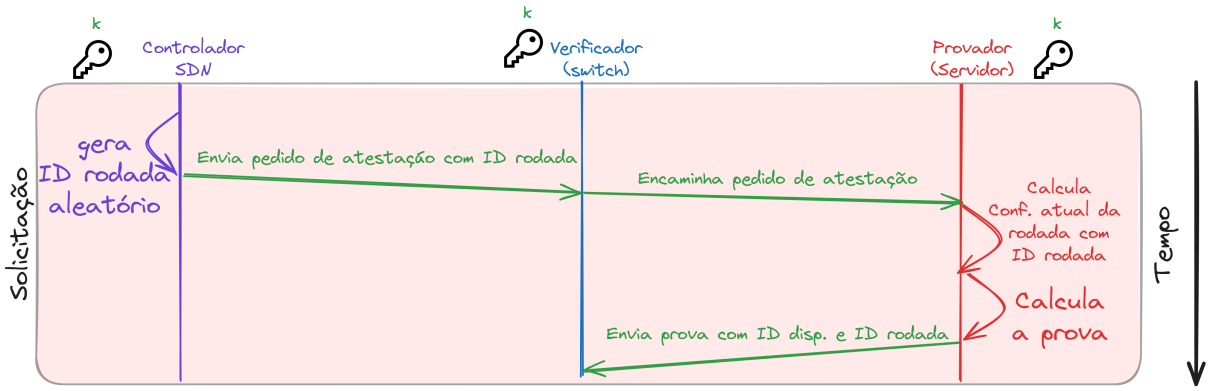

Figura 3.3 – Algoritmo de Solicitação da técnica proposta.

- Algoritmo de Verificação:
  - *switch* recebe d_i, ρ e $p_{i,r}$ do dispositivo i.
  - *switch* busca k_i a partir de d_i;
  - *switch* calcula $c_{i,r}$, dada por $A(c_i, d_i, r, n)$, utilizando c_i informado pelo controlador e os parâmetros recebidos na resposta da prova. Para facilitar o cálculo nas rodadas, o controlador pode gravar no *switch* o valor de c_i dado pela equação 3.2, visto que c_i e d_i não variam entre rodadas (a não ser que a configuração esperada mude), de forma que o *switch* precise apenas calcular $(c[3] \oplus r)$ e $(c[1] \oplus n)$ para obter $c_{i,r}$;

$$c_i = c_i[3] \| (c_i[2] \oplus d_i[1]) \| c_i[1] \| (c_i[0] \oplus d_i[0]) \tag{3.2}$$

  - *switch* calcula: $v_{i,r} = h(c_{i,r}, k_i)$ onde $v_{i,r}$ é a prova esperada do dispositivo i na rodada r;
  - *switch* verifica se $p_{i,r} \oplus v_{i,r} = 0$. Caso seja 0, o *switch* envia ao controlador um pacote de resposta com $d_i, r, p_{i,r}$ e $v_{i,r}$ através do canal dedicado de comunicação. Caso seja $\neq 0$, então o *switch* descarta o pacote de verificação. Embora seja intuitivo que o *switch* alerte o controlador apenas atestações que falharam, é custoso ao controlador gravar no *switch* o ID de rodada vigente, necessitando que o controlador mantenha o controle de rodadas de atestações. Com isso, o *switch* deve informar ao controlador as atestações válidas, para que este último possa manter o histórico de atestações dos dispositivos. Ainda assim, o cálculo da verificação, processo mais computacionalmente custoso, é realizado pelo *switch*. Além disso, visto que o *switch* não possui um controle da rodada de atestação atual, um provador malicioso pode enviar muitas provas inválidas de atestação (a exemplo, com $p_{i,r}$ sendo um valor nulo), para obrigar o *switch* a calcular muitas verificações. Caso os *switches* da rede encaminhem

estas verificações de provas inválidas para o controlador, pode ocorrer um ataque distribuído de negação de serviço (*Distributed Denial of Service*, DDoS) no controlador.

O Algoritmo de Verificação é ilustrado na Figura 3.4.

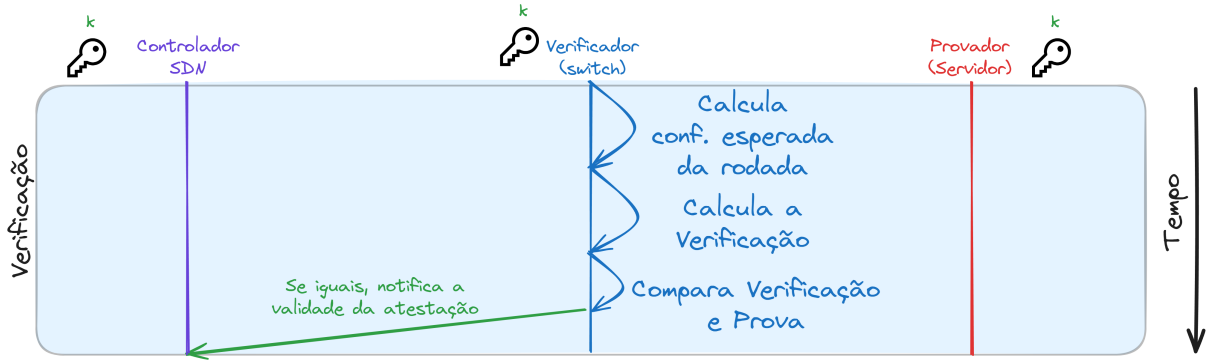

Figura 3.4 – Algoritmo de Verificação da técnica proposta.

Esse processo de solicitação e verificação realizado entre um provador e um verificador é ilustrado na Figura 3.5. É importante notar que o controlador não precisa

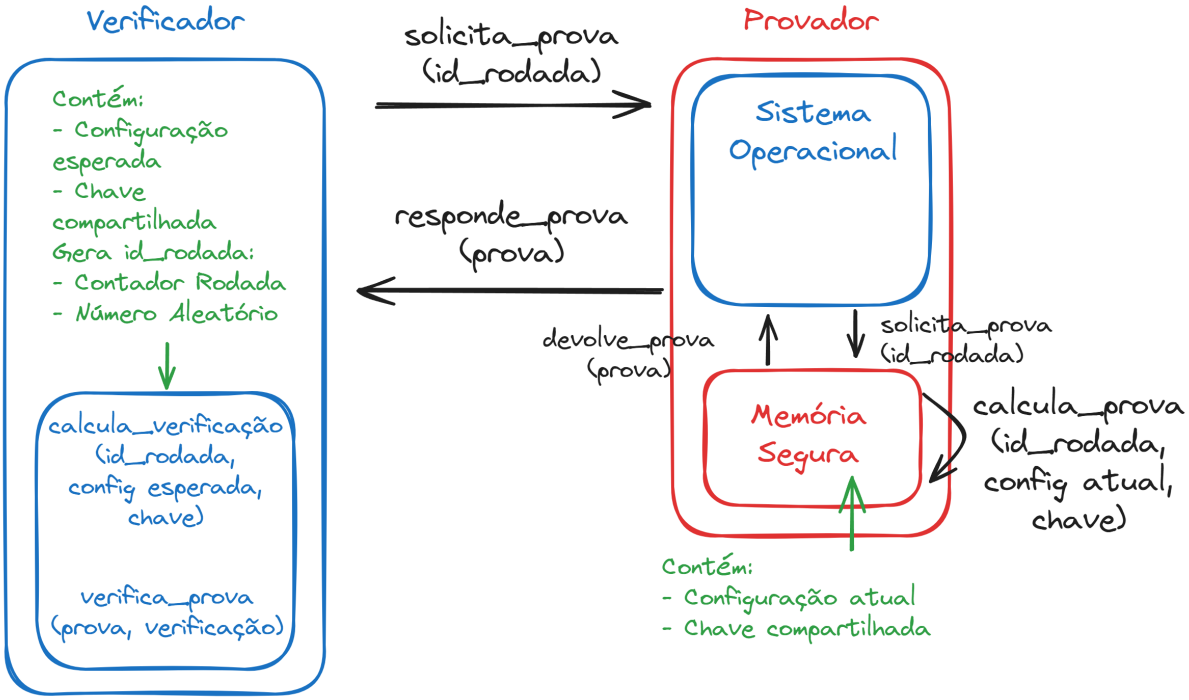

Figura 3.5 – Atestação remota no protocolo proposto.

realizar o cálculo de nenhum *hash*, mas apenas fazer operações de XOR ($\oplus$) e gerar números aleatórios (n). O *hash* da configuração esperada do dispositivo (c_i) pode ser solicitado pelo controlador ao mecanismo quando este estiver em um estado considerado íntegro e replicável (isso é, no caso de um *reboot*, o dispositivo pode retornar ao mesmo

estado esperado). Além disso, esse cálculo de XOR só precisa ser realizado a cada troca de configuração esperada de um dispositivo.

Com o protocolo definido, o próximo passo é a definição de tecnologias a serem utilizadas para sua implementação e avaliação, conforme descrito na próxima seção.

3.4 O algoritmo de cálculo de prova

Para este trabalho de pesquisa, foi utilizado um algoritmo de cifra de fluxo no lugar de um algoritmo de *hash* tradicional para cálculo da prova de atestação. O uso da cifra de fluxo implementada em plano de dados programável permite que a mesma porção do pipeline em um *switch* possa ser utilizada tanto para a operação de atestação quanto para a operação de cifração de tráfego.

Algoritmos de cifra de fluxo modernos possuem a característica de gerar um fluxo de bits (*keystream*) para a operação de XOR com a mensagem em claro a fim de gerar um texto cifrado, ou vice-versa. A função que gera o *keystream* em cifras de fluxo possui características similares a um *hash* criptográfico por ser uma função pseudoaleatória (*Pseudorandom function*, PRF) e possuir resistência à pré-imagem.

A escolha de algoritmos de cifra de fluxo ao invés de algoritmos de *hash* tradicionais é decorrente destes últimos necessitarem de mais rodadas e/ou mais dados para prover as propriedades de interesse para provas de atestação. Esse maior número de rodadas impacta a vazão obtida por *switches* programáveis e a maior quantidade de dados para cálculo do resumo impacta na escalabilidade da solução. Foram feitos estudos e implementações com diferentes algoritmos de cifra de fluxo, que apresentaram um bom desempenho, conforme será apresentado mais adiante.

Três algoritmos de cifra de fluxo modernos foram avaliados: Salsa20, ChaCha20 e Forro14. Em um cenário de cifração/decifração utilizando essas cifras de fluxo, são utilizados como entrada uma chave secreta (k) de 256 bits, um *nonce* público (n) de 64 bits, um contador público de blocos de cifração (t) de 64 bits e quatro constantes públicas (s) que somam 128 bits. Estes parâmetros são os mesmos utilizados na implementação de referência em linguagem C e na implementação em PDP. A junção desses parâmetros produz uma matriz de estado inicial de 512 bits, organizada em 16 elementos de 32 bits, para que seja calculado o *keystream* utilizado na operação de XOR da cifra de fluxo. Para cada conjunto de 512 bits a ser cifrado ou decifrado, um novo *keystream* de 512 bits deve ser calculado. Esse cálculo utiliza os mesmos parâmetros k, n e s, mas incrementa o valor do contador t para cada novo conjunto de 512 bits calculado.

O cálculo desse *keystream* é realizado a partir de funções que modificam os

elementos da matriz utilizando operações de adição, rotação e ou-exclusivo. Essas funções são chamadas de *Quarter Round Function* (QR), pois quatro execuções desta função compõem o cálculo de uma rodada do algoritmo. O algoritmo de cifra de fluxo que utiliza QRs é composto de conjuntos de 2 rodadas, pois os elementos da matriz processados em cada QR são extraídos das colunas da matriz em rodadas ímpares e das diagonais da matriz em rodadas pares. O número de rodadas define a segurança provida pelo algoritmo, sendo indicado ao final do nome deste (por exemplo, "ChaCha20" indica 20 rodadas e "Forro14" indica 14 rodadas). O algoritmo Forro14 possui um nível de segurança similar ao algoritmo ChaCha20, mas realizando seis rodadas a menos, devido a mudanças internas nas estruturas de cálculo.

O algoritmo ChaCha20 executa as QRs de forma paralela, pois cada QR utiliza elementos diferentes da matriz como parâmetro. Já o algoritmo Forro14 introduz uma co-dependência entre as QRs ao utilizar um elemento que foi modificado pela QR anterior, tornando o algoritmo sequencial e trazendo um novo desafio a implementações de alto desempenho.

3.4.1 Escolha do algoritmo de cifra de fluxo em plano de dados programável

O algoritmo ChaCha20 foi implementado e avaliado para o plano de dados programável (PDP) baseado na ASIC Tofino com o objetivo de prover sigilo para conjuntos de dados de 512 a 4096 bits em intervalos de 512 bits de tamanho. (YOSHINAKA *et al.*, 2022). A Figura 3.6 ilustra a implementação do algoritmo na arquitetura Tofino, onde existem dois pipelines de processamento de pacotes.

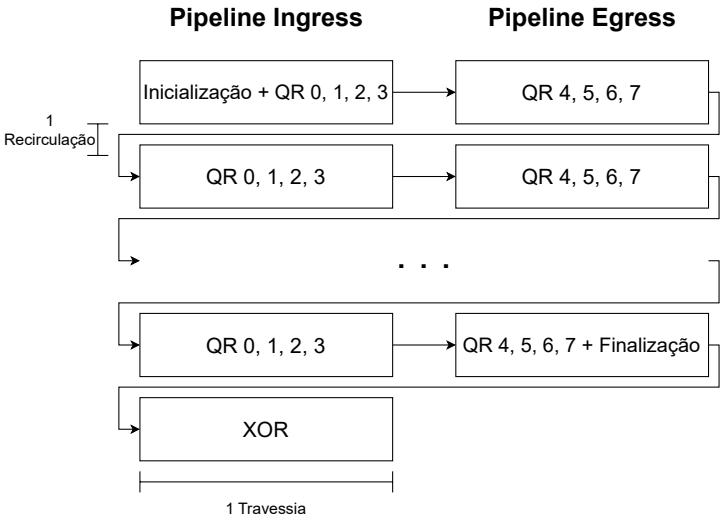

Figura 3.6 – Algoritmo ChaCha20 com paralelização de QRs em Plano de Dados Programável.

Para essa proposta de atestação, foi realizada uma implementação e avaliação

em PDP do algoritmo Forro14 em comparação ao algoritmo ChaCha20 para o caso de cifração de 512 bits (64B) de dados, um valor suficiente para o contexto de atestação remota. A Figura 3.7 ilustra a implementação do Forro14 na arquitetura Tofino.

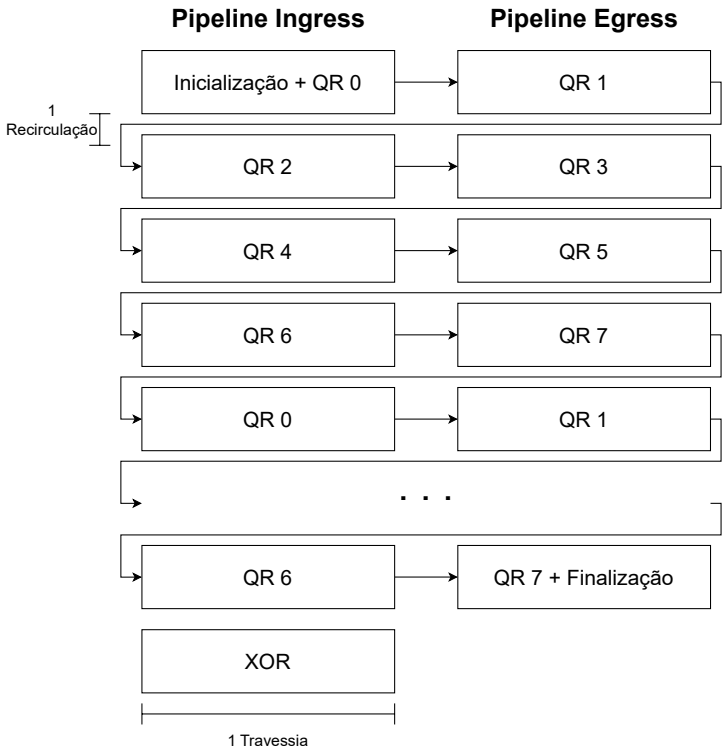

Figura 3.7 – *Design* do Algoritmo Forro14 em Plano de Dados Programável.

Devido à estrutura sequencial do algoritmo Forro14, sua implementação difere da implementação do algoritmo ChaCha20, projetado para a paralelização de cálculos, quanto ao uso de recursos e à vazão máxima de dados. Com essa diferença, a implementação do algoritmo Forro14 na arquitetura Tofino implica o uso de mais recirculações, o que reduz sua vazão máxima. Enquanto a implementação do algoritmo ChaCha20 precisa de 11 recirculações (ou 21 travessias de *pipeline*) , o algoritmo Forro14 precisa de 28 recirculações (ou 57 travessias de *pipeline*), pois pode realizar o cálculo de apenas 1 QR por travessia devido à dependência do resultado do QR anterior para o cálculo do QR atual, ocasionada pela técnica de polinização. No entanto, o fato de menos cálculos serem realizados de forma paralela implica em um menor uso de recursos para o mesmo nível de segurança. A Figura 3.8 compara as vazões de saída obtidas para cada vazão de entrada em relação ao algoritmo de cifra de fluxo utilizado, incluindo uma implementação do algoritmo ChaCha20 sem a paralelização de cálculos nos QRs. Já a Figura 3.9 ilustra um comparativo de uso dos recursos para cada um destes algoritmos. Os valores do gráfico representam a ocupação de cada recurso da arquitetura do *switch* programável em relação ao total disponível para construção do *pipeline*.

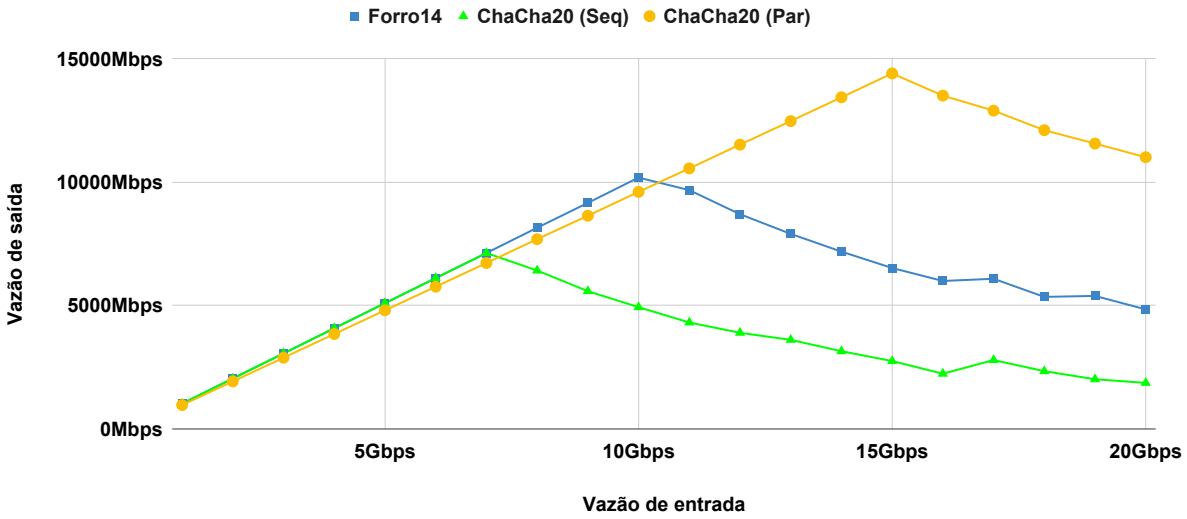

Figura 3.8 – Vazão de saída dos algoritmos de cifra de fluxo no *switch* Tofino para diversas vazões de entrada.

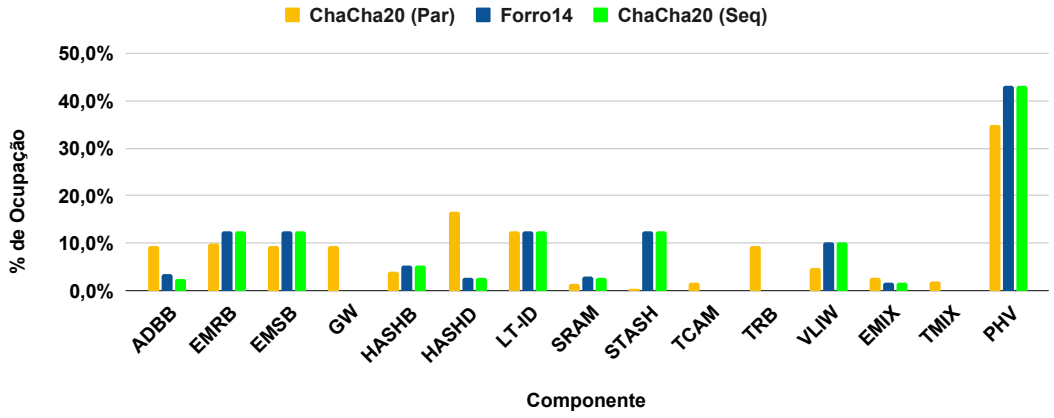

Figura 3.9 – Ocupação de recursos do Tofino pelos algoritmos de cifra de fluxo.

Como pode ser observado, o algoritmo Forro14 possui uma vazão máxima inferior ao algoritmo ChaCha20 paralelizado, mas maior em comparação ao algoritmo ChaCha20 sequencial. Ainda assim, a implementação do algoritmo Forro14 utiliza menos recursos do *switch* programável para prover o mesmo nível de segurança que o algoritmo ChaCha20, como pode ser visto nos parâmetros ADBB (*Action Data Bus Bytes*), HASHD (*Hash Distribution Unit*), EMIX (*Exact Match Input Crossbar*). O maior uso de EMRB (*Exact Match Result Bus*), EMSB (*Exact Match Search Bus*), HASHB (*Hash Bits*), SRAM (*Static RAM*) e *Stash* se dá pela priorização do uso de buscas exatas em tabelas em detrimento de buscas ternárias, visto que a SRAM é um recurso abundante no *switch*. Já o maior uso de VLIW (*Very Long Instruction Words*) e PHV (*Packet Header Vectors*) se dá pela maior quantidade de *actions* definidas para o processamento dos QRs, visto que a implementação do ChaCha20 utiliza apenas 24 *actions* diferentes (12 actions para colunas e 12 actions para diagonais), enquanto na implementação do Forro foi necessário

definir 12 actions para cada QR, totalizando 96 *actions* diferentes.

Para o esquema de atestação proposto, o *switch* precisa calcular apenas um *keystream* de 512 bits. Com isso, o algoritmo Forro14 apresenta uma opção mais econômica em recursos para o cálculo da prova de atestação em comparação ao algoritmo ChaCha20 em plano de dados programável, como pode ser visto em mais detalhes no Anexo A.

3.4.2 Adaptações para a atestação remota

No caso do esquema proposto para atestação, os parâmetros públicos (n, t e s) que compõem a matriz inicial são alterados, de forma a tornar o *keystream* gerado dependente dos parâmetros da rodada. Neste caso, os 256 bits de parâmetros públicos são trocados por um *hash* de 256 bits obtidos da configuração equipamento e da política definida de atestação.

Os 256 bits públicos são obtidos por meio do *hash* c_i calculado utilizando o algoritmo SHA-256 e os arquivos e parâmetros definidos na política o. Desses 256 bits obtidos, um XOR é feito com 4 parâmetros, conforme a função $A(c_i, d_i, r, n)$ descrita na Eq. 3.1.

Destaca-se, novamente, que os parâmetros da rodada não são secretos, visto que um atacante pode obtê-los observando as trocas de mensagens entre o *switch* e o mecanismo de atestação ou entre o controlador e o mecanismo de atestação.

Esse XOR é utilizado para que o controlador não precise calcular um *hash* por dispositivo a cada rodada, mas ainda torna a prova de atestação diferente para cada dispositivo e cada rodada, mesmo que a mesma chave e configuração sejam utilizadas em todos os dispositivos. Além disso, o XOR permite que a entrada do algoritmo de cifra de fluxo seja sempre 512 bits e que a função de hash da configuração possa ser qualquer, mesmo que gere menos que 256 bits, sem prejuízos à segurança do algoritmo de cifra de fluxo, visto que os parâmetros utilizados no XOR são públicos. No entanto, recomenda-se que sejam utilizadas funções de *hash* com saída de 256 bits, para reduzir a chance de colisão entre o *hash* da configuração esperada e o *hash* de uma configuração maliciosa do atacante.

O mesmo *nonce* pode ser reutilizado em rodadas de atestação diferentes se o ID da rodada for gerado por meio de um *timestamp* de 64 bits com precisão de nanossegundos, visto que este representa um valor monotônico e não é facilmente previsível, de forma que um atacante não possa solicitar a prova antecipadamente ao mecanismo de atestação para respondê-la quando solicitada.

Mesmo com conhecimento desses parâmetros públicos, o atacante não pode

calcular uma prova de atestação válida, pois não possui os 256 bits da chave secreta k presente na memória segura do mecanismo de atestação, implicando na necessidade de calcular e armazenar provas possíveis para 2^{256} possibilidades de k antecipadamente.

Da mesma forma, descobrir uma chave por meio de uma prova calculada implica quebrar a segurança do algoritmo de cifra de fluxo. A segurança dos algoritmos de cifra de fluxo citados aumenta conforme a quantidade de rodadas de cálculo do keystream, pois isso aumenta a complexidade de tempo e armazenamento para buscar a chave de 256 bits utilizada como parâmetro de entrada. De acordo com Coutinho, o melhor ataque atualmente conhecido possui complexidade de tempo e armazenamento de 2^{224} possibilidades para 7 rodadas do algoritmo ChaCha, enquanto o algoritmo Forro possui segurança similar a 20 rodadas do algoritmo ChaCha. (COUTINHO, 2023a)

3.5 Modelo adversarial

Para verificar formalmente a segurança de uma proposta, considera-se o modelo de ataque ao qual a proposta é submetida. Este modelo define as capacidades de um atacante, sua motivação, objetivos e as garantias que a solução deve prover contra este ataque. Tomando como base os modelos de ataques propostos no artigo de Ambrosin *et al.* (AMBROSIN *et al.*, 2020) e considerando que a proposta não deverá lidar com atacantes com capacidades que requerem acesso físico ao dispositivo, sendo essa proteção delegada ao controle de acesso do data center, considera-se um atacante com capacidade de software.

Para esquivar da verificação, um atacante tentaria interceptar a comunicação e falsificar uma prova válida $p_{i,r}$ por meio do conhecimento prévio da resposta esperada ($v_{i,r}$). Para evitar este tipo de ataque, o mecanismo deve possuir um segredo compartilhado com o verificador, como a chave secreta k, que não permita ao atacante se passar pelo mecanismo. Para evitar que o atacante possa computar a resposta correta antes da solicitação ocorrer, o resultado de cada atestação deve ser dependente de um fator probabilístico, como o *nonce* n. O modelo adversarial é ilustrado na figura 3.10.

Nesse cenário, o atacante tem a capacidade de comprometer um dispositivo ou injetar código malicioso no mesmo e limpar quaisquer traços de sua intrusão em tempo não-negligenciável, quando necessário. Isto permite que o atacante possa aproveitar janelas entre verificação e uso de um dispositivo para realizar a modificação de binários e configurações em um servidor e ter acesso a informações confidenciais ou modificar a execução esperada de uma carga de trabalho.

Além disso, o atacante é capaz de ler as comunicações entre verificador e

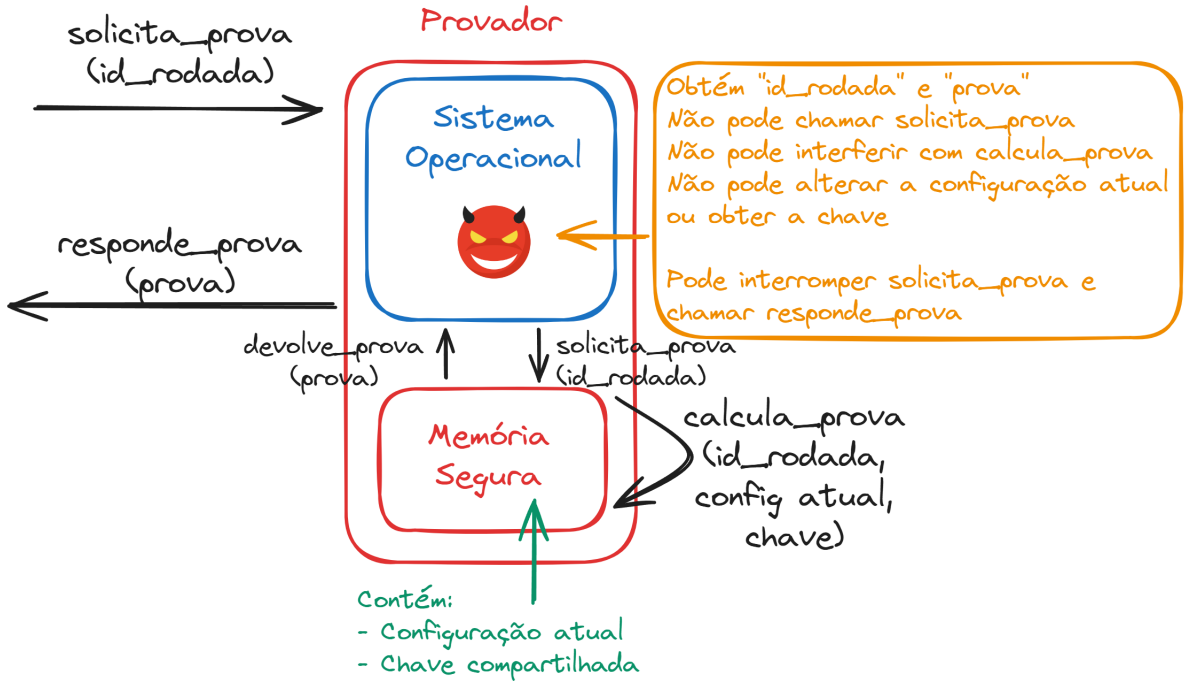

Figura 3.10 – Modelo adversarial considerado na proposta.

mecanismo de atestação, sendo capaz de conhecer o valor de ρ (id_rodada) e de $p_{i,r}$ (prova). No entanto, assume-se que o controlador possui um canal seguro de comunicação com o verificador e com o mecanismo, impedindo que o atacante possa obter a chave k_i nessas comunicações. Além disso, o atacante não é capaz de obter informações da memória do *switch* ou do controlador.

Caso o atacante consiga a chave k_i, ele será capaz de forjar a prova válida $p_{i,r} = h(A(c_i, d_i, r, n), k_i)$ e respondê-la ao *switch*, visto que conhece a configuração esperada c_i por ter obtido acesso a ela antes de comprometer o equipamento.

Esta abordagem de aproveitar a janela de tempo entre a atestação e o efetivo uso de uma plataforma computacional é chamada de "ataque *Time-of-Check-to-Time-of-Use*" ($TOCTTOU$). A técnica de atestação remota proposta torna o processo mais rápido, permitindo uma verificação mais frequente de acordo com o *speedup* alcançado, conforme será discutido no Capítulo 5, e possibilitando a detecção ocorrer antes do atacante esconder seus rastros.

3.6 Considerações do capítulo

Neste capítulo foi apresentada a solução de atestação remota em plano de dados programável, indicando o esquema, o protocolo e o algoritmo de cálculo de prova.

Para validação da solução, foi descrito o modelo adversarial identificado na

literatura e que pode ser aplicado ao contexto de data centers. Além disso, foram apresentados os requisitos técnicos relevantes para uma solução de CRA que também podem ser aplicados para o cenário proposto de atestação remota distribuída.

No próximo capítulo será detalhada a implementação da técnica proposta, abordando a codificação do protocolo na forma de um cabeçalho de pacotes de rede, a implementação do verificador em plano de dados programável e a implementação do provador utilizando uma tecnologia de memória segura.

4 Projeto e prototipação da proposta de atestação remota distribuída

Neste capítulo será detalhada a implementação realizada do protocolo de atestação remota distribuída em plano de dados programável, a qual está disponível no repositório público do Github[1].

O contexto de atestação remota é composto pelo verificador e pelo provador. Serão detalhadas a seguir as implementações do verificador em uma arquitetura de plano de dados programável e CPU de propósito geral. Também será detalhada a implementação do provador em uma plataforma com a tecnologia de memória segura.

Como descrito no Capítulo 3, será utilizada uma versão modificada do cálculo de *keystream* do algoritmo Forro14 para realizar o cálculo da prova de atestação e da verificação. Neste capítulo será explicado como os parâmetros para o cálculo da prova são transferidos entre controlador, verificador e provador.

4.1 Implementação do verificador em plano de dados programável

Como detalhado na seção 2.3, a implementação do verificador baseado em PDP é feita utilizando a linguagem P4 para definição de *pipelines* de processamentos de pacote em *switches* programáveis da arquitetura Intel Tofino. Além disso, são definidos cabeçalhos personalizados para o protocolo P4DRA que carregam informações de controle e os dados processados durante a solicitação e verificação de provas de atestação.

O cabeçalho básico do protocolo P4DRA (denominado "p4dra_oper") utiliza um campo "operação" de 8 bits, que indica a operação a ser realizada, um campo "id_rodada" de 64 bits e um campo "nonce" também de 64 bits que representam respectivamente os parâmetros r e n dos algoritmos de atestação propostos na Seção 3.3. Os valores possíveis para o campo "operação" são:

- 0x0: indica uma operação de **início**, onde o controlador envia o pacote ao *switch* para iniciar uma rodada de atestação, informando os campos id_rodada (r) e nonce (n);
- 0x1: indica uma operação de **solicitação**, onde o *switch* encaminha a solicitação de prova aos dispositivos. Nessa solicitação, o *switch* pode encaminhar a solicitação de

[1] https://github.com/regras/p4dra

diferente formas: via *unicast*, *multicast* ou *broadcast*;

- 0x2: indica uma operação de **resposta** do dispositivo ao *switch*, onde são adicionados os campos "id_dispositivo" (d_i) de 128b e a prova ($p_{i,r}$) de 512b;
- 0x3: utilizado internamente pelo *switch*, indica que uma verificação está sendo calculada por meio da função *hash* $h(A(c_i, d_i, r, n)\|k_i)$ para os parâmetros d_i, r e n recebidos e a configuração c_i gravada pelo controlador no *switch*;
- 0x4: indica uma verificação a ser encaminhada para o controlador caso uma prova fornecida não seja condizente com a verificação calculada. Nesse caso, são encaminhados r, n, d_i, a verificação calculada ($v_{i,r}$) e a prova $p_{i,r}$ recebida. Caso a verificação seja inválida, o *switch* descarta o pacote silenciosamente, evitando onerar o tráfego do controlador com provas inválidas;
- 0x5: enviado do controlador ao mecanismo de atestação do dispositivo, indica uma operação de *setup*. Esse pacote pode, idealmente, ter seu conteúdo cifrado, para que apenas o mecanismo de atestação possa recuperar seu conteúdo. Essa operação informa a chave compartilhada (k) e o id dispositivo (i) que devem ser utilizados pelo mecanismo para cálculo da prova $p_{i,r} = h(A(c_i, d_i, r, n)\|k_i)$.

Os cabeçalhos de solicitação (oper 0x0 e 0x1) são ilustrados na Figura 4.1, o cabeçalho de resposta (oper 0x2) é ilustrado na Figura 4.2, os cabeçalhos de verificação (oper 0x3 e 0x4) são ilustrados na Figura 4.3 e o cabeçalho de configuração (oper 0x5) é ilustrado na Figura 4.4.

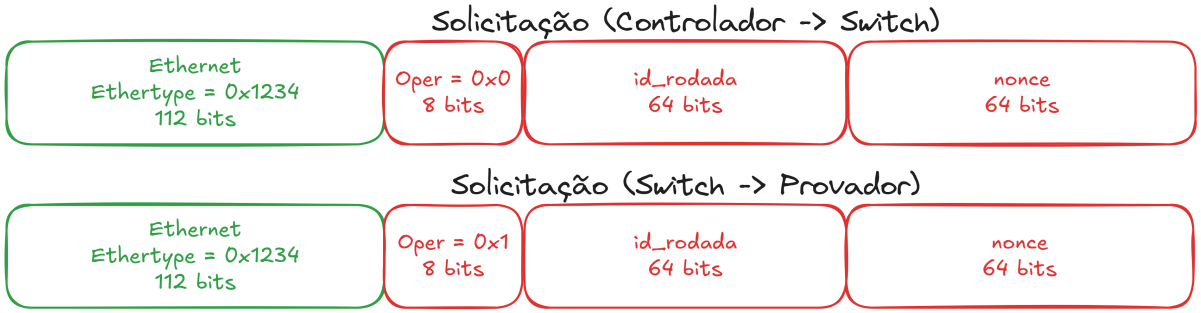

Figura 4.1 – Cabeçalhos de solicitação utilizados no protocolo P4DRA.

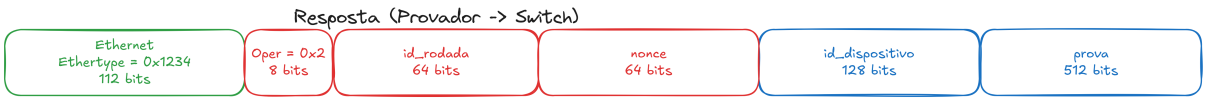

Figura 4.2 – Cabeçalho de resposta utilizado no protocolo P4DRA.

No caso do descarte de verificações inválidas pelo *switch*, um dispositivo é considerado como não atestado a não ser que o controlador seja notificado do contrário.

Figura 4.3 – Cabeçalhos de verificação utilizados no protocolo P4DRA.

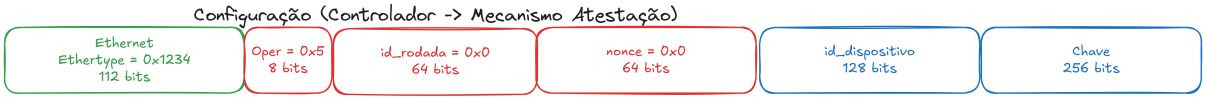

Figura 4.4 – Cabeçalhos de configuração utilizado no protocolo P4DRA.

Isso implica que o controlador e o *switch* devem possuir um canal de comunicação confiável, onde a indisponibilidade do *switch* também indique uma indisponibilidade neste canal, mostrando ao controlador que não será possível fazer a atestação dos dispositivos conectados a esse *switch*.

Quando um pacote de valor 0x0 chega ao *switch*, este encaminha aos dispositivos conectados a ele o pedido de atestação, alterando o valor do campo "operação" para 0x1. Ao receber um pacote com o campo operação com valor 0x1, uma aplicação executada no provador irá solicitar uma prova ao mecanismo de atestação por meio de uma chamada à memória segura, informando o *nonce* (n) e o id da rodada (r) a serem utilizados. O mecanismo de atestação utiliza a função $A(c_i, d_i, r, n)$ para calcular a configuração atual da rodada ($c_{i,r}$) e calcula a prova $p_{i,r} = h(\hat{c}_{i,r} \| k_i)$, retornando-o à aplicação no provador.

A aplicação, então, gera um pacote do protocolo P4DRA com o campo "Oper" (operação) com valor 0x2 e encaminha ao *switch* informando seu id de dispositivo (d_i) e a prova calculada ($p_{i,r}$). Ao receber esse pacote, o *switch* começa a calcular uma verificação utilizando como parâmetros a configuração esperada do dispositivo (c_i) e a chave do dispositivo (k_i) gravados pelo controlador e buscados pelo *switch* em suas tabelas a partir do id do dispositivo (d_i).

O *pipeline* elaborado para o protocolo é orientado pelo valor do campo "*Oper*" no cabeçalho do pacote recebido. Como demonstrado na Figura 4.5, o valor do campo irá definir o circuito (*Action*) a ser utilizado para processamento do pacote. No circuito de "Encaminhar pedido", o valor do campo Oper no cabeçalho é alterado para 0x1 e o pedido é encaminhado para o *MAC Address* de destino. Esse circuito também pode ser utilizado para que o *switch* encaminhe o pedido em formato de *multicast*, definindo um valor FF:FF:FF:FF:FF:FF no *MAC address* de destino e replicando o pacote para todas as portas de dispositivos conectados a ele. Essa funcionalidade de *multicast* não foi implementada nesta prova de conceito, devendo o controlador enviar um pedido de atestação para cada provador por meio de transmissão *unicast*.

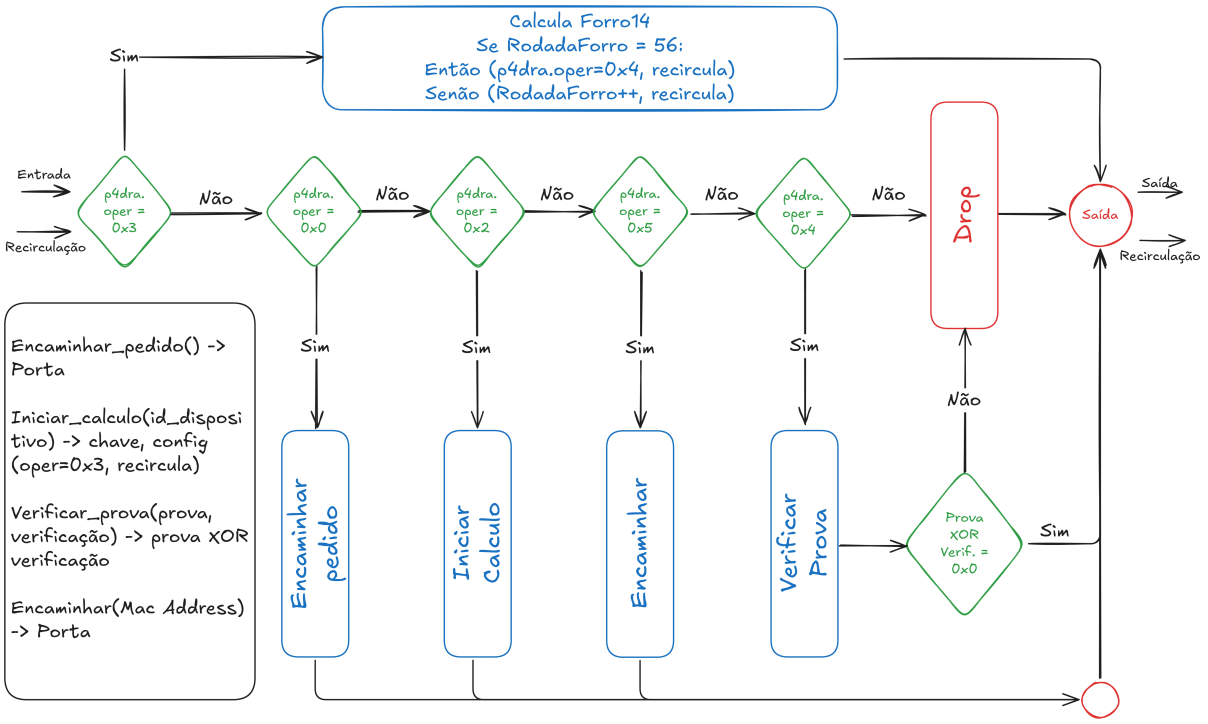

Figura 4.5 – Fluxograma da implementação em PDP.

Já no circuito de "Iniciar Cálculo", são coletadas na tabela do *switch* a chave e a configuração do dispositivo (c_i) por meio do d_i fornecido. Como o d_i não varia entre rodadas, o controlador precisa calcular e gravá-lo no *switch* apenas no Algoritmo de *Setup* (ilustrado na Figura 3.2), dispensando seu cálculo em toda rodada de atestação. Com essas informações, o *switch* inicia o cálculo da verificação da prova de atestação, montando a matriz de estado inicial do algoritmo Forro14 e alterando o campo Oper para 0x3, indicando que na próxima recirculação o pacote deve entrar no circuito de cálculo do Forro14.

No circuito de cálculo do Forro14, o pacote recebe um cabeçalho com a rodada de cálculo do keystream do Forro14 ("RodadaForro"), indicando o *Quarter Round* (QR) do algoritmo que deverá ser executado. Na implementação são executados 2 QRs do algoritmo por travessia do pacote (um no *pipeline "Ingress"* e outro no *pipeline "Egress"*), sendo incrementado o cabeçalho "RodadaForro" a cada cálculo feito. Por fim, o pacote é recirculado para continuar seu processamento no *switch*.

Como o algoritmo Forro14 possui 14 rodadas de 4 QRs cada, são necessárias 56 rodadas para seu cálculo completo, ou 28 travessias do pacote. Com isso, são necessárias 28 recirculações de pacotes para seu cálculo, sendo que o cabeçalho Oper é alterado para 0x4 após a travessia do *pipeline* com o cabeçalho RodadaForro com valor 56. Nesse momento, o cabeçalho com o campo RodadaForro é removido, mantendo apenas o cabeçalho da matriz de estado para ser finalizada (somada com a matriz dos parâmetros iniciais).

No circuito "Verificar prova" é realizada a finalização da matriz de estado e a comparação entre a prova calculada pelo *switch* e a prova recebida do provador. A finalização é realizada por meio da soma da matriz de estado inicial com a matriz de estado calculada após a aplicação dos QRs. Para realizar esse processo, é necessário que o *switch* busque novamente as informações do provador a partir do d_i. Essa busca repetida ocorre devido às operações diferentes realizadas entre o circuito de inicialização e o circuito de finalização da matriz. Teoricamente, seria possível otimizar esse passo com um passo comum de carregamento dos parâmetros em metadados antes da inicialização ou finalização, mas isso impactaria toda a compilação do código, podendo inviabilizar a construção de um *pipeline* carregável na ASIC. Essa otimização deverá ser avaliada em um trabalho futuro.

Após a finalização da matriz, a verificação é realizada etapa a etapa por meio de operações de XOR entre intervalos comuns da prova fornecida com a verificação calculada. Se em todas essas etapas a operação de XOR resultar em zero, significa que as provas são iguais e que o pacote pode ser encaminhado ao controlador. Caso contrário, o pacote é descartado.

O circuito "Encaminhar" meramente encaminha o pacote para uma porta de saída de acordo com um registro na tabela de encaminhamento do *switch*. Essa operação é necessária para encaminhar o pacote de configuração que o controlador envia ao mecanismo de atestação. Como os dados nesta operação não são de interesse do *switch*, ele não extrai nem modifica informações.

Outras operações podem coexistir no *switch*, como operações de encaminhamento L2/L3, ACLs e outros circuitos. Para isso, basta que o circuito do P4DRA seja isolado dos demais circuitos por meio do uso de uma estrutura de decisão (*if*) que verifique a validação do cabeçalho "p4dra_oper" antes da verificação do valor do campo Oper. Até mesmo o circuito de cálculo do Forro14 pode ser utilizado fora do pretexto de cálculo da prova de atestação.

4.2 Implementação do provador com memória segura

Considerando o cenário de data centers, os servidores possuem um BMC geralmente baseado na arquitetura ARM. Esse BMC possui acesso a interfaces e recursos para inventariar e gerenciar os dispositivos conectados à placa principal do servidor.

Conforme o modelo adversarial descrito na Seção 3.5, o BMC precisa de uma solução de memória segura para proteger a chave k do atacante com capacidades ativas no sistema operacional.

Considerando o cenário de data centers, bem como a possibilidade de expansão desta técnica de atestação remota para os cenários de IoT e redes celulares, foi escolhida a tecnologia de memória segura TrustZone disponibilizada pela ARM. Essa tecnologia cria um Ambiente de Execução Confiável (*Trusted Execution Environment*, TEE) e permite que operações sensíveis à segurança, como criptografia, e seus dados, como chaves e outros parâmetros secretos, possam ser realizadas mesmo em um sistema operacional não-confiável, por meio de uma segmentação de recursos baseada em hardware.

Neste TrustZone, a computação é dividida em 2 "mundos", chamados *Normal World* (NW) e *Secure World* (SW). No NW é executado o *Rich Execution Environment* (REE), geralmente um sistema operacional Unix completo (com drivers e aplicações). Já no SW é executado o TEE, um ambiente com menos recursos disponíveis, cujos objetivos são executar as Aplicações Confiáveis (*Trusted Applications*, TA) e armazenar os segredos em um *Storage* Seguro, um espaço da memória secundária do dispositivo que é cifrado pelo firmware, de forma que apenas o SW possa ler seu conteúdo. As TAs não podem ser modificadas em tempo de execução e devem ser assinadas pela chave do fabricante da plataforma para serem carregadas na inicialização segura.

A comunicação entre o NW e SW é feita por meio de APIs específicas de Chamadas ao Monitor Seguro (*Secure Monitor Calls*, SMC), um controlador altamente privilegiado no *firmware* do sistema. Cada fabricante de plataformas pode especificar seu próprio TEE e sua própria API de comunicação.

Para viabilizar o desenvolvimento de soluções utilizando o TrustZone, a fabricante de sistemas embarcados Linaro mantém uma solução aberta chamada *Open and Portable Trusted Execution Environment* (OPTEE) (LINARO, 2024). Essa solução é de código-aberto e pode ser carregada em diversas plataformas de hardware diferentes baseadas em ARM ou em emulador QEmu (*Quick Emulator*).

Utilizando o OPTEE, foi desenvolvida uma TA capaz de calcular uma prova utilizando os parâmetros de "id_rodada" e "nonce" informados no Algoritmo de Solicitação (ilustrado na Figura 3.3). Para armazenamento do "id_dispositivo" e "chave", o Algoritmo de Setup é utilizado. No protótipo desenvolvido, essa comunicação do Algoritmo de Setup ocorre de forma aberta, mas ela poderia ser cifrada com uma chave compartilhada.

Essa técnica poderia, a princípio, ser implementada utilizando um TPM. No entanto, um chip TPM possui funções criptográficas específicas que, em contexto geral, não são tão eficientes para plano de dados programável. Seria necessário fazer a extensão do TPM de forma a permitir que este execute o cálculo da prova com os parâmetros seguros.

Uma limitação encontrada para essa implementação do OPTEE como provador é o fato do SW não ter acesso ao sistema de arquivos do NW. Isso implica que o SW não pode obter as informações relevantes de configuração para fazer a atestação. Uma possível solução é utilizar o módulo TPM embarcado no servidor como um ponto comum de configuração, de forma que o SW possa acessar as configurações do dispositivo sem a interferência de um atacante no NW. Os impactos dessa implementação podem ser verificados em um trabalho futuro.

4.3 Atendimento dos requisitos técnicos

Por fim, verificamos a aderência da abordagem adotada aos requisitos levantados na literatura, conforme descrito na Seção 3.2.

Informação atualizada: *a atestação deve refletir a informação referente à última verificação.* Esse requisito é satisfeito utilizando o "*nonce*" (n) e o "id da rodada" (r) para diferenciar atestações e identificar a última atestação solicitada.

Informação compreensiva: *a informação contida na atestação deve permitir o verificador atestar o estado do dispositivo.* Esse requisito é satisfeito pela definição de uma política (o) para definir a configuração (c_i) a ser utilizada no cálculo da prova ($p_{i,r}$).

Mecanismo confiável: *o verificador deve ser capaz de confiar na informação de um provador mesmo que haja um adversário ativo na rede.* Esse requisito é satisfeito pela chave compartilhada (k) entre controlador, *switch* e o mecanismo de atestação do dispositivo, que garante a autenticidade da atestação.

Acesso exclusivo: *Apenas o mecanismo de atestação do provador deve possuir acesso à chave k.* Esse requisito é satisfeito utilizando o TEE para proteção dos segredos no Storage Seguro.

Sem vazamentos: *A atestação não pode vazar nenhuma informação sobre k para o atacante.* Esse requisito é satisfeito pela segurança provida pelo algoritmo Forro14 no cálculo do *keystream*, visto que mesmo que um atacante possua todos os outros parâmetros, não é possível descobrir k_i, a não ser por um ataque que esgote todas as possibilidades da chave k_i.

Imutabilidade: *O mecanismo de atestação não pode ser modificado por um atacante ativo ou passivo.* Esse requisito é satisfeito pela gravação do mecanismo em uma memória restrita. Quaisquer modificações nesta memória só podem ser realizadas por meio de um canal de comunicação seguro entre controlador e mecanismo ou por meio do carregamento de uma nova aplicação confiável assinada pelo fabricante no momento da inicialização do provador.

Execução atômica: *A execução do mecanismo de atestação não pode ser interrompido por nenhuma ação no dispositivo até sua conclusão.* O *Secure Monitor Call* não expõe APIs que permitam a interrupção da execução da aplicação confiável no TEE. Com isso, o sistema operacional não possui controle sobre a execução da aplicação confiável .

Chamada controlada: *O mecanismo de atestação só pode ser chamado pelo entrypoint esperado.* Esse requisito não foi implementado no protótipo desenvolvido. Ele poderia ser implementado de forma que, ao encaminhar a solicitação no Algoritmo de Solicitação, o *switch* calculasse uma prova de autenticação por meio de uma segunda chave compartilhada. Essa prova de autenticação seria calculada utilizando a geração de *keystream* do Forro14 (sem as modificações propostas no P4DRA), de forma que o mecanismo de atestação pudesse calcular a mesma prova de autenticação com a chave compartilhada.

Eficiência: *O protocolo de atestação deve ser mais eficiente do que se atestar um dispositivo de cada vez.* Esse requisito é satisfeito pela delegação das verificações do controlador para cada *switch*, viabilizando atestações em paralelo e mais rápidas que de forma sequencial.

Agregação de resultados: *Deve haver um mecanismo para agregar os resultados de cada verificação individual.* Esse requisito é satisfeito pela verificação granular: o *switch* responde ao controlador apenas as atestações válidas e pelo canal dedicado, reduzindo o uso de rede de operações para a transmissão das provas de atestação.

Topologia de Atestação: *Definir a estrutura usada para atravessar a topologia de dispositivos (Spanning-Tree, Mesh ou Pub/Sub).* Esse requisito é mais significativo em redes IoT, já que sua organização influencia na forma de envio das consultas e de agregação das atestações em uma CRA. Data centers possuem uma topologia *Spanning-Tree*, considerando que *switches* e provadores formam uma árvore de dispositivos, com canais exclusivos entre *switches* e controlador.

Tolerância a falhas: *Definir se o protocolo é resistente a modificações de topologia ou se dispositivos podem sair e entrar da topologia.* Esse requisito é satisfeito pelo gerenciamento do mapa de rede pelo controlador presente no modelo, que informa ao *switch* quaisquer modificações de configuração válida ou de posição de dispositivos, sendo o mapa da rede gerado pelos endereços MAC conhecidos de cada porta. A adição ou remoção de um dispositivo na rede implica na mudança da topologia conhecida pelo controlador, o que pode disparar uma ação de adaptação dos verificadores de cada dispositivo.

Atestação de controle de fluxo: *Indica se o protocolo suporta controle de fluxo, que significa que um protocolo pode mudar seu comportamento em tempo de execução, mesmo sem alterar os binários presentes. Portanto, é preciso atestar também o fluxo de instruções presentes na memória para garantir integridade.* Esse requisito é satisfeito pelo atacante

não ser capaz de interferir na operação do mecanismo em memória segura, pois ele não poderá alterar as instruções ou dados utilizados para o cálculo da prova $p_{i,r}$.

4.4 Considerações do capítulo

Neste capítulo foram apresentadas as implementações realizadas para a prova de conceito da técnica de atestação remota proposta. Foi descrita a implementação do verificador realizada em plano de dados programáveis com o detalhamento dos cabeçalhos utilizados para as comunicações entre os agentes no processo de atestação.

Essa implementação é particularmente desafiadora devido às limitações da arquitetura TNA, que não foi concebida com o objetivo de realizar operações criptográficas diretamente no processamento de pacotes. Pode-se citar a dificuldade de alocação dos dados através da travessia do *pipeline* de processamento como o principal desafio.

Já a implementação do provador foi realizada em uma plataforma emulada com a tecnologia ARM TrustZone, detalhando a comunicação entre o provador e a sua memória segura, responsável por proteger a chave secreta (k). Por fim, foi analisado o cumprimento pelas implementações realizadas dos requisitos técnicos levantados na literatura. A seguir, serão apresentados os experimentos conduzidos para a avaliação de desempenho e comparativo com a técnica tradicional de um verificador baseado em CPUs.

5 Avaliação experimental do protocolo de atestação remota

Para avaliar o desempenho da técnica proposta quanto à redução da janela de ataque para cenários de ataque *TOCTTOU*, foram realizados experimentos com o objetivo de avaliar o tempo de processamento dos componentes do sistema de forma individualizada, o tempo em função da quantidade de dispositivos a serem atestados em cada rodada e o impacto nos recursos de processamento e armazenamento de dados.

5.1 Planejamento e descrição dos experimentos

Para realizar o levantamento de dados, foram elaborados cinco experimentos: três com o objetivo de verificar a escalabilidade da técnica e dois para comparar os cenários de verificador central baseado em CPU x86 e verificador distribuído baseado em PDP. Os dados brutos dos experimentos estão disponíveis no repositório público do GitHub[1].

Esses experimentos foram executados em um testbed com um servidor x86 e um *switch* programável baseado na ASIC Tofino. O servidor é modelo Lenovo SR630 com um processador Intel Xeon Silver 4208 8c/16t 2.10GHz, 128GB de memória DDR4 2400MHz e uma interface de rede Mellanox MT27710 (ConnectX-4 Lx) de 25Gbps. Já o *switch* é um Edgecore DCS800 com capacidade de 32 portas 100Gbps QSFP28 em dois *pipelines* na ASIC Tofino, tendo embarcado um *Network Operating System* (NOS) Ubuntu Server 18.04 executando em um Intel® Pentium® D-1517 com 8 GB de RAM DDR4. Esse NOS será o responsável por fazer a inserção dos registros nas tabelas do *pipeline* definido. A verificação também poderia ser processada em SmartNICs conectadas a um servidor. A comparação entre um verificador com SmartNICs e um verificador em switch programável Tofino pode ser realizada como um trabalho futuro.

Para os experimentos, foi gerado um dataset de 1 milhão de conjuntos de "id_dispositivo", "chave" e "prova". O id_dispositivo (de 16 bytes) foi gerado por meio de um contador variando de 1 a 1 milhão com zeros à esquerda (7 caracteres ASCII) concatenado a uma representação dos nanosegundos (9 caracteres ASCII, variando de "000000000" a "999999999") do momento em que o número foi gerado. Esse conjunto de 16 caracteres ASCII produz uma representação de 16B (bytes) que é convertida em uma *hexstring* de 32 caracteres.

[1] https://github.com/regras/p4dra

Para a geração da chave, foram gerados e concatenados valores aleatórios ao id_dispositivo gerado, sendo calculado o hash SHA-256 desse valor que produz uma *hexstring* de 64 caracteres (32B).

Por fim, foi aplicado o algoritmo proposto para o cálculo da prova, realizando as modificações na implementação de referência do Forro14 para alterar os parâmetros de composição da matriz inicial conforme descritos na Seção 3.4.2. Essa prova é representada em uma *hexstring* de 128 caracteres. Portanto, cada conjunto de *hexstrings* armazenado em memória ocupa o equivalente a 224B (224 caracteres) de memória. Para a geração desse dataset com as provas dos dispositivos em uma rodada, foi escolhido o conjunto de caracteres "12345678" para os valores do id_rodada e do nonce, representando 8B cada em duas *hexstrings* de 16 caracteres cada.

Com o objetivo avaliar o desempenho do *switch* programável no cálculo e verificação de provas, o servidor de teste é conectado por meio de uma interface 10Gbps ao *switch* Tofino executando a implementação da técnica de atestação em PDP. Nesse cenário, a interface Mellanox capaz de 25Gbps tem sua taxa reduzida para 10Gbps de forma a descartar oscilações no desempenho de envio e recebimento de dados devido a possíveis efeitos físicos das conexões. Ainda assim, um pacote de pedido de verificação possui 888b (bits), sendo 112b do cabeçalho ethernet, 8b da operação, 128b do id_rodada e nonce, 128b do id_dispositivo e 512b da prova. Portanto, mesmo o envio de 1 milhão desses pacotes requer apenas 888Mbps de taxa de transmissão em cada direção.

Para os experimentos, a "configuração esperada" e a "configuração atual" são strings vazias (preenchidas com valores nulos) devido à limitação de acesso à configuração do dispositivo pelo *Secure World*, conforme descrito na seção 4.2. Nesse caso, as provas de atestação sempre serão válidas. Ainda assim, essa modificação não impacta nos valores coletados, pois as mesmas operações são executadas independente do *hash* da configuração esperada (c_i).

5.1.1 Atestação remota baseada em x86

O primeiro experimento foi executado de forma emulada dentro do servidor de testes, sem o uso de interconexões físicas. A Figura 5.1 sumariza o ambiente de teste descrito a seguir.

Neste cenário, um verificador baseado em linguagem C carrega o dataset de id_dispositivos e chaves para poder fazer a verificação das provas fornecidas pelos provadores. Para representar a conexão de rede, um tipo de interface de rede virtual *in-kernel* chamada *veth* é utilizada. Nesta interconexão, uma interface virtual *veth0* é conectada a uma interface virtual *veth1* por meio de drivers do próprio kernel. Essa interface é co-

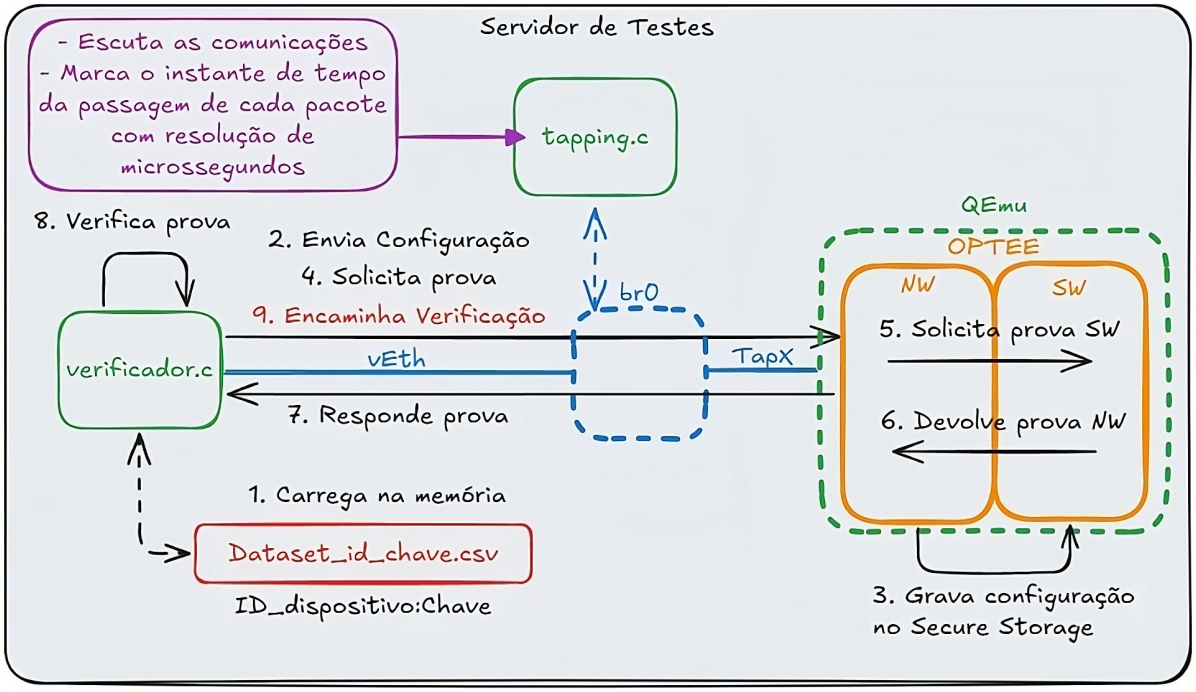

Figura 5.1 – Experimento 1: testes com verificador baseado em x86 e provador baseado em Arm64 com OP-TEE.

nectada a uma *bridge* chamada *br0*, responsável por conectar o verificador ao provador, representando o ambiente de rede. Para emular o provador, uma instância do OPTEE executando em QEMU com 2 vCPUs e 545MB de RAM é conectada à *bridge* br0 com uma interface do tipo *Tap*. Nesse cenário, apenas o *Normal World* (NW) do TrustZone emulado possui acesso à rede, devendo as interações com o *Secure World* (SW) serem realizadas por meio do *Normal World* com o uso do *Secure Monitor Call* (SMC). Além disso, um programa executando no servidor de testes (tapping.c) faz a escuta do tráfego na br0 utilizando a biblioteca *libpcap* e extrai um resumo do tráfego do protocolo P4DRA registrado com um *timestamp* com microssegundos de precisão. Esse resumo extraído é a fonte de dados para os resultados dos experimentos.

Conforme os passos na Figura 5.1, (passo 1) após carregar os dados em memória, o verificador (2) envia o pacote de operação 0x5 para o OPTEE indicando o id do dispositivo (d_i) e a chave (k_i) a serem gravados no *Storage* Seguro (durante o Algoritmo de Setup). Ao receber este pacote, (3) um *handler* executando como serviço no NW faz uma chamada ao programa cliente para solicitar à *Trusted Application* (TA) no SW a inserção destes dados no *Storage* Seguro. Os passos 1 a 3 não são considerados nas medições de nenhum dos experimentos, visto que são passos que só precisam ser executados pelo controlador quando as informações de um dispositivo são alteradas ou quando um dispositivo novo é conectado à rede.

Com a configuração inserida, (4) o verificador envia ao OPTEE um pacote

com a operação 0x1, solicitando uma prova de atestação e informando o id da rodada (r) e o *nonce* (n) para cálculo da atestação. O *handler* no NW recebe esse pacote, (5) solicita uma prova ao SW por meio do cliente TEE informando os parâmetros recebidos.

A TA então (6) calcula essa prova utilizando os parâmetros recebidos e os parâmetros gravados durante a configuração e devolve a prova calculada ($p_{i,r}$) ao cliente TEE. O *handler* então (7) obtém essa prova e elabora um pacote de resposta (oper 0x2) para o verificador, enviando-o pela rede. Por fim, o verificador recebe esta prova e faz o (8) cálculo da verificação com a configuração esperada utilizando os parâmetros compartilhados. Normalmente o processo terminaria neste ponto em uma atestação válida, mas para realizar a medição do desempenho, um pacote de verificação (Oper 0x4) é (9) enviado ao OP-TEE apenas para a medição de tempo ser realizada com o tapping.c.

Durante o experimento, esse processo da solicitação de prova até o envio da verificação foi realizado 100 vezes, com intervalos de 1 segundo entre cada solicitação enviada. A Tabela 5.1 mostra a média, desvio padrão, mínimo e máximo do tempo em microssegundos entre:

- *Tempo desde o pedido até a resposta* (Δ_{r-p}): o tempo decorrido entre a solicitação de atestação enviada pelo controlador (passo 4) e a resposta do provador (passo 7) passarem pelo tapping.c;
- *Tempo desde a resposta até a verificação* (Δ_{v-r}): o tempo decorrido entre a resposta do provador (passo 7) e a verificação pelo controlador (passo 9) passarem pelo tapping.c; e
- *Tempo desde o pedido até a verificação* (Δ_{v-p}): o tempo decorrido entre a solicitação (passo 4) e a verificação (passo 9) passarem pelo tapping.c.

	Experimento 1 - Controlador x86 e OP-TEE		
	$\Delta_{r\text{-}p}$ (µs)	**$\Delta_{v\text{-}r}$ (µs)**	**$\Delta_{v\text{-}p}$ (µs)**
Média:	266 996	2 946	269 862
Desvio padrão:	6 704	178	6 711
Erro padrão:	670	18	671
Mínimo:	254 619	2 127	257 430
Máximo:	285 373	3 149	288 447

Tabela 5.1 – Experimento 1: tempos de solicitação, resposta e verificação.

É importante destacar nessa tabela o tempo médio que um verificador baseado em x86 leva para realizar a verificação de uma prova: em média 2,946 microssegundos. O tempo do pedido à resposta é o tempo médio levado por um dispositivo OP-TEE para realizar o cálculo da prova de atestação e respondê-la para o verificador. Esse processo de atestação contempla os processos de Solicitação e Verificação, conforme a Figura 5.2, destacando que o passo de "Encaminha verificação ao Provador" não é realizado na execução normal do processo, sendo utilizada apenas para a coleta de medições.

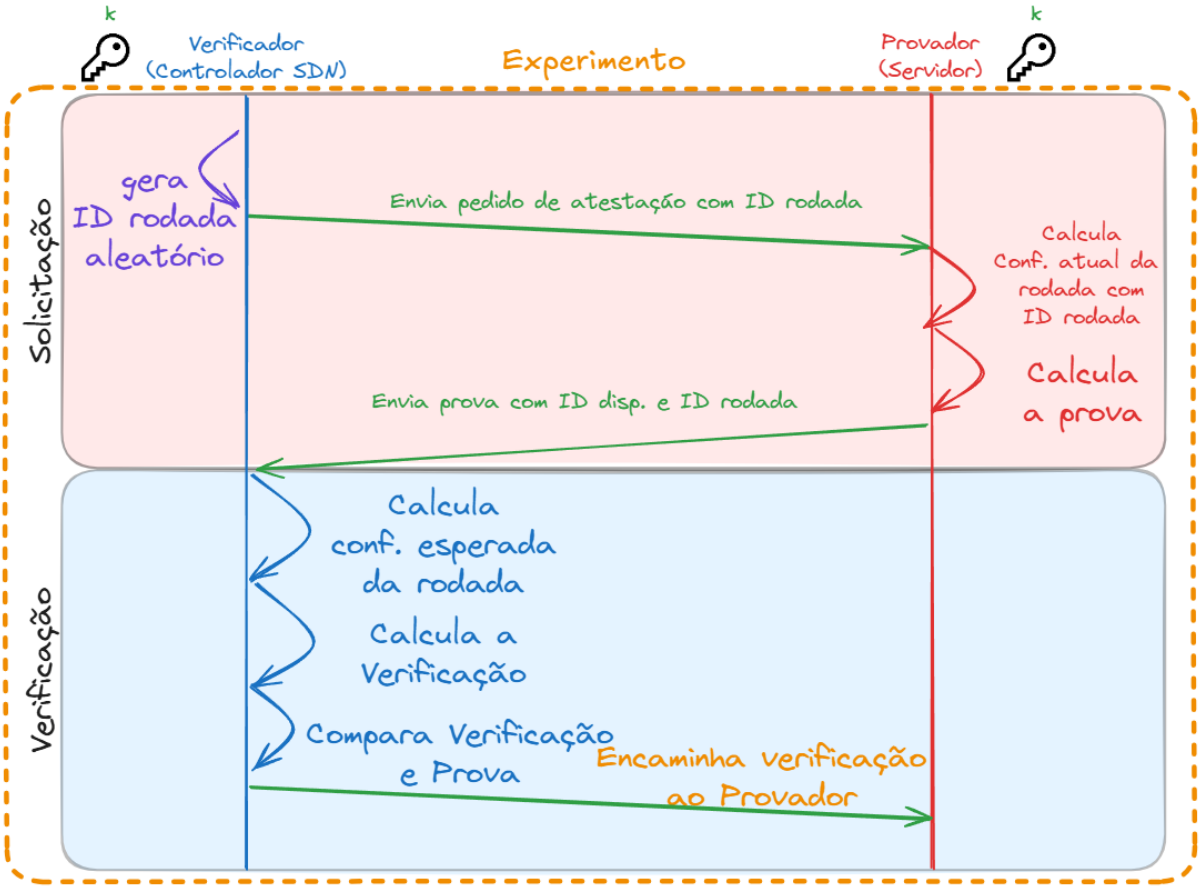

Figura 5.2 – Experimento 1: processos do protocolo contemplados no experimento.

A seguir, será detalhado outro experimento realizado para a medição com a presença do *switch* programável baseado na ASIC *Tofino* realizando a verificação da prova em nome do controlador.

5.1.2 Atestação remota baseada em Tofino

Neste segundo experimento foi adicionada a presença do *switch* programável como um intermediário entre o controlador e o *OP-TEE*, de forma que a comunicação entre eles atravesse o *switch*. Para isso, a *bridge* br0 é configurada com regras *ebtables* de forma que as comunicações entre a veth do controlador e a interface eth que segue ao

switch, e entre a interface tap do provador e a interface eth do *switch* são liberadas, mas a comunicação direta, não.

Da mesma forma, a medição é realizada pela aplicação tapping.c escutando as comunicações da *bridge* alocada no servidor. Os passos a serem realizados são os mesmos que no primeiro experimento, mas nesse caso o *switch* também (1) recebe as informações de dispositivos por meio de um controlador embarcado (esse controlador embarcado é utilizado apenas para configurar o *switch*, mas o mesmo processo poderia ser realizado pelo controlador no servidor) e (8) a verificação é realizada pelo programa P4 em execução no plano de dados. Por fim, a verificação é (9) encaminhada ao controlador. A Figura 5.3 ilustra os passos do experimento.

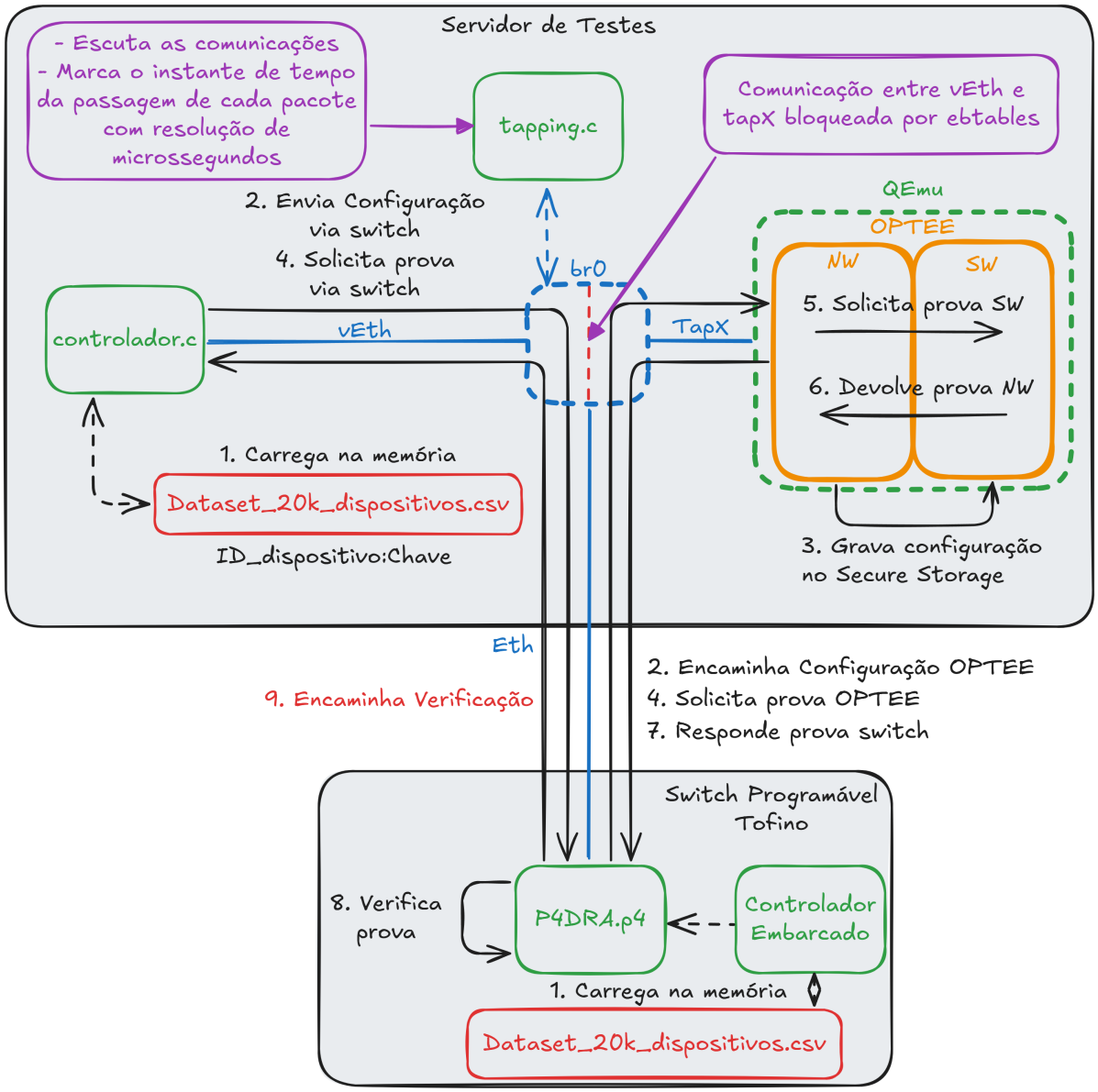

Figura 5.3 – Experimento 2: testes com verificador baseado em PDP e provador baseado em Arm64 com OP-TEE.

Da mesma forma, foram realizadas 100 solicitações de atestação do controlador ao provador, mas com a verificação sendo realizada pelo *switch*. A Tabela 5.2 demonstra as mesmas estatísticas coletadas, mas com destaque à média do tempo entre a resposta e a verificação, indicando o tempo que o *switch* programável levou para fazer a verificação. Também é importante destacar o tempo entre pedido e resposta, indicando que a presença do *switch* ou a necessidade da travessia da interface física não teve influência no tempo entre solicitação e resposta, já que este foi, em média, menor do que o tempo com a conexão apenas com interfaces virtuais (veth, tap e br0) do *kernel*.

	Experimento 2 - Switch Tofino e OP-TEE		
	$\Delta_{r\text{-}p}$ (µs)	**$\Delta_{v\text{-}r}$ (µs)**	**$\Delta_{v\text{-}p}$ (µs)**
Média:	265 676	44	265 720
Desvio padrão:	5 888	9	5 887
Erro padrão:	589	1	589
Mínimo:	252 894	21	252 938
Máximo:	284 404	88	284 448

Tabela 5.2 – Experimento 2: tempos de solicitação, resposta e verificação.

Como pode ser verificado na tabela, o tempo médio para verificação pelo *switch* foi de 44 microssegundos, ou uma taxa aproximadamente 66 vezes mais rápida que a do verificador baseado em x86. A Figura 5.4 mostra as operações que estão sendo avaliadas por esse experimento. O destaque em comparação à Figura 5.2 é a presença do verificador intermediário nos processos. Da mesma forma, o passo de encaminhar a prova ao controlador acontece sempre que a atestação tem sucesso, ou seja, na maioria das vezes em casos normais, onde ataques não estão ocorrendo. Mas, para fins de medição, o envio é realizado independente do resultado da verificação.

Infelizmente, não foi encontrado na literatura o tempo de verificação de provas de atestação em um cenário real de data centers. No entanto, o processo tradicional de atestação remota em data centers realiza a verificação de assinaturas digitais do hash enviado pelo provador para garantir autenticidade, o que não é diretamente comparável ao processo proposto nessa pesquisa.

A seguir, será avaliada a escalabilidade da técnica para cada um dos componentes utilizados: o verificador baseado em x86, o verificador baseado em PDP e o provador desenvolvido no OP-TEE.

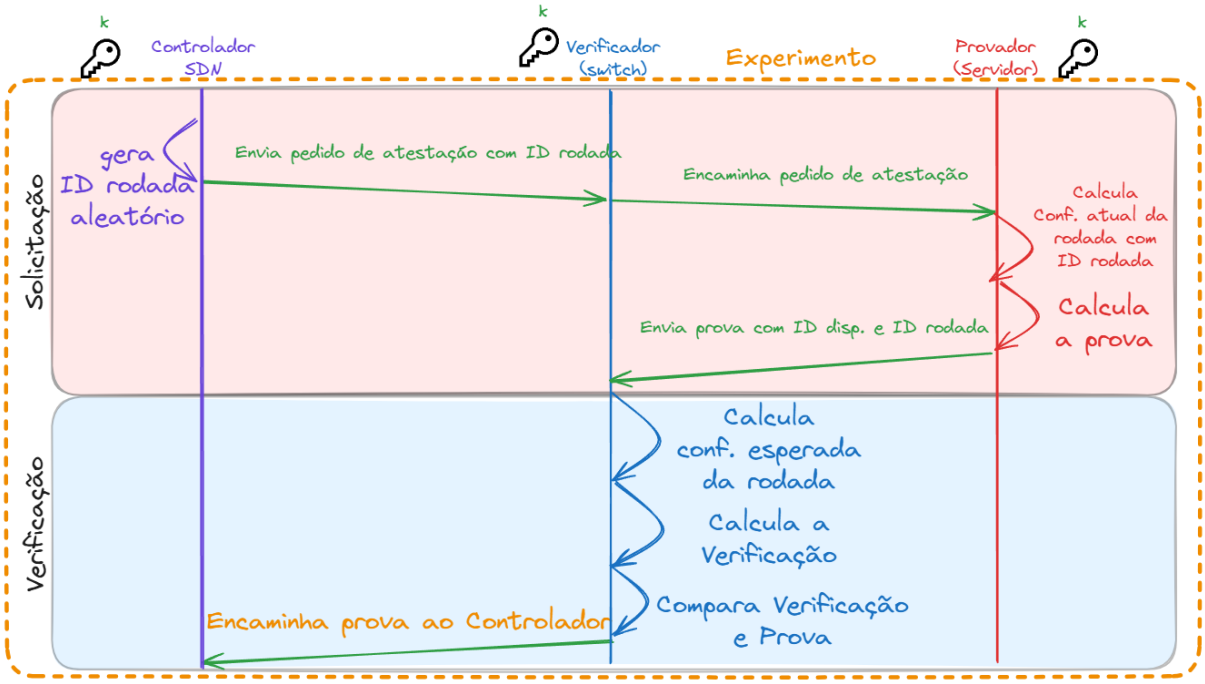

Figura 5.4 – Experimento 2: processos do protocolo contemplados no experimento.

5.2 Análise de escalabilidade

Um outro aspecto a ser verificado é a escalabilidade dessa técnica de atestação. Os experimentos apresentaram os valores com rodadas de atestação realizadas com apenas um único dispositivo, o que não representa a realidade em cenários de data center, em que a quantidade de dispositivos a serem atestados podem chegar à casa de centenas de milhares, ou até mesmo cenários de IoT, onde milhões de dispositivos podem ser atestados.

Para isso, serão demonstrados 3 experimentos, cada um com um foco diferente no teste de escalabilidade: o primeiro experimento avaliou a capacidade de resposta a solicitações de atestação do provador, o segundo experimento testou a capacidade de verificação do verificador baseado em x86 e o último experimento verificou a capacidade de verificação do verificador baseado em PDP.

A Figura 5.5 ilustra o cenário do primeiro experimento de escalabilidade: neste cenário, um programa emissor.c irá carregar na memória o dataset de dispositivos e enviará 250 pedidos de atestação ao provador, de forma que ele deva calcular todas as provas e respondê-las no menor tempo possível. A Figura 5.6 ilustra o processo que foi testado nesse experimento. Os resultados são sumarizados na Tabela 5.3, onde Δ_{r-p} indica o tempo entre os passos 4 e 7 e (Δ_{r-r}) (*tempo entre respostas enviadas*) o tempo entre duas respostas do provador serem capturadas pelo tapping.c.

Esse tempo da solicitação à resposta aumenta continuamente, pois o provador precisa enfileirar e carregar as provas uma a uma. É importante destacar que o tempo

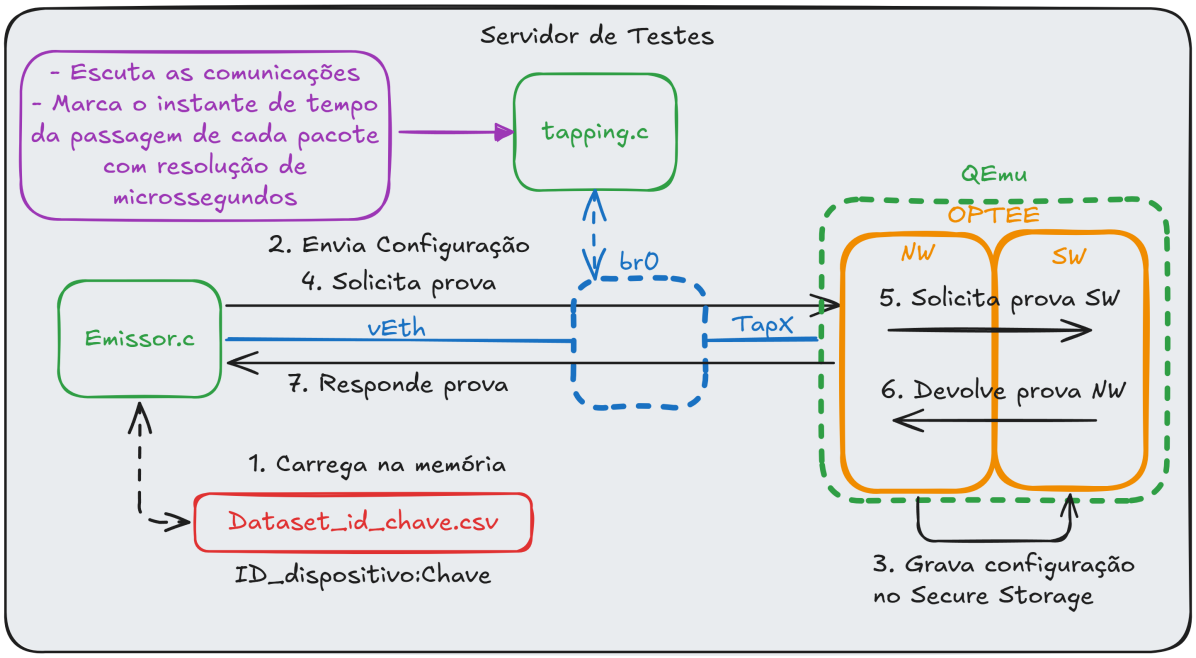

Figura 5.5 – Experimento 3: testes de escalabilidade do provador baseado em Arm64 com OP-TEE.

Experimento 3 - Escalabilidade provador		
Solicitação	**$\Delta_{r\text{-}p}$ (µs)**	**$\Delta_{r\text{-}r}$ (µs)**
1	269.420	0
2	567.565	298.145
3	859.965	292.400
4	1.135.839	275.874
...	...	...
247	68 481 120	265 557
248	68 778 891	297 771
249	69 054 895	276 004
250	69 317 922	263 027
Média:		275 318
Desvio padrão:		12 179
Erro padrão:		770
Mínimo:		255 889
Máximo:		309 188

Tabela 5.3 – Experimento 3: tempos de solicitação e resposta do provador.

médio entre respostas é condizente com o tempo médio verificado nos outros dois experimentos.

Esse cenário de várias solicitações a um dispositivo em curto período é irreal, visto que cada dispositivo só precisará responder a uma solicitação por rodada. No entanto, destaca a importância de considerar o tempo de resposta da prova pelos provadores, de forma que o período de atestação não seja menor do que este tempo para evitar o

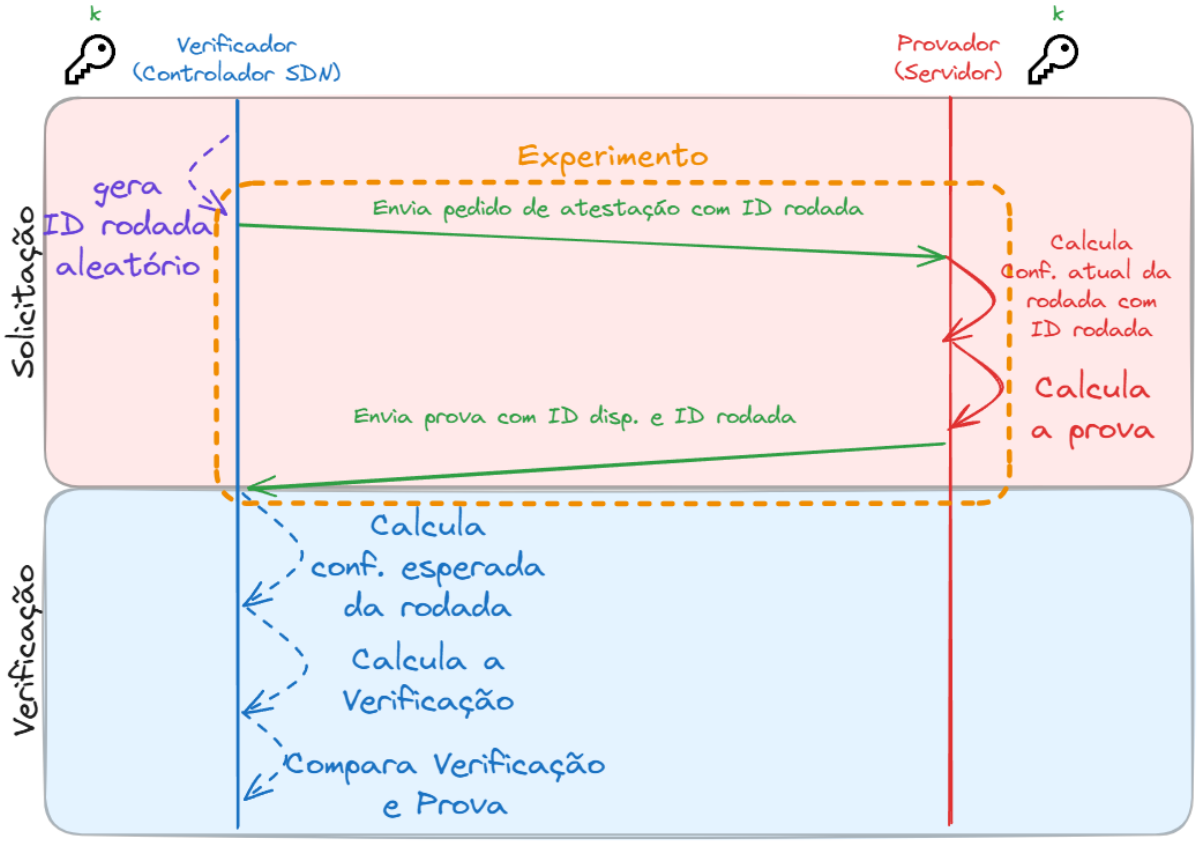

Figura 5.6 – Experimento 3: processos do protocolo contemplados no experimento.

enfileiramento das solicitações. Considerando que o tempo médio para o provador testado é de aproximadamente 275 milissegundos, parece ser razoável requerer que as atestações ocorram com uma frequência de, por exemplo, uma solicitação por segundo, pelo menos.

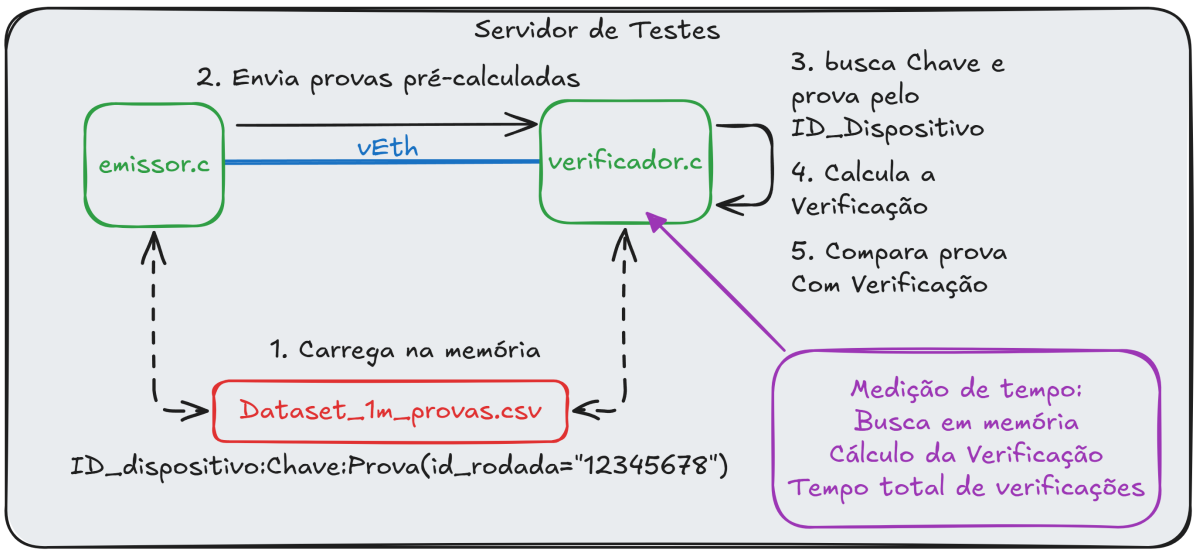

Figura 5.7 – Experimento 4: testes de escalabilidade do verificador baseado em x86.

A Figura 5.7 ilustra o experimento com foco no teste de escalabilidade do verificador baseado em x86. Nesse cenário, um programa emissor.c carrega os dados de 1

milhão de provas pré-calculadas, de forma a enviar uma quantidade variável de provas para verificação em sequência, simulando o comportamento desta quantidade de provadores respondendo ao pedido de atestação de uma vez. A Figura 5.8 ilustra os processos que são avaliados nesse experimento.

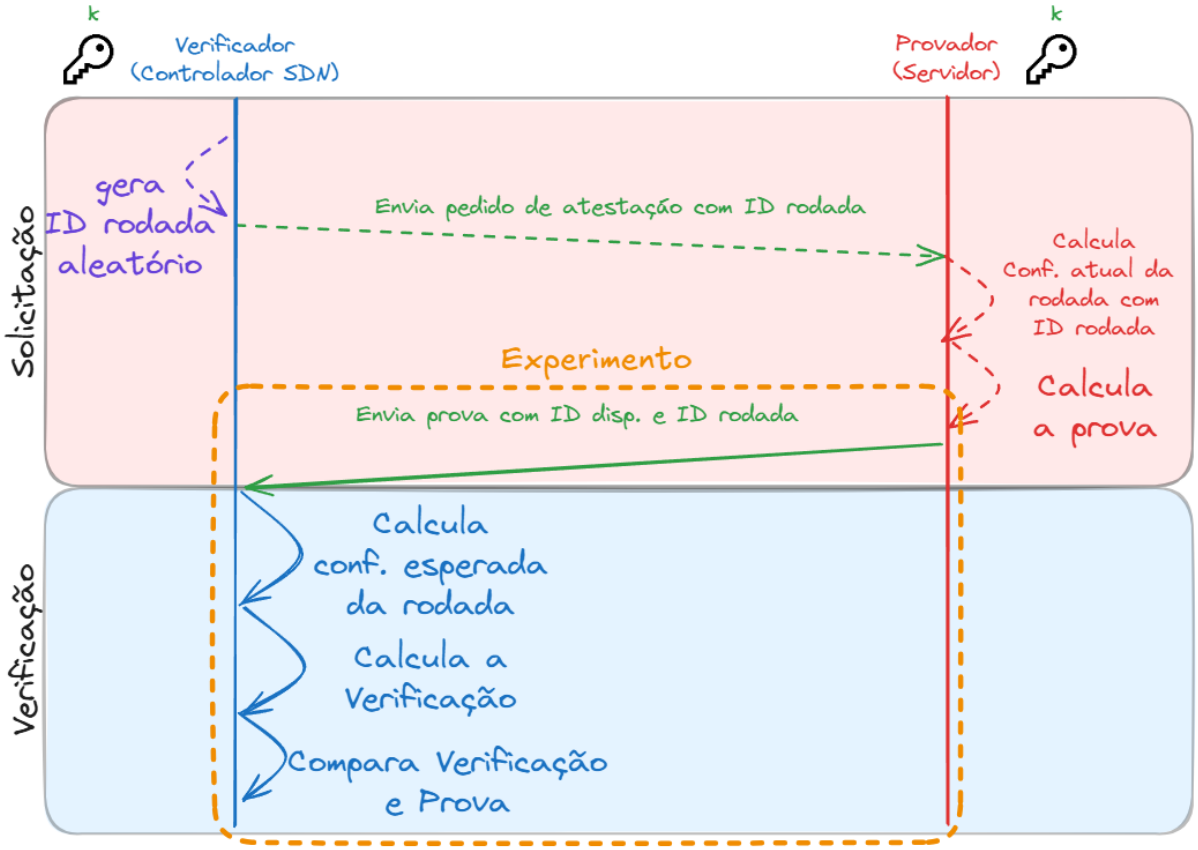

Figura 5.8 – Experimento 4: processos do protocolo contemplados no experimento.

Para verificação, o experimento é realizado 100 vezes e são extraídas as estatísticas de tempo para cada quantidade de dispositivos. A medição de tempo é realizada no próprio verificador e são extraídos três medidas para cada quantidade de dispositivos: (i) o tempo para processamento de toda a verificação, (ii) o tempo para busca da chave em memória pelo ID do dispositivo que enviou a prova, e (iii) o tempo utilizado apenas para o cálculo da prova. Essas estatísticas são mostradas na Tabela 5.4, onde os tempos são dados em microssegundos. Foram feitas medições de 2.500 a 20.000 dispositivos, com intervalos de 2.500 em 2.500. Esse intervalo foi pensado para comparações com o plano de dados, devido a limitações que serão explicadas mais adiante. Ainda assim, esse intervalo representa uma margem viável para diversos tamanhos de data centers.

Como pode ser verificado na Tabela 5.4, a busca em memória é o menor tempo decorrido medido do processo. Essa busca foi modificada para este experimento de forma a causar o menor impacto possível no processo de verificação. No processo realizado para o experimento, as provas são enviadas e os dados dos dispositivos são armazenados de

		Quantidade Dispositivos							
Operação	**Estatísticas**	**2 500**	**5 000**	**7 500**	**10 000**	**12 500**	**15 000**	**17 500**	**20 000**
Busca Memória	**Média:**	326	650	990	1 315	1 641	1 982	2 298	2 638
	Desvio padrão:	21	31	51	50	65	65	64	107
	Erro padrão:	2	3	5	5	6	7	6	11
	Mínimo:	286	580	903	1 202	1 542	1 854	2 191	2 496
	Máximo:	441	770	1 283	1 514	1 846	2 200	2 492	3 432
Cálculo Prova	**Média:**	1 655 622	3 273 498	4 892 504	6 519 213	8 118 563	9 725 693	11 345 803	12 971 479
	Desvio padrão:	8 606	7 412	11 491	17 928	22 350	23 661	17 557	18 161
	Erro padrão:	861	741	1 149	1 793	2 235	2 366	1 756	1 816
	Mínimo:	1 647 855	3 265 008	4 883 520	6 505 582	8 105 299	9 700 454	11 326 409	12 942 364
	Máximo:	1 688 121	3 309 621	4 929 423	6 605 835	8 240 267	9 875 753	11 385 135	13 015 043
Verificação Total	**Média:**	1 703 768	3 368 488	5 034 989	6 706 547	8 351 976	10 006 007	11 672 657	13 343 103
	Desvio padrão:	8 874	8 072	12 057	18 623	23 255	25 018	18 042	18 730
	Erro padrão:	887	807	1 206	1 862	2 325	2 502	1 804	1 873
	Mínimo:	1 695 054	3 358 729	5 024 230	6 692 255	8 339 375	9 979 039	11 652 539	13 311 946
	Máximo:	1 736 725	3 406 917	5 072 354	6 796 615	8 480 366	10 166 481	11 713 315	13 386 909
Tempo médio de atestação/dispositivo		682	674	671	671	668	667	667	667

Tabela 5.4 – Experimento 4: tempos de verificação de um verificador baseado em x86 em função da quantidade de dispositivos atestados.

forma sequencial, de tal forma que o índice da prova recebida é igual ao índice de vetor de dados dos dispositivos na memória do verificador, reduzindo a busca a apenas um acesso direto à memória. Esse processo é uma forma de realizar os experimentos com menor custo e sem precisar levar em consideração métodos de busca de dados em memória, *cache*, ambiente de execução e implementações de banco de dados. Para um cenário real, o ID do dispositivo poderia ser um índice para um banco de dados do tipo chave-valor (*key=value*), indexando a chave do dispositivo e a configuração esperada.

Ainda conforme a tabela, o verificador baseado em x86 é capaz de verificar 2500 dispositivos em até 1,736 segundos, o que inviabiliza realizar uma verificação por segundo para todos eles. Pode-se calcular que o verificador leva em média 667 microssegundos por dispositivos para realizar todo o processo de verificação de 20.000 dispositivos, o que permite que até 1499 dispositivos possam ser atestados dentro de 1 segundo. A Figura 5.9 apresenta o tempo médio de verificação por dispositivo em função da quantidade de dispositivos verificados pelo verificador baseado na arquitetura x86.

Conforme a quantidade de dispositivos a serem verificados aumenta, o verificador baseado em x86 se torna um gargalo, implicando no aumento da janela de ataque *TOCTTOU* nos dispositivos. Para atestar 20.000 dispositivos são necessários em média 13,34 segundos para a atestação de todos dispositivos. Foram feitos experimentos únicos (não repetidos) para 100.000 e 1.000.000 de dispositivos, sendo constatado que o tempo médio de atestação por dispositivo se manteve relativamente constante, visto que são necessários 71,14 segundos para 100.000 dispositivos (711μs por dispositivo) e 680,93 segundos para 1.000.000 de dispositivos (681μs por dispositivo).

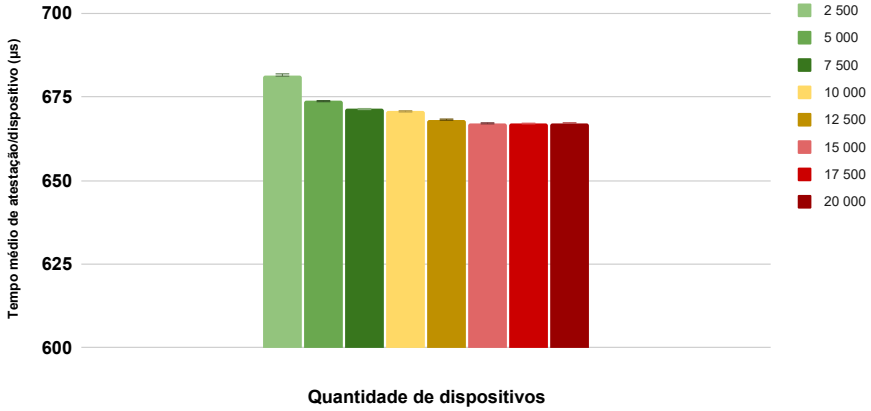

Figura 5.9 – Gráfico do tempo médio de verificação por dispositivo para o verificador baseado em x86.

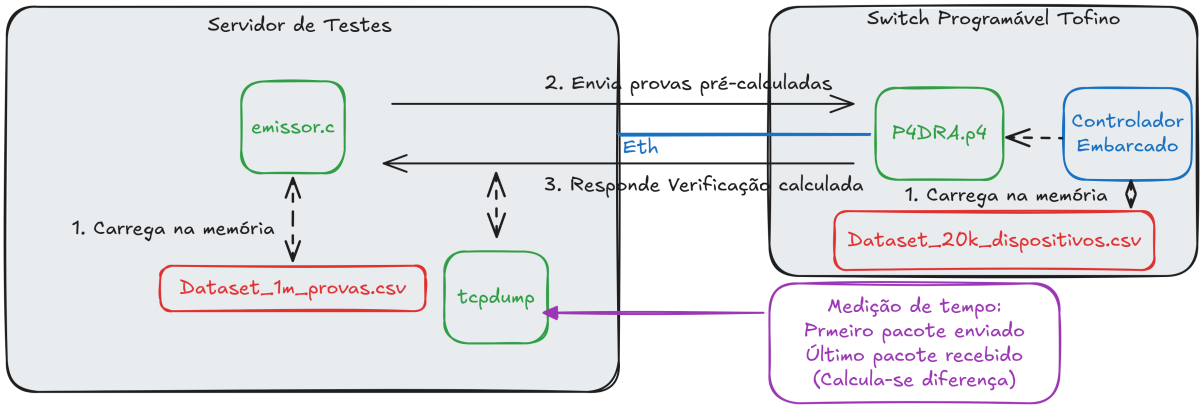

Figura 5.10 – Experimento 5: testes de escalabilidade do verificador baseado em PDP.

O último experimento avaliou a capacidade de um verificador baseado em plano de dados programável (no caso, na arquitetura Tofino). A Figura 5.10 ilustra o cenário de experimentação e a Figura 5.11 ilustra os processos testados nesse experimento. Pela ASIC programável ser uma implementação baseada em hardware, ela possui uma vantagem considerável em relação à implementação baseada em software para um processador de propósito geral como o verificador x86. No entanto, a elaboração de uma implementação em um hardware programável projetado para o processamento de pacotes traz diversas dificuldades.

Para esses experimentos, devido às restrições de construção do *pipeline* no hardware Tofino, o máximo de dispositivos que podem ser atestados por *switch* é de 20.000 dispositivos. O verificador baseado em PDP é limitado a 20.000 dispositivos, pois ao superar a limitação de endereço de memória SRAM acessível em cada estágio (10MB para a arquitetura TNA), o compilador de *pipeline* duplica a tabela para o estágio seguinte e continua a alocação de memória utilizando o próximo estágio. Como a implementação utiliza quase todos os estágios disponíveis, o compilador falha em gerar uma definição

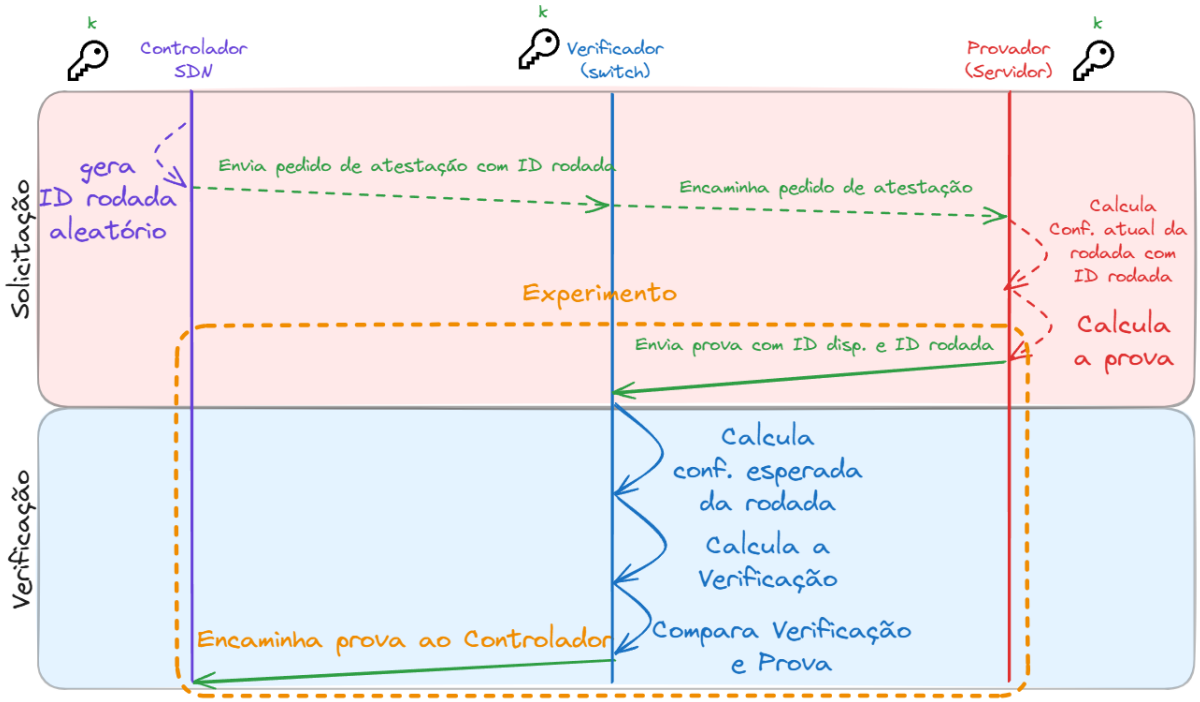

Figura 5.11 – Experimento 5: processos do protocolo contemplados no experimento.

de *pipeline* válida para armazenar mais do que 20.000 registros nas tabelas, visto que a duplicação de uma tabela de um estágio em outro causa impacto em diversos outros recursos na compilação de um *pipeline* viável.

Como a presença desses *switches* dividem a rede em topologia de árvore, a ideia é que um único *switch* não faça a atestação de todos os dispositivos da rede, mas que o trabalho seja dividido entre os *switches* que compõem a topologia. Uma discussão a respeito da topologia de rede e de possíveis abordagens da divisão de provadores entre os *switches* será realizada mais adiante.

A Tabela 5.5 apresenta os tempos medidos de 100 repetições do experimento, onde são coletados os instantes de envio da prova de atestação por um emissor e a verificação respondida pelo *switch*. Assim como no experimento 2, mesmo as verificações que falharem são encaminhadas de volta pela rede para a coleta de métricas. Como pode ser calculado pela tabela, a verificação baseada em PDP leva até aproximadamente 5 microssegundos por dispositivo (máximo) para um *switch* capaz de atestar 20.000 dispositivos. A Figura 5.12 apresenta o tempo médio de verificação por dispositivo em função da quantidade de dispositivos verificados pelo verificador baseado na arquitetura Tofino.

A tabela 5.6 e a Figura 5.13 apresentam um comparativo entre os verificadores baseados em x86 e PDP pelas métricas coletadas nos experimentos 4 e 5.

Como pode ser observado, a atestação baseada em PDP Tofino tem uma velocidade de atestação aproximadamente 154x mais rápida para 20.000 dispositivo do que

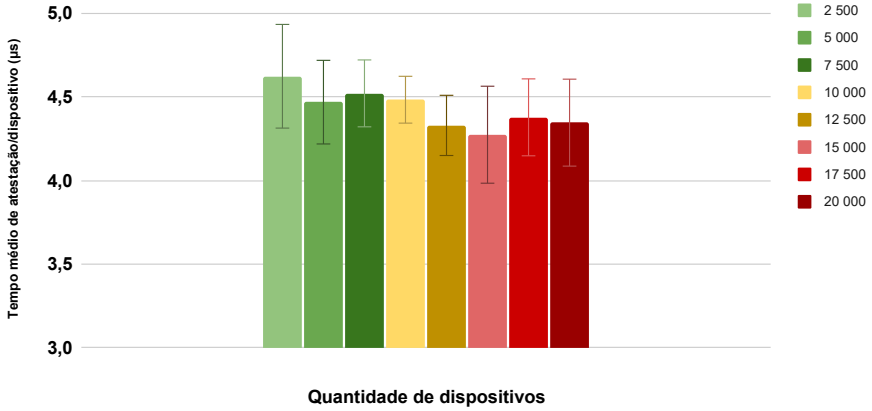

Figura 5.12 – Gráfico do tempo médio de verificação por dispositivo para o verificador baseado em PDP.

		Quantidade Dispositivos							
Operação	**Estatísticas**	**2 500**	**5 000**	**7 500**	**10 000**	**12 500**	**15 000**	**17 500**	**20 000**
Verificação Total	**Média:**	11 557	22 340	33 904	44 826	54 112	64 099	76 601	86 909
	Desvio padrão:	776	1 291	1 533	1 434	2 291	4 420	4 116	5 236
	Erro padrão:	78	129	153	143	229	442	412	524
	Mínimo:	10.183	20.106	29.881	41.811	49.835	58.757	68.588	77.835
	Máximo:	14.767	26.648	37.508	48.147	61.300	93.850	87.468	99.076
Tempo médio de atestação/dispositivo		5	4	5	4	4	4	4	4

Tabela 5.5 – Experimento 5: tempos de verificação de um verificador baseado em PDP em função da quantidade de dispositivos atestados.

	Tempo (microssegundos)		
Quantidade de dispositivos	**Tofino (switch programável)**	**x86 (processo completo)**	***Speedup***
2 500	11 557	1 703 768	147x
5 000	22 340	3 368 488	151x
7 500	33 904	5 034 989	149x
10 000	44 826	6 706 547	150x
12 500	54 112	8 351 976	154x
15 000	64 099	10 006 007	156x
17 500	76 601	11 672 657	152x
20 000	86 909	13 343 103	154x

Tabela 5.6 – Comparação entre os desempenhos dos verificadores baseados em x86 e PDP.

a implementação baseada em x86. Portanto, o verificador baseado em PDP pode realizar rodadas de atestações em média 151,6x mais frequentes do que o verificador baseado em x86.

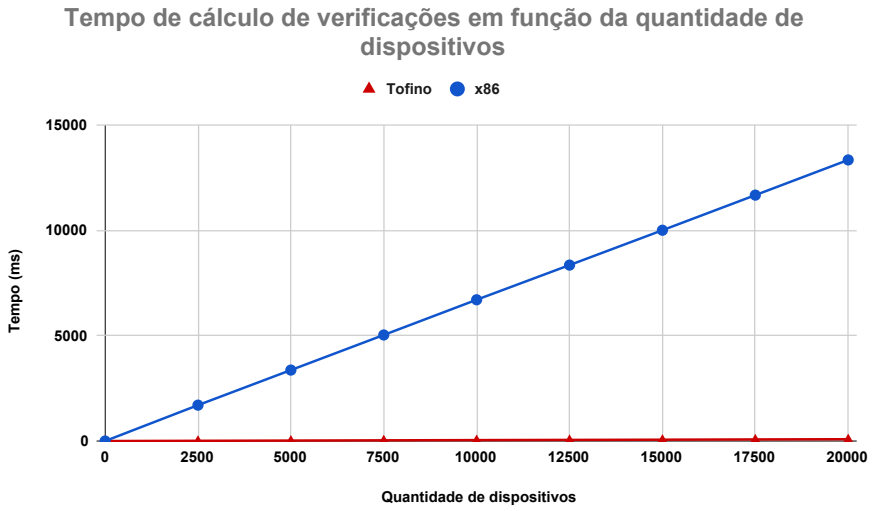

Figura 5.13 – Gráfico de comparação entre os desempenhos dos verificadores baseados em x86 e PDP.

5.3 Análise de uso de recursos

Com a implementação dos verificadores em arquiteturas diferentes, o uso de recursos em cada tipo de verificador não pode ser diretamente comparado. Neste caso, será feita uma análise específica para cada arquitetura. Considerando que os experimentos anteriores exploraram as questões de tempo, o que é diretamente refletido pelo processamento realizado nos dispositivos, o foco será a análise de armazenamento de dados em memória.

No caso da arquitetura x86, o verificador precisa armazenar em sua memória 3 informações para cada provador: o id_dispositivo, codificado em uma hexstring de 32 caracteres (32B), a chave codificada em uma hexstring de 64 caracteres (64B) e a configuração esperada codificada em uma hexstring de 64 caracteres (64B), totalizando 160B por provador. A informação do id_rodada (codificado em 32B), a configuração do dispositivo na rodada (codificado em 64B), a prova fornecida (codificado em 128B) e a verificação calculada (codificado em 128B) são armazenadas durante o cálculo da prova de verificação. A Tabela 5.7 sumariza a quantidade de memória necessária para a verificação por rodada em função da quantidade de dispositivos. O "Armazenamento em repouso" refere-se aos valores que não são alterados durante os processos de solicitação e verificação (ou seja, durante as rodadas de atestação). Já o "Armazenamento durante o cálculo" refere-se ao armazenamento em memória utilizado durante uma rodada de atestação. Por fim, o "Total" refere-se à soma de uso dos recursos em repouso e durante uma rodada de atestação. Como o ID_rodada não varia em função da quantidade de dispositivos, seu valor é fixo para qualquer quantidade de dispositivos.

Dispositivos	Armazenamento em repouso (Bytes)				Armazenamento durante o cálculo (Bytes)					total (B)
	Configurações	Chave	ID_dispositivo	Subtotal	Config. Rodada	ID_rodada	Prova	Verificação	Subtotal	
1	64	64	32	**160**	64	64	128	128	**384**	**544**
100	6 400	6 400	3 200	**16 000**	6 400	64	12 800	12 800	**32 064**	**48 064**
2 500	160 000	160 000	80 000	**400 000**	160 000	64	320 000	320 000	**800 064**	**1 200 064**
5 000	320 000	320 000	160 000	**800 000**	320 000	64	640 000	640 000	**1 600 064**	**2 400 064**
7 500	480 000	480 000	240 000	**1 200 000**	480 000	64	960 000	960 000	**2 400 064**	**3 600 064**
10 000	640 000	640 000	320 000	**1 600 000**	640 000	64	1 280 000	1 280 000	**3 200 064**	**4 800 064**
12 500	800 000	800 000	400 000	**2 000 000**	800 000	64	1 600 000	1 600 000	**4 000 064**	**6 000 064**
15 000	960 000	960 000	480 000	**2 400 000**	960 000	64	1 920 000	1 920 000	**4 800 064**	**7 200 064**
17 500	1 120 000	1 120 000	560 000	**2 800 000**	1 120 000	64	2 240 000	2 240 000	**5 600 064**	**8 400 064**
20 000	1 280 000	1 280 000	640 000	**3 200 000**	1 280 000	64	2 560 000	2 560 000	**6 400 064**	**9 600 064**
100 000	6 400 000	6 400 000	3 200 000	**16 000 000**	6 400 000	64	12 800 000	12 800 000	**32 000 064**	**48 000 064**
1 000 000	64 000 000	64 000 000	32 000 000	**160 000 000**	64 000 000	64	128 000 000	128 000 000	**320 000 064**	**480 000 064**

Tabela 5.7 – Armazenamento em memória necessário para cálculo de verificações no verificador baseado em x86.

Há também de se considerar a existência de uma tolerância de tempo para rodadas de atestação, implicando em rodadas de atestação sobrepostas. Por exemplo, a frequência de atestação pode ser uma vez por segundo e o dispositivo ter uma tolerância de 4 segundos para responder à atestação, algo compreensível em cenários de alta latência de rede ou com alta variação de tempo de resposta de provadores (como em IoT, por exemplo). Nesse caso, a necessidade de armazenamento durante o cálculo irá variar pela razão entre tolerância e frequência de atestação, ou seja, $r_s = \lceil \tau / T \rceil$, onde r_s é a quantidade de rodadas sobrepostas, τ é a tolerância em segundos, T é o período entre solicitações de atestação em segundos e $\lceil\ \rceil$ é o arredondamento para o inteiro igual ou imediatamente superior.

Considerando a possibilidade de rodadas de atestação sobrepostas, a quantidade de memória em bytes necessária para armazenar todos os dados utilizados no cálculo é dada pela função $f(d, r_s) = 160d + 320dr_s + 64r_s$, onde d é a quantidade de dispositivos a serem atestados e r_s é a quantidade de rodadas sobrepostas.

Ao avaliar o uso de recursos de armazenamento utilizados no verificador baseado em PDP, vemos que 15 componentes diferentes da arquitetura TNA são utilizados. A quantidade de recursos utilizados é definida no momento da compilação do *pipeline*, visto que a quantidade de registros nas tabelas do *switch* impacta na forma de construção do *pipeline*.

A Figura 5.14 apresenta o uso de recursos do *switch* para os casos de atestação de 200 dispositivos e de 20.000 dispositivos. Como destaque do gráfico, podemos citar o *Action Data Bus Bytes* (ADBB), um barramento utilizado para os resultados de operações realizadas no *switch* de acordo com o *match* em tabelas, a *Static RAM* (SRAM) utilizada para armazenar as chaves de busca em tabelas e os parâmetros de *actions*, e os *Packet Header Vectors* (PHV), que representam registradores utilizados em cada estágio de processamento do *pipeline*.

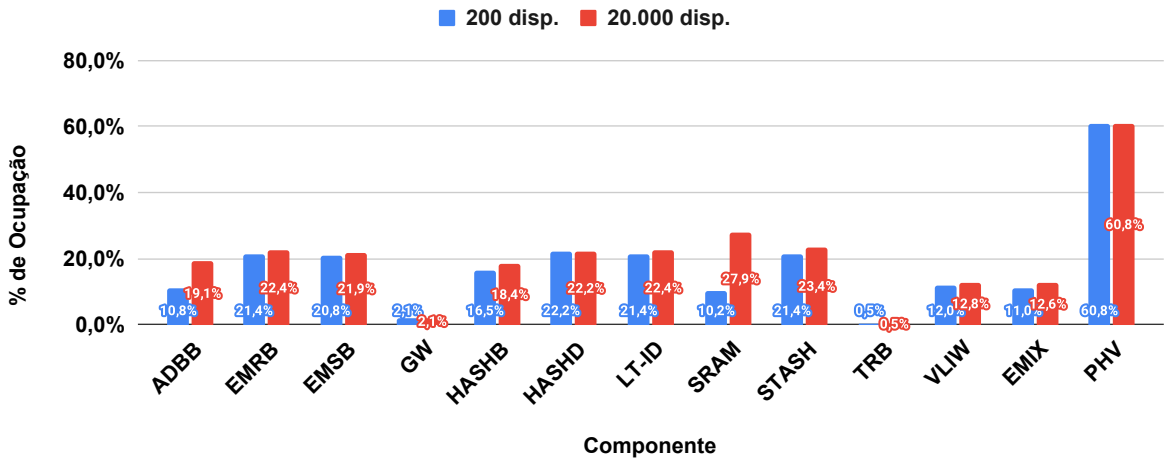

Figura 5.14 – Gráfico de comparação de usos de recurso do *pipeline* Tofino gerado.

Como pode ser observado, o ADBB e a SRAM tiveram um aumento significativo quando definido um *pipeline* com capacidade de verificação de 20.000 dispositivos contra um *pipeline* capaz de verificar até 200 dispositivos. Isso ocorre devido ao uso de mais SRAM implicar na utilização de mais estágios para poder realizar as buscas em uma mesma tabela, aumentando o uso do barramento de resultado dessas buscas, visto que os barramentos de outros estágios também precisarão ser utilizados.

Já sobre o PHV, é importante destacar que não houve alteração no uso, mesmo com a replicação da tabela em estágios diferentes. O PHV é um recurso crítico para a implantação de funções no *switch*, representando a quantidade de operações diferentes que um *switch* programável pode desempenhar em seu plano de dados. Ainda assim, mesmo com o uso de 60,8% de PHVs, o *switch* pode receber outras funções para operação, desde que não tão complexas como operações criptográficas.

A exemplo, a simples inclusão de 2 tabelas no *pipeline* para o processamento de encaminhamento L2 (baseado no MAC de destino) e encaminhamento L3 (baseado em *longest-prefix match*, LPM, em redes IPv4) com 20480 registros disponíveis em cada tabela não causa um impacto tão significativo nos recursos do *switch*, conforme Tabela 5.8. É importante notar que recursos que não são utilizados, como a *Ternary Content-Addressable Memory* (TCAM) e o *Ternary Match Input Crossbar* (TMIX), passam a ser utilizados para os resultados de busca por prefixo (LPM) do encaminhamento L3.

Assim como o caso do verificador baseado em x86, há de se considerar cenários com tolerância de tempo para respostas de provas de atestação e rodadas sobrepostas. No entanto, da forma que o código P4 foi estruturado (conforme descrito na seção 4.1), o *switch* não precisa guardar informações referente à rodada, bastando utilizar as informações que são enviadas no cabeçalho do pacote para calcular e verificar a prova, não variando o uso de armazenamento (em SRAM e PHVs) para qualquer pacote de qualquer

Componente	Pipeline sem encaminhamento L2/L3			Pipeline com encaminhamento L2/L3	
	200 dispositivos	20 000 dispositivos	Variação percentual	20 000 dispositivos	Variação percentual (sobre 20 000)
ADBB	10,8%	19,1%	**76,9%**	19,5%	**2,1%**
EMRB	21,4%	22,4%	**4,7%**	22,9%	**2,2%**
EMSB	20,8%	21,9%	**5,3%**	23,4%	**6,8%**
GW	2,1%	2,1%	**0,0%**	3,1%	**47,6%**
HASHB	16,5%	18,4%	**11,5%**	19,3%	**4,9%**
HASHD	22,2%	22,2%	**0,0%**	22,2%	**0,0%**
LT-ID	21,4%	22,4%	**4,7%**	24,5%	**9,4%**
SRAM	10,2%	27,9%	**173,5%**	29,3%	**5,0%**
STASH	21,4%	23,4%	**9,3%**	24,0%	**2,6%**
TCAM	0,0%	0,0%	**0,00%**	13,9%	**0,00%**
TRB	0,5%	0,5%	**0,0%**	1,0%	**100,0%**
VLIW	12,0%	12,8%	**6,7%**	13,3%	**3,9%**
EMIX	11,0%	12,6%	**14,5%**	13,1%	**4,0%**
TMIX	0,0%	0,0%	**0,00%**	1,0%	**0,00%**
PHV	60,8%	60,8%	**0,0%**	61,3%	**0,8%**

Tabela 5.8 – Impacto de recursos no *switch* Tofino para função de verificação de 200 ou 20.000 dispositivos, com verificação adicional do impacto de funções de encaminhamento L2/L3 no mesmo *pipeline*.

rodada de atestação. Ou, em outra perspectiva, o *switch* não precisa manter um estado para cada rodada.

Como o responsável por solicitar a prova é o controlador, ele deve aguardar a resposta da atestação vinda do *switch*, indicando se uma atestação foi válida. Ao receber essa informação, o controlador pode manter uma *flag* indicando que aquele dispositivo foi válido para aquela rodada. Ao iniciar cada rodada, o controlador pode realizar uma varredura da rodada que saiu da janela de tolerância para reagir aos provadores que falharam em suas atestações.

5.4 Discussões

Considerando os dados levantados nos experimentos, alguns pontos podem ser discutidos quanto a estratégias de adoção desta técnica. A primeira discussão diz respeito à alocação do verificador dentro de uma rede de data center tradicional, baseada na topologia *Fat Tree* e dividida entre as camadas de núcleo, agregação e acesso. A Figura 5.15 ilustra um exemplo dessa topologia.

Nesse cenário, também considera-se os roteadores da camada de núcleo como dispositivos de redes programáveis, mas com funções mais especializadas para alta demanda e, consequentemente, com mais recursos disponíveis.

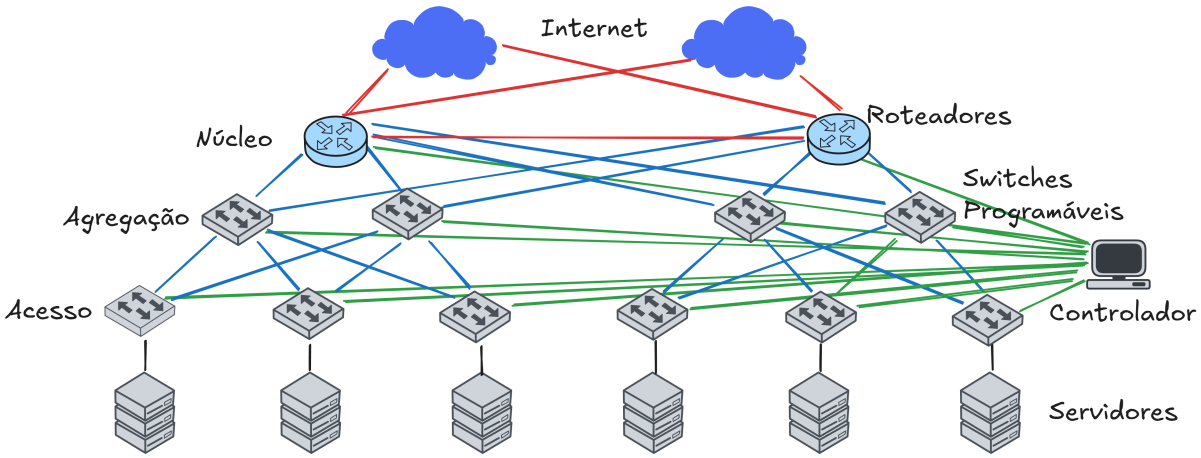

Figura 5.15 – Topologia de rede *Fat Tree* utilizada em data centers com a presença de *switches* programáveis.

A função de verificador poderia ser distribuída entre os *switches* da camada de acesso, onde cada um possui, em média, de 100 a 200 dispositivos (provadores conectados a ele (incluindo dispositivos de processamento e armazenamento). Nesse caso, os *switches* da camada de acesso teria uma ocupação considerável dos seus PHVs, possivelmente inviabilizando que outras aplicações mais complexas, como Firewall, IDS/IPS e cifração de tráfego, fossem realizadas por estes. Além disso, a redução de SRAM acumulada disponível ao longo de toda a rede (isto é, considerando a soma de toda SRAM disponível em todos os dispositivos programáveis da rede), necessária para funções como *In-network Caching* e *Network-assisted Machine Learning*, seria maior, pois a compilação do *pipeline* alocaria a SRAM disponível para uma função, fosse ela utilizada ou não.

Em uma estimativa conservadora, onde cada *switch* da camada de agregação teria abaixo de si outros 30 *switches* da camada de acesso (estimando-se *switches* de 32 portas com *dual-home*, onde cada *switch* da camada inferior é conectado a 2 *switches* da camada superior para redundância), cada *switch* da camada de agregação teria de 3000 a 6000 dispositivos para verificar. Conforme medições feitas e comparadas na Tabela 5.6, o tempo médio de verificação de todos estes dispositivos ainda estaria abaixo de 30 milissegundos em verificadores baseados em PDP, mas com pouco impacto no PHV total disponível se considerarmos que os *switches* da camada de acesso estariam livres para outras funções.

Assim sendo, a alocação dessa função de atestação em camadas mais altas da rede de data center centraliza as operações e reduz o consumo de recursos na perspectiva geral da rede. Além disso, pode ser mais eficiente que a atestação realizada por um verificador central, a qual ocuparia recursos que poderiam ser alugados aos *tenants* do provedor de nuvem.

Buscou-se a redução do impacto de funções de busca em memória e disco pelo

verificador baseado em x86 nos experimentos, justamente para não haver impactos que poderiam variar em decorrência de utilização de banco de dados especializados, *caching* e o tipo de dispositivo de armazenamento secundário utilizado. A utilização destas tecnologias poderiam aumentar o tempo de busca dos dados apresentados na Tabela 5.4.

As atestações frequentes e periódicas reduzem a janela que um atacante possui para poder fazer modificações no sistema e obter resultados sem ser detectado. É importante relembrar que no modelo adversarial proposto, o atacante não é capaz de definir a configuração que o mecanismo de atestação coletará para o cálculo do *hash* da configuração do dispositivo (c_i) para o cálculo da prova ($p_{i,r}$).

Quanto à evasão da detecção por meio de uma resposta forjada pelo atacante, considerando um período entre rodadas de atestação de 1 segundo e que um atestador leve em média 270ms para calcular e responder a prova de atestação com o ID da rodada (ρ) provido, o atacante possui aproximadamente 730ms para desfazer as alterações e solicitar a prova correta para o mecanismo de atestação. Mesmo que o atacante consiga desfazer as alterações dentro do tempo previsto e esquivar a detecção, o tempo para extrair resultados do comprometimento do dispositivo é consideravelmente reduzido em comparação a uma atestação sob-demanda.

Quanto à comparação com métodos de Atestação Remota Coletiva (CRA), a verificação granular realizada pelos *switches* disponíveis na rede programável possibilita a rápida identificação dos dispositivos atacados e afetados pelo ataque, permitindo ações de reação mais rápidas e precisas.

5.5 Considerações do capítulo

Neste capítulo foram apresentados os experimentos realizados para coleta de métricas de desempenho e impacto de recursos para a técnica de atestação proposta. Foram apresentados 2 experimentos de forma a verificar o desempenho de todo o cenário para a atestação de um único dispositivo, sendo um experimento com a verificação realizada por uma implementação baseada em x86 e o outro experimento com a verificação realizada por uma implementação baseada em hardware programável Tofino.

Também foram apresentados outros três experimentos com o objetivo de se avaliar a escalabilidade de cada um dos componentes do cenário: o verificador baseado em x86, o verificador baseado em PDP e o provador baseado na tecnologia ARM TrustZone. Também foram levantados e discutidos o uso de recursos em cada uma das arquiteturas de implementação do verificador comparadas.

Por fim, foi realizada uma discussão acerca do posicionamento do verificador

baseado em PDP em uma topologia de data center, comparando o impacto em recursos disponíveis na rede para funções em PDP e a quantidade de dispositivos alocados sob cada *switch*.

A seguir, serão discutidos possíveis trabalhos futuros para expandir as inveti-gações realizadas para a redução da janela de atestação, buscando evitar ataques do tipo *TOCTTOU*.

6 Conclusão e trabalhos futuros

O trabalho propôs um método de atestação remota em um ambiente de data center com plano de dados programável, reduzindo o uso de recursos de CPU, memória e rede de um controlador central.

Em comparação a abordagens de CRA, a técnica de atestação remota proposta permite a atestação de dispositivos individuais em uma rede, viabilizando sua rápida detecção e isolamento, em contraste a uma atestação geral de todo o conjunto de dispositivos. A janela de atestação se tornou curta e frequente o suficiente para dificultar ataques do tipo *TOCTTOU* em comparação à atestação realizada sob-demanda e sem causar uma negação de serviço nos dispositivos atestados.

Em uma rede com plano de dados programável, o método de cálculo da prova e da verificação poderia ser implementado tanto no ASIC do plano de dados do *switch*, quanto em um servidor com processador x86. Com isso, este trabalho verificou qual abordagem é mais eficiente em relação à frequência de atestações, demonstrando que a atestação utilizando PDP pode reduzir em média 151,6 vezes a janela de tempo de atestação de dispositivos quando comparado com o verificador baseado em x86.

Ainda assim, esta proposta é uma exploração não exaustiva de um método de atestação e podem haver trabalhos futuros como os que serão discutidos a seguir.

6.1 Melhorias à implementação do provador

O provador implementado para os experimentos utilizando a plataforma OPTEE possui algumas limitações que requerem investigações e esforços adicionais para torná-lo efetivo em um cenário prático.

Como primeira limitação, denota-se a necessidade de um canal de comunicação seguro entre o mecanismo de atestação presente no *Secure World* (SW) do OPTEE e o verificador baseado em x86 ou o controlador no cenário com o verificador baseado em PDP. Considerando que a plataforma OPTEE é restritiva quanto aos recursos de memória disponíveis no SW, não foi possível elaborar uma implementação que aplique esse canal seguro de comunicação inicial.

Para a implementação deste canal, seria necessário estender a capacidade de memória que a plataforma disponibiliza para o SW e utilizar a implementação de um algoritmo de cifração presente no SW para realizar a comunicação com o controlador,

de forma que uma chave pré-compartilhada e configurada manualmente ou um algoritmo de chave pública utilizando uma infraestrutura de chave pública privativa do data center fosse utilizado para prover este canal.

Além disso, o provador implementado para os experimentos não coleta as configurações do dispositivo para calcular a prova de atestação, sendo esse passo realizado nos experimentos com um valor nulo representando a configuração do dispositivo. Embora o uso deste valor nulo não tenha impactos nos experimentos realizados, uma implementação em ambiente real requer que o mecanismo de atestação colete essas informações de acordo com a política definida.

Para a implementação desta coleta de estado do dispositivo, será necessário estender a implementação do OPTEE de forma a adicionar drivers assinados no SW que possam realizar o acesso a dispositivos como um *Trusted Platform Module* (TPM) embarcado no servidor, ou em arquivos e configurações relevantes do sistema operacional em execução, mas sem a intermediação do sistema operacional em execução no *Normal World* (NW). A interação com o sistema operacional no NW criaria uma superfície de ataque desnecessária e permitiria que a um atacante manipular o cálculo da prova de verificação.

6.2 Melhorias à implementação do verificador baseado em PDP

Na implementação realizada para o verificador baseado na arquitetura Tofino, a inserção das informações dos dispositivos em registros de tabelas ocorrem em dois momentos diferentes: ao realizar a construção da matriz de estado inicial para o início do cálculo da prova, e ao realizar a finalização da matriz de estado calculada após as rodadas com os valores utilizados para construir a matriz inicial.

Essa duplicidade de informações onera consideravelmente o uso de memória estática (SRAM) do *switch* programável, possivelmente reduzindo a quantidade máxima de dispositivos que podem ser atestados por *switch*. Como trabalho futuro, é interessante investigar uma reestruturação do código de forma a executar a construção e finalização da matriz com uma mesma *action* do *pipeline*, o que permitiria reduzir a quantidade de SRAM necessária para construção do *pipeline* e possivelmente aumentando a quantidade máxima de dispositivos atestados por *switch*.

6.3 Implementação em outros planos de dados programáveis

Switches programáveis são apenas um dos tipos de planos de dados programável existentes atualmente. Cada vez mais a programabilidade de dispositivos alavanca a

inovação aberta de soluções, tornando possível a criação de aplicações otimizadas para diversos cenários. Com isso, outros planos de dados programáveis poderiam ser investigados como alternativas tecnológicas a um verificador baseado em CPUs.

Por exemplo, *SmartNICs* são uma tecnologia que poderia acelerar esse processo de verificação de provas em-rede, desonerando a CPU de provadores ou do controlador de desempenhar essa função e liberando-o para outras atividades de propósito geral. Um verificador alocado em uma *SmartNIC* acoplada a um servidor poderia ter um desempenho similar ao do PDP utilizado neste trabalho, permitindo liberar o núcleo da rede programável para outras funções de rede e até para operações de *In-Network Computing*, como aprendizado de máquina.

Do ponto de vista de uma *SmartNIC* embarcada em um provador, uma interface baseada em SPDM que permita o acesso às configurações relevantes para a atestação poderia permitir atestações compatíveis com a velocidade de comunicação deste protocolo, reduzindo a janela de atestação ao nível de microssegundos e garantindo uma segurança maior para a rede.

Outra abordagem possível, considerando a atestação realizada por provedores de nuvem como Azure e AWS, é a carga de trabalho de *tenants* serem atestadas como uma prestação de serviço (algo como um *Attestation-as-a-Service*). Isso poderia ser realizado por meio de tecnologias de enclave como Intel SGX para o provador em memória segura, e o processamento em plano de dados programável diretamente no *kernel* do sistema operacional baseado nas tecnologias *eBPF* (*extended Berkeley Packet Filter*) (EBPF.IO, 2024) e *XDP* (*eXpress DataPath*) (VISOR, 2024), ou *DPDK* (*Dataplane Development Kit*) (FOUNDATION, 2024). Da mesma forma, essas tecnologias poderiam ser utilizadas para realizar a verificação de provas diretamente *in-kernel* (no caso do conjunto eBPF/XDP) ou com travessia otimizada para o *userspace* (com a tecnologia DPDK).

6.4 Avaliação da técnica de atestação remota proposta em outros cenários

Embora o cenário foco deste trabalho seja data centers, uma técnica de atestação remota similar à aqui proposta poderia ser avaliada para cenários de redes celulares e IoT, como citado em algumas partes do texto.

No caso do cenário de redes celulares, os dispositivos de processamento de borda poderiam ser atestados por uma rede programável disponível na infraestrutura de rede de provedores de rede metropolitanas, assegurando a conformidade de operação de

estações rádio base.

Já no cenário de IoT, originalmente explorado pela abordagem de CRA, os dispositivos poderiam ser atestados por um IoT Gateway com plano de dados programável ou por meio de outros dispositivos de redes programáveis presentes na infraestrutura de um provedor de rede metropolitana, por exemplo. Para o protocolo proposto, não há diferença se o dispositivo está diretamente conectado ao verificador ou com outros equipamentos de comutação intermediários, como gateways dos padrões *Zigbee* ou *LoRaWAN*, desde que a comunicação com o verificador possa ocorrer por meio do protocolo *Ethernet*.

Em ambos os casos, experimentos melhor direcionados a esses cenários precisariam ser elaborados e executados, de forma a obter métricas que reflitam claramente seus parâmetros. Por exemplo, dispositivos IoT dificilmente conseguiriam executar uma solução completa baseada em OP-TEE devido a restrições de memória, devendo ser adotados outros critérios para a memória segura, bem como janelas de atestações maiores para evitar negações de serviço. Além disso, como demonstrado neste trabalho, um verificador baseado em x86 levaria minutos para verificar uma escala de milhões de dispositivos, enquanto um verificador baseado em PDP não seria capaz de escalar a essa quantidade de dispositivos, tornando necessária a investigação de outras abordagens mais especializadas para esse tipo de cenário.

6.5 Outras abordagem de atestação remota em PDP

A abordagem de atestação deste trabalho foi baseada em uma cifra de fluxo, mas esse tipo de cifra não é comumente utilizado em cenários de atestação remota. São mais comuns o uso de hashes criptográficos (como SHA-256) e algoritmos de assinatura digital para prover o resumo e a autenticidade das provas, respectivamente. O uso da assinatura digital traz consigo a necessidade da instalação de uma Autoridade Certificadora para a emissão de certificados aos dispositivos a serem atestados, além do seu processo de verificação ser computacionalmente mais pesado do que o cálculo de uma autenticação de mensagem baseada em *hash*.

Além disso, esses algoritmos de assinatura digital e hash não são adequados para o uso em plano de dados programável devido ao tamanho dos parâmetros ou a quantidade de rodadas necessárias para o cálculo. Assim sendo, caso decida-se por explorar adiante o uso de PDP para soluções de atestação remota, outros algoritmos que provêem autenticidade poderiam ser investigados. A exemplo, um algoritmo recentemente proposto chamado “P4Chaskey” (FRANCISCO *et al.*, 2024) pode ser um candidato interessante para o cenário de IoT, devido à sua baixa carga computacional e poucas rodadas em PDP.

Algoritmos que ainda não foram investigados em PDP, como os da família SHAKE e TurboSHAKE ou da família BLAKE, poderiam ser avaliados como alternativas para cenários de requisitos de segurança mais altos, como cenários de missão crítica.

6.6 Avaliação formal de segurança do protocolo proposto

O algoritmo Forro14 pode prover outros níveis de segurança a depender da quantidade de rodadas realizadas, o que poderia ajustar os custos de processamento para diversos cenários. Assim como o algoritmo ChaCha possui as versões de 8, 12 e 20 rodadas, que proveem maior resistência a ataques de recuperação de chave, o algoritmo Forro possui as versões com 6, 10 e 14 rodadas que proveem os mesmos níveis de segurança.

Esses algoritmos são investigados e criptoanalisados na literatura para verificar seu nível de segurança provido. Embora o cálculo da prova de atestação proposto neste trabalho utilize apenas valores públicos e a chave secreta, ainda é necessário conduzir uma análise de segurança aprofundada para garantir que parâmetros escolhidos não viabilizariam ataques de descoberta da chave. Portanto, é necessário que um trabalho futuro conduza uma análise de segurança mais detalhada do protocolo proposto para que haja formas de mitigar possíveis ataques ao cálculo da prova e descoberta da chave secreta.

Referências

3RD, D. E. E.; JONES, P. *US Secure Hash Algorithm 1 (SHA1)*. RFC Editor, 2001. RFC 3174. (Request for Comments, 3174). Disponível em: <https://www.rfc-editor.org/info/rfc3174>. Citado na página 25.

AMBROSIN, M.; CONTI, M.; LAZZERETTI, R.; RABBANI, M. M.; RANISE, S. Collective remote attestation at the internet of things scale: State-of-the-art and future challenges. *IEEE Communications Surveys & Tutorials*, IEEE, v. 22, n. 4, p. 2447–2461, 2020. Citado 3 vezes nas páginas 19, 28 e 44.

AMD. *AMD Secure Encrypted Virtualization (SEV)*. 2023. Disponível em: <https://www.amd.com/pt/developer/sev.html>. Acesso em: 30/09/2023. Citado na página 19.

ARCISZEWSKI, S. *XChaCha: eXtended-nonce ChaCha and AEAD_XChaCha20_Poly1305*. [S.l.], 2020. Work in Progress. Disponível em: <https://datatracker.ietf.org/doc/draft-irtf-cfrg-xchacha/03/>. Citado na página 104.

ARM. *TrustZone for Cortex-M*. 2023. Disponível em: <https://www.arm.com/technologies/trustzone-for-cortex-m>. Acesso em: 30/09/2023. Citado na página 19.

AUMASSON, J.-P.; HENZEN, L.; MEIER, W.; PHAN, R. C.-W. Sha-3 proposal BLAKE. *Submission to NIST*, v. 92, p. 194, 2008. Citado na página 25.

AUMASSON, J.-P.; MEIER, W.; PHAN, R. C.-W.; HENZEN, L. *The Hash Function BLAKE*. 1st. ed. [S.l.]: Springer Publishing Company, Incorporated, 2016. ISBN 3662525976. Citado na página 26.

AWS. *Cryptographic attestation*. 2023. Disponível em: <https://docs.aws.amazon.com/enclaves/latest/user/set-up-attestation.html>. Acesso em: 30/09/2023. Citado na página 21.

BANKS, A. S.; KISIEL, M.; KORSHOLM, P. Remote attestation: a literature review. *arXiv preprint arXiv:2105.02466*, 2021. Citado 2 vezes nas páginas 28 e 33.

BERNSTEIN, D. J. The poly1305-aes message-authentication code. In: GILBERT, H.; HANDSCHUH, H. (Ed.). *Fast Software Encryption*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005. p. 32–49. ISBN 978-3-540-31669-5. Citado na página 26.

BERNSTEIN, D. J. Chacha, a variant of salsa20. 01 2008. Citado na página 26.

BERNSTEIN, D. J. The salsa20 family of stream ciphers. In: _____. *New Stream Cipher Designs: The eSTREAM Finalists*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008. p. 84–97. ISBN 978-3-540-68351-3. Disponível em: <https://doi.org/10.1007/978-3-540-68351-3_8>. Citado na página 26.

BERNSTEIN, D. J. *et al.* Chacha, a variant of salsa20. In: CITESEER. *Workshop record of SASC*. [S.l.], 2008. v. 8, p. 3–5. Citado na página 93.

BERTONI, G.; DAEMEN, J.; PEETERS, M.; ASSCHE, G. V. Keccak. In: _____. *Advances in Cryptology – EUROCRYPT 2013*. Springer Berlin Heidelberg, 2013. p. 313–314. ISBN 9783642383489. Disponível em: <http://dx.doi.org/10.1007/978-3-642-38348-9_19>. Citado na página 26.

BIHAM, E.; DUNKELMAN, O. A framework for iterative hash functions-haifa. *IACR Cryptology ePrint Archive*, v. 2007, p. 278, 01 2007. Citado na página 25.

BOSSHART, P.; DALY, D.; GIBB, G.; IZZARD, M.; MCKEOWN, N.; REXFORD, J.; SCHLESINGER, C.; TALAYCO, D.; VAHDAT, A.; VARGHESE, G. *et al.* P4: Programming protocol-independent packet processors. *ACM SIGCOMM Computer Communication Review*, ACM New York, NY, USA, v. 44, n. 3, p. 87–95, 2014. Citado 3 vezes nas páginas 22, 90 e 92.

CHEN, X. Implementing aes encryption on programmable switches via scrambled lookup tables. In: *Proceedings of the Workshop on Secure Programmable Network Infrastructure*. [S.l.: s.n.], 2020. p. 8–14. Citado 2 vezes nas páginas 90 e 93.

CONTI, M.; KALIYAR, P.; LAL, C. Censor: Cloud-enabled secure iot architecture over sdn paradigm. *Concurrency and Computation: Practice and Experience*, Wiley Online Library, v. 31, n. 8, p. e4978, 2019. Citado 2 vezes nas páginas 27 e 28.

COSTA, F. G. Pipo-tg: parameterizable high performance traffic generation. Universidade Federal do Pampa, 2023. Citado na página 98.

COUTINHO, M. Design, diffusion, and cryptanalysis of symmetric primitive. 2023. Citado 2 vezes nas páginas 44 e 102.

COUTINHO, M. *forro_cipher*. 2023. <https://github.com/murcoutinho/forro_cipher>. Online: Acesso em 28-05-2024. Citado na página 98.

COUTINHO, M.; PASSOS, I.; BORGES, F. The design and implementation of xforró14-poly1305: a new authenticated encryption scheme. In: *Anais do XXIII Simpósio Brasileiro de Segurança da Informação e de Sistemas Computacionais*. Porto Alegre, RS, Brasil: SBC, 2023. p. 456–469. ISSN 0000-0000. Disponível em: <https://sol.sbc.org.br/index.php/sbseg/article/view/27226>. Citado na página 104.

COUTINHO, M.; PASSOS, I.; VÁSQUEZ, J. C. G.; SARKAR, S.; MENDONÇA, F. L. de; JR, R. T. de S.; BORGES, F. Latin dances reloaded: Improved cryptanalysis against salsa and chacha, and the proposal of forró. *Journal of Cryptology*, Springer, v. 36, n. 3, p. 18, 2023. Citado 3 vezes nas páginas 27, 90 e 94.

DAMGÅRD, I. B. A design principle for hash functions. In: SPRINGER. *Conference on the Theory and Application of Cryptology*. [S.l.], 1989. p. 416–427. Citado na página 25.

DANG, H. T.; BRESSANA, P.; WANG, H.; LEE, K. S.; ZILBERMAN, N.; WEATHERSPOON, H.; CANINI, M.; PEDONE, F.; SOULÉ, R. P4xos: Consensus as a network service. *IEEE/ACM Transactions on Networking*, IEEE, v. 28, n. 4, p. 1726–1738, 2020. Citado 2 vezes nas páginas 13 e 92.

DATTA, R.; CHOI, S.; CHOWDHARY, A.; PARK, Y. P4guard: Designing p4 based firewall. In: IEEE. *MILCOM 2018-2018 IEEE Military Communications Conference (MILCOM)*. [S.l.], 2018. p. 1–6. Citado na página 90.

DIOP, A.; LAURENT, M.; LENEUTRE, J.; TRAORÉ, J. Cora: A scalable collective remote attestation protocol for sensor networks. In: *ICISSP*. [S.l.: s.n.], 2020. p. 84–95. Citado na página 27.

DMTF. *Redfish*. 2024. Disponível em: <https://www.dmtf.org/standards/redfish>. Acesso em: 27/09/2024. Citado na página 20.

DMTF. *Security Protocols and Data Models Working Group*. 2024. Disponível em: <https://www.dmtf.org/standards/spdm>. Acesso em: 27/09/2024. Citado na página 21.

DWORKIN, M.; BARKER, E.; NECHVATAL, J.; FOTI, J.; BASSHAM, L.; ROBACK, E.; DRAY, J. *Advanced Encryption Standard (AES)*. [S.l.]: Federal Inf. Process. Stds. (NIST FIPS), National Institute of Standards and Technology, Gaithersburg, MD, 2001. Citado na página 93.

EBPF.IO. *eBPF*. 2024. Disponível em: <https://ebpf.io/>. Acesso em: 26/11/2024. Citado na página 80.

FERNANDES, E. L.; ROTHENBERG, C. E. Openflow 1.3 software switch. *Salao de Ferramentas do XXXII Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuıdos SBRC*, UFSC Florianópolis, Brazil, p. 1021–1028, 2014. Citado 3 vezes nas páginas 22, 90 e 91.

FOUNDATION, L. *DPDK*. 2024. Disponível em: <https://www.dpdk.org/>. Acesso em: 26/11/2024. Citado na página 80.

FRANCISCO, M.; FERREIRA, B.; RAMOS, F. M. V.; MARIN, E.; SIGNORELLO, S. P4chaskey: An efficient mac algorithm for pisa switches. In: *7th European P4 Workshop (EuroP4'24)*. [S.l.: s.n.], 2024. p. –. Citado na página 81.

GAO, Y.; WANG, Z. A review of p4 programmable data planes for network security. *Mobile Information Systems*, Hindawi Limited, v. 2021, p. 1–24, 2021. Citado na página 28.

Gigabyte. *BMC*. 2024. Disponível em: <https://www.gigabyte.com/Glossary/bmc>. Acesso em: 27/09/2024. Citado na página 20.

GOOGLE. *Atestado Remoto de Máquinas Desagregadas*. 2023. Disponível em: <https://cloud.google.com/docs/security/remote-attestation?hl=pt-br>. Acesso em: 30/09/2023. Citado na página 21.

HAUSER, F.; HäBERLE, M.; MERLING, D.; LINDNER, S.; GUREVICH, V.; ZEIGER, F.; FRANK, R.; MENTH, M. A survey on data plane programming with p4: Fundamentals, advances, and applied research. *Journal of Network and Computer Applications*, v. 212, p. 103561, 2023. ISSN 1084-8045. Disponível em: <https://www.sciencedirect.com/science/article/pii/S1084804522002028>. Citado na página 92.

HEIDEKER, A.; SILVA, D.; KLEINSCHMIDT, J. H.; KAMIENSKI, C. Otimizaçao de tráfego iot-lorawan usando programaçao de plano de dados em p4. In: SBC. *Anais do XLI Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos*. [S.l.], 2023. p. 239–252. Citado na página 28.

INTEL. *Intel Software Guard Extensions (Intel SGX)*. 2023. Disponível em: <https://www.intel.com.br/content/www/br/pt/architecture-and-technology/software-guard-extensions.html>. Acesso em: 30/09/2023. Citado na página 19.

INTEL. *Intel Tofino 2*. 2023. Disponível em: <https://www.intel.com/content/www/us/en/products/details/network-io/intelligent-fabric-processors/tofino-2.html>. Acesso em: 30/09/2023. Citado na página 22.

JACQUIN, L.; SHAW, A. L.; DALTON, C. Towards trusted software-defined networks using a hardware-based integrity measurement architecture. In: IEEE. *Proceedings of the 2015 1st IEEE Conference on Network Softwarization (NETSOFT)*. [S.l.], 2015. p. 1–6. Citado na página 27.

JIN, X.; LI, X.; ZHANG, H.; SOULÉ, R.; LEE, J.; FOSTER, N.; KIM, C.; STOICA, I. Netcache: Balancing key-value stores with fast in-network caching. In: *Proceedings of the 26th Symposium on Operating Systems Principles*. [S.l.: s.n.], 2017. p. 121–136. Citado na página 92.

KFOURY, E. F.; CRICHIGNO, J.; BOU-HARB, E. An exhaustive survey on p4 programmable data plane switches: Taxonomy, applications, challenges, and future trends. *IEEE Access*, v. 9, p. 87094–87155, 2021. Citado na página 92.

KREUTZ, D.; RAMOS, F. M.; VERISSIMO, P. E.; ROTHENBERG, C. E.; AZODOLMOLKY, S.; UHLIG, S. Software-defined networking: A comprehensive survey. *Proceedings of the IEEE*, Ieee, v. 103, n. 1, p. 14–76, 2014. Citado 2 vezes nas páginas 22 e 90.

LEAO, H. C.; RIKER, A.; ABELÉM, A. J. Agregaçao e desagregaçao de dados iot em redes definidas por software utilizando p4. In: SBC. *Anais do XIII Workshop de Pesquisa Experimental da Internet do Futuro*. [S.l.], 2022. p. 29–34. Citado na página 28.

LI, G.; ZHANG, M.; LIU, C.; KONG, X.; CHEN, A.; GU, G.; DUAN, H. Nethcf: Enabling line-rate and adaptive spoofed ip traffic filtering. In: IEEE. *2019 IEEE 27th international conference on network protocols (ICNP)*. [S.l.], 2019. p. 1–12. Citado na página 90.

LINARO. *OP-TEE*. 2024. Disponível em: <https://www.trustedfirmware.org/projects/op-tee/>. Acesso em: 24/11/2024. Citado na página 52.

LING, Z.; YAN, H.; SHAO, X.; LUO, J.; XU, Y.; PEARSON, B.; FU, X. Secure boot, trusted boot and remote attestation for arm trustzone-based iot nodes. *Journal of Systems Architecture*, v. 119, p. 102240, 2021. ISSN 1383-7621. Disponível em: <https://www.sciencedirect.com/science/article/pii/S1383762121001661>. Citado na página 18.

MADUREIRA, A. L. R. Iotp: on supporting iot data aggregation through programmable data planes. Universidade Federal da Bahia, 2021. Citado na página 28.

MAHRACH, S.; MJIHIL, O.; HAQIQ, A. Scalable and dynamic network intrusion detection and prevention system. In: SPRINGER. *Innovations in Bio-Inspired Computing and Applications: Proceedings of the 8th International Conference on Innovations in Bio-Inspired Computing and Applications (IBICA 2017) held in Marrakech, Morocco, December 11-13, 2017*. [S.l.], 2018. p. 318–328. Citado na página 90.

MICROSOFT. *Microsoft Azure Attestation*. 2023. Disponível em: <https://azure.microsoft.com/en-us/products/azure-attestation>. Acesso em: 30/09/2023. Citado na página 21.

NIR, Y.; LANGLEY, A. *ChaCha20 and Poly1305 for IETF Protocols*. RFC Editor, 2015. RFC 7539. (Request for Comments, 7539). Disponível em: <https://www.rfc-editor.org/info/rfc7539>. Citado na página 98.

NIST. *SHA-3 standard permutation-based hash and extendable-output functions*. [s.n.], 2015. Disponível em: <http://dx.doi.org/10.6028/NIST.FIPS.202>. Citado 2 vezes nas páginas 25 e 26.

NIST. *NIST Transitioning Away from SHA-1 for All Applications*. 2022. Disponível em: <https://csrc.nist.gov/news/2022/nist-transitioning-away-from-sha-1-for-all-apps>. Acesso em: 27/09/2024. Citado na página 25.

OKADA, A.; KIHARA, S.; OKAZAKI, Y. *Disaggregated Computing Will Change the World*. 2023. Disponível em: <https://www.rd.ntt/e/research/JN202105_13586.html>. Acesso em: 30/09/2023. Citado na página 24.

OLIVEIRA, I.; NETO, E.; IMMICH, R.; FONTES, R.; NETO, A.; RODRIGUEZ, F.; ROTHENBERG, C. E. dh-aes-p4: On-premise encryption and in-band key-exchange in p4 fully programmable data planes. In: *2021 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*. [S.l.: s.n.], 2021. p. 148–153. Citado na página 13.

O'CONNOR, J.; AUMASSON, J.-P.; NEVES, S.; WILCOX-O'HEARN, Z. *BLAKE3: one function, fast everywhere*. 2020. Disponível em: <https://github.com/BLAKE3-team/BLAKE3-specs/blob/master/blake3.pdf>. Acesso em: 30/09/2023. Citado na página 32.

PETERSON, L.; CASCONE, C.; DAVIE, B. *Software-Defined Networks: A Systems Approach*. Systems Approach LLC, 2021. ISBN 9781736472101. Disponível em: <https://sdn.systemsapproach.org/switch.html>. Citado 2 vezes nas páginas e 92.

RIVEST, R. L. *The MD5 Message-Digest Algorithm*. RFC Editor, 1992. RFC 1321. (Request for Comments, 1321). Disponível em: <https://www.rfc-editor.org/info/rfc1321>. Citado na página 25.

ROGAWAY, P.; SHRIMPTON, T. Cryptographic hash-function basics: Definitions, implications, and separations for preimage resistance, second-preimage resistance, and collision resistance. In: ROY, B.; MEIER, W. (Ed.). *Fast Software Encryption*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004. p. 371–388. ISBN 978-3-540-25937-4. Citado na página 25.

SCHOLZ, D.; OELDEMANN, A.; GEYER, F.; GALLENMÜLLER, S.; STUBBE, H.; WILD, T.; HERKERSDORF, A.; CARLE, G. Cryptographic hashing in p4 data planes. In: IEEE. *2019 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*. [S.l.], 2019. p. 1–6. Citado na página 92.

SIVARAMAN, V.; NARAYANA, S.; ROTTENSTREICH, O.; MUTHUKRISHNAN, S.; REXFORD, J. Heavy-hitter detection entirely in the data plane. In: *Proceedings of the Symposium on SDN Research*. [S.l.: s.n.], 2017. p. 164–176. Citado na página 90.

SLINGERLAND, C. *13 Top Cloud Service Providers Globally*. 2024. Disponível em: <https://www.cloudzero.com/blog/cloud-service-providers/>. Acesso em: 27/09/2024. Citado na página 20.

SULTANA, N.; SHANDS, D.; YEGNESWARAN, V. A case for remote attestation in programmable dataplanes. In: *Proceedings of the 21st ACM Workshop on Hot Topics in Networks*. [S.l.: s.n.], 2022. p. 122–129. Citado na página 27.

TCG. *What is a Root-of-Trust (RoT)?* 2024. Disponível em: <https://trustedcomputinggroup.org/about/what-is-a-root-of-trust-rot/>. Acesso em: 27/09/2024. Citado na página 18.

TCG. *What is a Trusted Platform Module (TPM)?* 2024. Disponível em: <https://trustedcomputinggroup.org/about/what-is-a-trusted-platform-module-tpm/>. Acesso em: 27/09/2024. Citado na página 19.

TOKUSASHI, Y.; MATSUTANI, H.; ZILBERMAN, N. Lake: the power of in-network computing. In: IEEE. *2018 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*. [S.l.], 2018. p. 1–8. Citado na página 92.

VIEIRA, M. A.; CASTANHO, M. S.; PACÍFICO, R. D.; SANTOS, E. R.; JÚNIOR, E. P. C.; VIEIRA, L. F. Fast packet processing with ebpf and xdp: Concepts, code, challenges, and applications. *ACM Computing Surveys (CSUR)*, ACM New York, NY, USA, v. 53, n. 1, p. 1–36, 2020. Citado na página 92.

VISOR, I. *XDP eXpress Data Path*. 2024. Disponível em: <https://www.iovisor.org/technology/xdp>. Acesso em: 26/11/2024. Citado na página 80.

WATSEN, K.; ABRAHAMSSON, M.; FARRER, I. *Secure Zero Touch Provisioning (SZTP)*. RFC Editor, 2019. RFC 8572. (Request for Comments, 8572). Disponível em: <https://www.rfc-editor.org/info/rfc8572>. Citado na página 31.

XIONG, Z.; ZILBERMAN, N. Do switches dream of machine learning? toward in-network classification. In: *Proceedings of the 18th ACM workshop on hot topics in networks*. [S.l.: s.n.], 2019. p. 25–33. Citado na página 13.

YAN, Y.; ZHUANG, J.; NEJABATI, R.; SIMEONIDOU, D. P4-enabled smart nic for intra-server network virtualization acceleration. In: *2020 Asia Communications and Photonics Conference (ACP) and International Conference on Information Photonics and Optical Communications (IPOC)*. [S.l.: s.n.], 2020. p. 1–3. Citado na página 25.

YOO, S.; CHEN, X. Secure keyed hashing on programmable switches. In: *Proceedings of the ACM SIGCOMM 2021 Workshop on Secure Programmable network INfrastructure*. [S.l.: s.n.], 2021. p. 16–22. Citado 3 vezes nas páginas 28, 32 e 90.

YOSHINAKA, Y.; TAKEMASA, J.; KOIZUMI, Y.; HASEGAWA, T. On implementing chacha on a programmable switch. In: *Proceedings of the 5th International Workshop on P4 in Europe*. [S.l.: s.n.], 2022. p. 15–18. Citado 3 vezes nas páginas 28, 40 e 94.

ZHENG, C.; RIENECKER, B.; ZILBERMAN, N. Qcmp: Load balancing via in-network reinforcement learning. In: *Proceedings of the 2nd ACM SIGCOMM Workshop on Future of Internet Routing & Addressing*. [S.l.: s.n.], 2023. p. 35–40. Citado na página 92.

ANEXO A – Artigo publicado no SBSeg 2024

Este artigo foi publica nos anais do XXIV Simpósio Brasileiro de Segurança da Informação e de Sistemas Computacionais, ocorrido em 2024 em São José dos Campos/SP[1], páginas 399 a 414, com título "Implementação e avaliação da cifra de fluxo Forro14 em hardware programável Tofino usando a linguagem P4".

A.1 Resumo

O paradigma de redes definidas por software (SDN) habilitou diversas inovações em redes de computadores, principalmente na programabilidade do processamento de pacotes. Neste trabalho, investigou-se a viabilidade e os impactos em recursos computacionais do algoritmo de cifra de fluxo Forro14 em hardware de switch programável Tofino usando a linguagem P4. Para fins de comparação, foi analisado também o algoritmo ChaCha20 quanto a seu desempenho e impacto no mesmo switch. Constatou-se que o algoritmo Forro14 tem um desempenho melhor usando menos recursos que o ChaCha20 para comunicações de até 10 Gbps. Entretanto, quando são adotadas técnicas de paralelização, ChaCha20 tem um desempenho melhor para taxas maiores de dados, mas utilizando mais recursos de processamento do dispositivo que Forro14.

A.2 Abstract

The software-defined networking (SDN) paradigm has enabled several innovations in computer networking, especially in programmable packet processing. This paper investigated the feasibility and impact on computing resources of the Forro14 stream cipher algorithm in the Tofino programmable hardware switch. For comparison purposes, the ChaCha20 algorithm was also analyzed in terms of its performance and impact on the same switch. It was observed that the Forro14 algorithm performs better using fewer resources than ChaCha20 for communications up to 10 Gbps. However, when parallelization techniques are adopted, ChaCha20 performs better for higher data rates but uses more processing resources than Forro14.

[1] <https://sol.sbc.org.br/index.php/sbseg/issue/view/1348>

A.3 Introdução

Nas últimas décadas, as redes de computadores passaram por mudanças de paradigma relevantes para cenários onde a gestão da rede e sua operação possuem maior importância financeira e maior desempenho, como em redes metropolitanas e redes de *data centers*. Essas mudanças caracterizam-se pela separação entre o encaminhamento de pacotes (plano de dados) e a inteligência que define a forma de fazer esse encaminhamento (plano de controle). Esse é o paradigma de redes de computadores definidas por software (*Software-Defined Networking*, SDN) (KREUTZ *et al.*, 2014) que traz a possibilidade de centralizar o controle de uma rede e utilizar software de controle que definem formas inovadoras de encaminhamento, possíveis pela programabilidade do plano de dados usando por exemplo o protocolo OpenFlow (FERNANDES; ROTHENBERG, 2014), ou mais recentemente a linguagem P4.

As tecnologias habilitadoras de SDN trazem diversas novas possibilidades de inovação não só em engenharia de tráfego, processamento de pacotes e otimização de recursos computacionais, mas também em segurança de redes de computadores. Nessa linha, novas formas de trazer garantias de segurança para as redes vêm sendo exploradas em diversas frentes, como detecção de intrusão (MAHRACH *et al.*, 2018), *firewall* (DATTA *et al.*, 2018), detecção de ataque de negação de serviço (SIVARAMAN *et al.*, 2017), IP *Spoofing* (LI *et al.*, 2019) e até mesmo operações de *hash* (YOO; CHEN, 2021) e criptografia (CHEN, 2020) diretamente no encaminhamento de pacotes. Tais implementações possuem impactos no desempenho e nos recursos computacionais dos equipamentos de comutação. Portanto, busca-se soluções que provejam as garantias de segurança com o menor impacto nos recursos e desempenho.

Dentro desse contexto, este trabalho tem por objetivo a implementação e avaliação de uma cifra de fluxo brasileira chamada Forro14 (COUTINHO *et al.*, 2023b) em SDN, de forma a se verificar sua viabilidade em comparação com outra solução de sigilo nos critérios de vazão de dados e ocupação de recursos. Até onde sabemos, esta é a primeira iniciativa de implementar e avaliar tal cifra em um *switch* programável baseado em *hardware* Tofino e usando a linguagem P4 (BOSSHART *et al.*, 2014).

Aplicar esses algoritmos diretamente em equipamentos de encaminhamento de pacotes pode reduzir o uso de recursos de processamento de servidores, permitir a distribuição de processamento através da rede e até aproveitar a posição física de equipamentos para a setorização de processamento de dados. Nesse sentido, este trabalho é parte de um esforço maior para o desenvolvimento de um esquema de atestação remota distribuída e granular que permite a verificação de integridade em um contexto de *data centers* sem a oneração de um verificador central, de forma que os próprios *switches* da rede sejam ca-

pazes de verificar a integridade dos dispositivos conectados a ele. Isso exige que o *switch* seja capaz de calcular *hashes* ou cifras com o menor impacto possível em suas demais operações de encaminhamento e, consequentemente, faz-se necessário usar a linguagem P4.

É necessária a escolha de um algoritmo de cifra nesse contexto, o que nos leva à seguinte pergunta de pesquisa para este trabalho: *"No contexto de redes SDN baseadas em linguagem P4, a cifra de fluxo Forro14 é uma alternativa mais eficiente que a cifra ChaCha20 em uso de recursos (memória, portas etc.) e em vazão de dados?"*.

Para avaliação do algoritmo de cifra de fluxo destacado, foi feita uma implementação utilizando a tecnologia de plano de dados programável em um *switch* Tofino, e uma avaliação de desempenho com geração controlada de tráfego e coleta de métricas de vazão e utilização dos recursos do *switch*. A avaliação foi complementada pela comparação com a implementação de outra cifra de fluxo (ChaCha20), bastante conhecida na literatura e base para o projeto do Forro14.

A.4 Programabilidade do Plano de Dados com a linguagem P4

Um plano de dados programável é uma tecnologia que pode ser compreendida como uma especialização de SDN. Nos primeiros trabalhos com SDN, a abordagem utilizada se baseava no protocolo OpenFlow (FERNANDES; ROTHENBERG, 2014), um protocolo em que fabricantes de *switches* estruturam o *chip* de encaminhamento dos equipamentos de forma que algumas funcionalidades pré-estabelecidas no protocolo possam ser programadas no dispositivo por um controlador central.

Embora isso permita que diversas funcionalidades do *switch* possam ser controladas de forma centralizada, isso ainda limita o processamento de pacotes às definições do *chip* de encaminhamento do *switch* que é construído através de protocolos pré-conhecidos no contexto de rede, como *Ethernet*, IP, entre outros.

Para acelerar a inovação e liberar o operador das limitações de fabricantes de *switches*, surgiu o conceito de plano de dados programável. Nessa tecnologia, o fabricante define um *chip* de encaminhamento que pode ser programado, de forma similar à programação de uma FPGA (*Field Programmable Grid Array*) e trabalha na velocidade de linha do equipamento. O usuário é responsável por definir os protocolos e as operações que o *switch* deverá executar antes da sua operação na rede. O modelo de programação adotado para essa abordagem é o uso de tabelas *Match-Action*, onde o *switch* é compreendido como uma máquina de busca em tabelas e manipulação de dados em estruturas pré-definidas, direcionando os mesmos de acordo com os resultados das buscas.

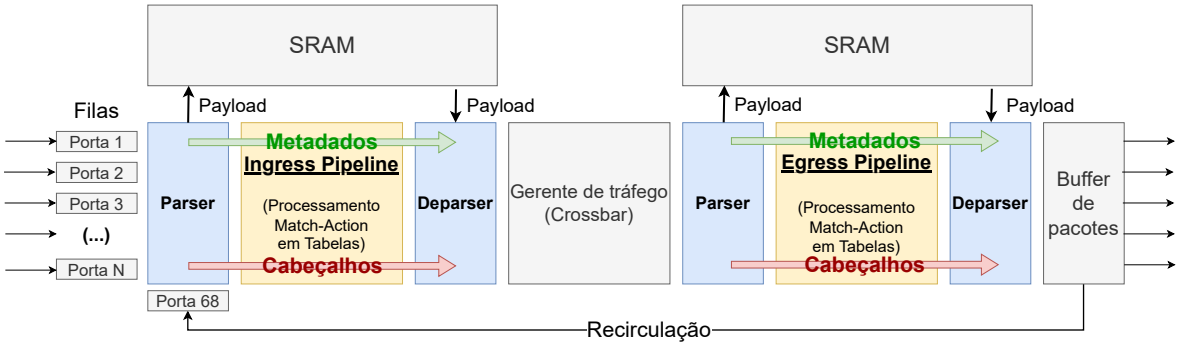

Figura A.1 – Arquitetura referência de um *switch* programável com P4. Fonte: Adaptado de *Systems Approach*. (PETERSON *et al.*, 2021).

Para realizar essa programação do chip, o usuário utiliza uma linguagem de domínio específico conhecida como P4 (*Programming Protocol-independent Packet Processors*) (BOSSHART *et al.*, 2014). O código em alto nível definido pelo usuário é acrescido de uma definição de arquitetura fornecida pelo fabricante do *switch* e inserido em um compilador que gera o código compatível com a programação do *chip*, na forma de bytecode ou JSON (*JavaScript Object Notation*). A Figura A.1 mostra uma arquitetura de referência do tipo *protocol-independent switch architecture* (PISA) que pode ser programada em P4.

A linguagem P4 se tornou um padrão para a programação de processadores de pacotes e é adotada em diversos tipos de plano de dados, como *switches* programáveis, SmartNICs (SCHOLZ *et al.*, 2019) e até plano de dados *in-kernel* como eBPF (VIEIRA *et al.*, 2020).

Com o uso desses planos de dados programáveis, diversas aplicações mais complexas puderam ser implementadas dentro da própria infraestrutura de encaminhamento de dados, o que viabilizou um novo paradigma chamado *in-network computing* (TOKUSASHI *et al.*, 2018), onde diversas operações de computação podem ser executadas diretamente no plano de dados, como algoritmos de consenso de sistemas distribuídos (DANG *et al.*, 2020), *cache* em rede (JIN *et al.*, 2017), aprendizado de máquina (ZHENG *et al.*, 2023), entre muitas outras (KFOURY *et al.*, 2021; HAUSER *et al.*, 2023).

No entanto, a linguagem P4 tem uma série de limitações que tornam desafiadora a implementação de algoritmos não diretamente ligados a tráfego de pacotes. Um exemplo sempre é dado: P4 não oferece suporte a laços de repetição. Uma nova iteração sobre dados em processamento é obtida com a reintrodução (recirculação) do pacote que contém tais dados. Por isso, implementações de cifradores como o AES e o ChaCha20 são complexas e exigem técnicas que são dependentes do hardware onde o processo será executado.

A.5 Cifras de Fluxo

Com o objetivo de prover sigilo em uma comunicação, algoritmos de criptografia podem ser utilizados para garantir que apenas detentores de uma chave criptográfica possam conhecer o conteúdo original de uma mensagem cifrada, sendo computacionalmente inviável a decifração do texto cifrado sem o conhecimento da chave.

Dentre os algoritmos de criptografia simétrica existentes, dois grandes grupos são definidos: *algoritmos de cifra em blocos*, nos quais blocos de dados de tamanho fixo são cifrados, e *algoritmos de cifra de fluxo*, nos quais os bits são cifrados um a um usando um fluxo gerado a partir de uma chave.

No contexto de algoritmos de cifra em blocos utilizados em redes programáveis, destaca-se a implementação do relevante algoritmo AES (DWORKIN *et al.*, 2001), com desempenho satisfatório em diversas aplicações. O algoritmo foi implementado utilizando uma técnica de *scrambled lookup tables*, onde os conteúdos das *S-Boxes* são previamente calculados de forma a simplificar partes do processo de cifração e decifração, usando buscas em tabela (CHEN, 2020).

Já no contexto de cifras de fluxo, um algoritmo comumente utilizado é o ChaCha20 (BERNSTEIN *et al.*, 2008). Esse algoritmo utiliza uma chave e um *nonce* para gerar uma matriz de estado 4x4 com índices de 32 bits, totalizando 512 bits, que é utilizada na cifração do conteúdo da mensagem em claro com uma operação de XOR bit a bit. Para que o algoritmo seja seguro, é necessário que cada matriz de estado calculada seja única, não permitindo que uma mesma matriz de estado seja utilizada para cifrar dois conjuntos de 512 bits da mensagem. Para isso, são utilizados *nonces* e contadores, com o *nonce* variando a cada mensagem e o contador variando a cada conjunto de 512 bits de uma mesma mensagem.

Esse algoritmo utiliza 20 rodadas de operações divididas em funções chamadas QR (*Quarter Round*), onde 4 QRs compõem uma rodada do algoritmo. Os QRs são operações de somas módulo 2^{32} e XORs seguidos de rotações de bits (conjunto conhecido por ARX, *Add-Rotate-Xor*) que causam a difusão das modificações nos bits por toda a matriz de estado. Os QRs são aplicados em colunas da matriz de estado em rodadas ímpares e nas diagonais em rodadas pares. A Figura A.2 demonstra as operações de cada QR e como os elementos de 32 bits da matriz de estado são coletados para cada QR. Os elementos C_i, K_i, T_i e N_i da matriz inicial correspondem, respectivamente, a **constante, chave, contador** e **nonce**. A matriz final após 20 rodadas é somada (módulo 2^{32}) elemento a elemento com a matriz inicial para produzir os bits do fluxo de chave (*keystream*) que será usado para fazer o XOR com um conjunto de 512 bits do texto em claro (mensagem). Para os próximos conjuntos de 512 bits da mensagem, uma nova matriz de estado deve

ser calculada com um incremento em um contador, gerando resultados diferentes a cada nova iteração.

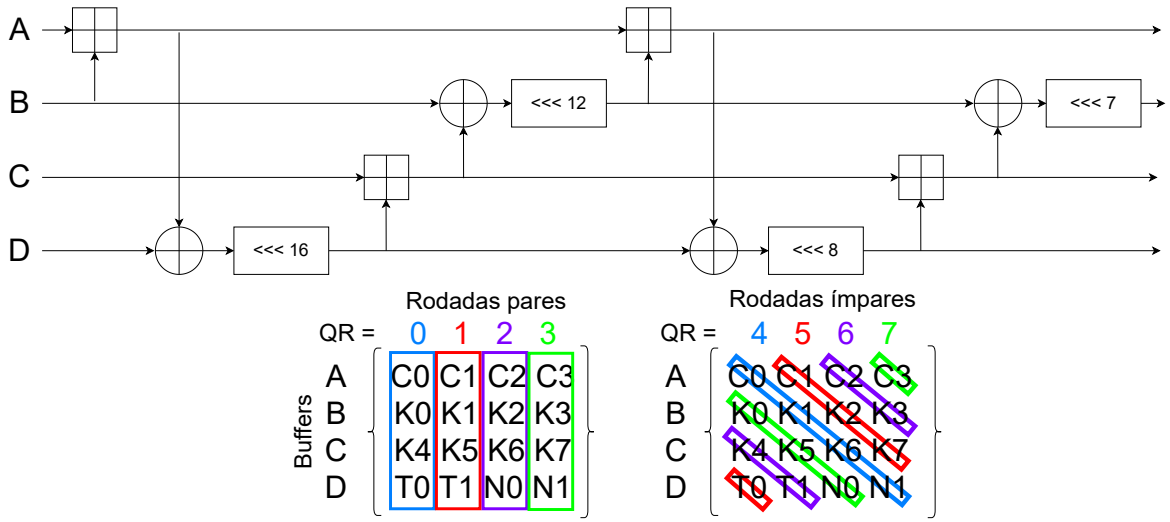

Figura A.2 – Esquema do Quarter Round e do uso dos elementos de 32 bits da matriz de estados no ChaCha20.

No contexto de SDN, o algoritmo ChaCha20 foi implementado em *switches* programáveis de forma paralela, sendo capaz de cifrar conjuntos de 512 a 3072 bits com vazão de 50 a 100% maior que a da implementação do algoritmo de cifra de blocos AES com chave de 256 bits (YOSHINAKA *et al.*, 2022).

Outro algoritmo de cifra de fluxo proposto recentemente é o chamado Forro14 (COUTINHO *et al.*, 2023b). Esse algoritmo utiliza uma estrutura muito similar à do ChaCha20 e adiciona uma técnica chamada *polinização* para aumentar a difusão da matriz de estado calculada, de forma a torná-la mais segura contra ataques de análise diferencial. Essa técnica torna a implementação do algoritmo sequencial, visto que a polinização cria uma dependência ao QR atual utilizar um elemento alterado no QR anterior, limitando a velocidade de operação do algoritmo por não permitir o aproveitamento de recursos de paralelização de mais baixo nível que possam estar disponíveis em certos dispositivos. A Figura A.3 mostra as operações de cada QR do Forro14 e como os valores da matriz de estado são coletados a cada QR. Observa-se uma reorganização dos elementos C_i, K_i, T_i e N_i da matriz inicial em comparação ao ChaCha20. O *buffer* E (pólen) é coletado de forma circular na primeira linha da matriz, começando pelo campo K_3 na última coluna.

Devido à técnica de polinização (parâmetro E), o algoritmo Forro14 requer menos processamento para ter uma segurança equivalente à do algoritmo ChaCha20, com uma redução de aproximadamente 30% no número de rodadas. No Forro14, são utilizadas mais operações de soma e menos operações de rotação de bits e XOR, tendo sido as rotações escolhidas de forma experimental para maximizar a segurança contra ataques de análise diferencial.

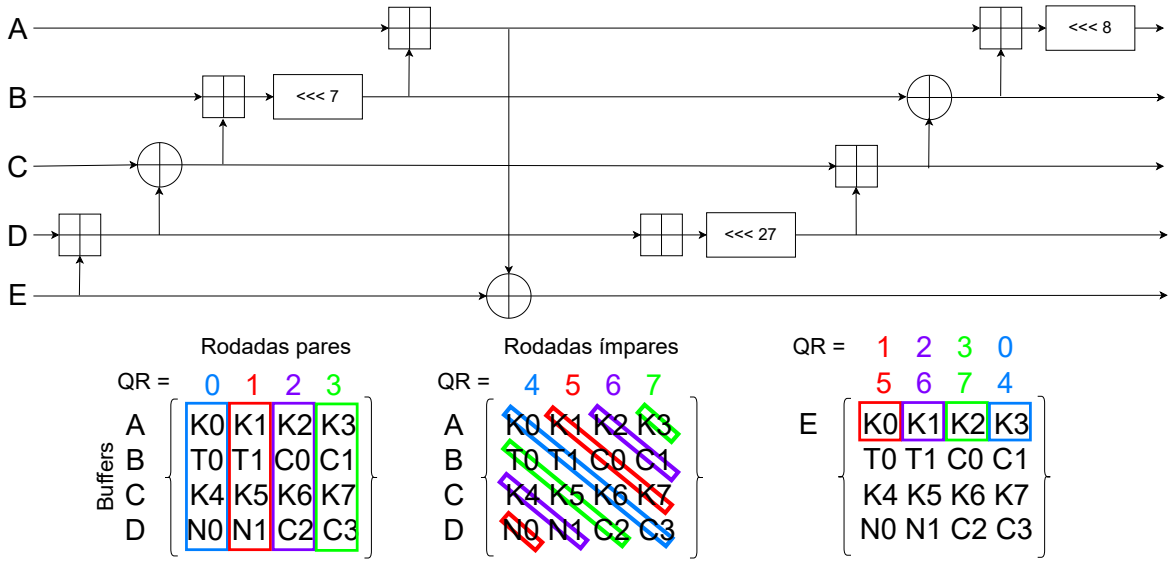

Figura A.3 – Esquema do *Quarter Round* e do uso dos elementos de 32 bits da matriz de estados do Forro14.

A.6 Implementação e avaliação do Forro14

Neste trabalho foi feita a implementação do algoritmo Forro14[2] e a avaliação de seu desempenho em *switches* programáveis em comparação com implementações paralela e sequencial do algoritmo ChaCha20.

A plataforma de SDN escolhida foi a ASIC (*Application-Specific Integrated Circuit*) Tofino da Intel. Esse ASIC programável permite o uso de *pipelines* de entrada e saída de 12 estágios, totalizando 24 estágios de processamento. O *switch* que embarca esse ASIC é um Edgecore Wedge 100BF-32X modelo DCS800 com 32 portas de 100 Gbps QSFP28.

A plataforma Tofino possui diversas limitações em sua configuração para que os estágios de pipeline trabalhem na velocidade de 100 Gbps sem gargalos. Como premissa da arquitetura de *switches*, não é possível utilizar estruturas de *loop* no processamento de pacotes, sendo necessária a recirculação de um mesmo pacote para esse fim. Com isso, o *switch* disponibiliza 2 portas internas utilizadas para recirculação de pacotes. Também é possível definir portas físicas como portas de *loopback* para recirculação.

A estrutura do pipeline do ChaCha20 implementado de forma paralela permite aproveitar uma travessia de um dos dois *pipelines* (*Ingress* ou *Egress*) para realizar 4 QRs juntos, visto que não há interdependência entre os resultados dos QRs. Portanto, com uma travessia no *pipeline* de entrada (*Ingress*) e uma travessia no *pipeline* de saída (*Egress*), é possível realizar 2 rodadas do algoritmo. Com 10 recirculações do pacote através de

[2] https://github.com/regras/p4-forro

portas internas do próprio *switch*, é possível realizar as 20 rodadas do ChaCha20. Para controle do estado entre as recirculações, a implementação espera receber pacotes com um novo cabeçalho após o Ethernet, contendo 3 informações: modo de operação (cifração ou decifração), número da rodada e um índice (de 0 a 5) do conjunto de 512 bits que está sendo processado do *payload* (mensagem), conforme a Figura A.4.

Quando em modo de cifração, a implementação do ChaCha20 paralelo realiza a geração de um *nonce* no próprio Tofino, enquanto no modo decifração é utilizado um *nonce* informado junto à mensagem cifrada. Para simplificação dos testes de desempenho, todos os algoritmos usaram um *nonce* fixo para realizar a decifração de uma mensagem gerada aleatoriamente.

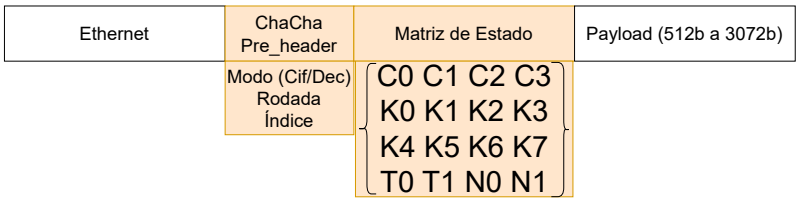

Figura A.4 – Estrutura de cabeçalhos do pacote de processamento do ChaCha20 paralelo em Plano de Dados Programável.

Em uma implementação direta, o algoritmo realiza as 12 operações básicas ao processar cada QR e assim o *pipeline* fica totalmente ocupado para o processamento, não sendo possível a inicialização e a finalização da matriz de estado, nem a cifração/decifração do conteúdo (o XOR do conjunto de 512 bits com a matriz). Entretanto, utilizando uma instrução especial da arquitetura Tofino chamada *Identity Hash*, é possível realizar uma operação de rotação de bits e uma operação de soma módulo 2^{32} ou XOR em um mesmo estágio com algumas adaptações no código P4. Com isso, é possível fazer a inicialização junto da primeira rodada e a finalização junto da última rodada, sendo necessária apenas uma recirculação a mais para se obter o cálculo do XOR com a matriz de estado. A Figura A.5 mostra as travessias do pacote para o cálculo da matriz de estado.

Essa mesma técnica foi utilizada para a implementação do algoritmo Forro14. No entanto, devido à natureza sequencial do algoritmo, não foi possível fazer o processamento de mais de um QR no mesmo *pipeline*, o que implica em quatro travessias (com uma recirculação entre cada duas travessias) para calcular os quatro QRs que equivalem a uma rodada do algoritmo. Com isso, são necessárias 28 recirculações para as 14 rodadas, o que resulta em 56 travessias para o cálculo da matriz de estado e uma extra para a cifração/decifração da mensagem. A quantidade de recirculações afeta diretamente a vazão que o algoritmo pode alcançar, visto que o mesmo pacote precisa passar várias vezes pelo *pipeline* e que as portas de recirculação também possuem uma fila para o acesso concorrente com as demais portas do *switch*. A implementação sequencial realiza a cifração de

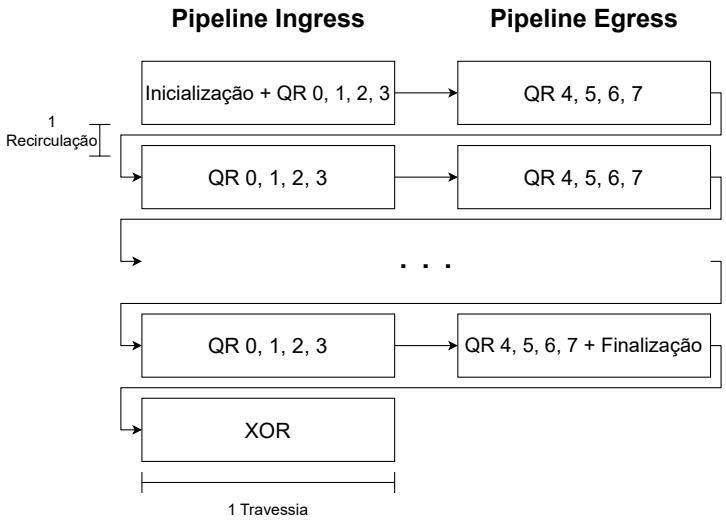

Figura A.5 – Algoritmo ChaCha20 paralelo em modo decifração em Plano de Dados Programável.

apenas 512 bits de dados, trabalhando com um campo de *payload* mínimo. A Figura A.6 representa o design do algoritmo Forro14 no plano de dados.

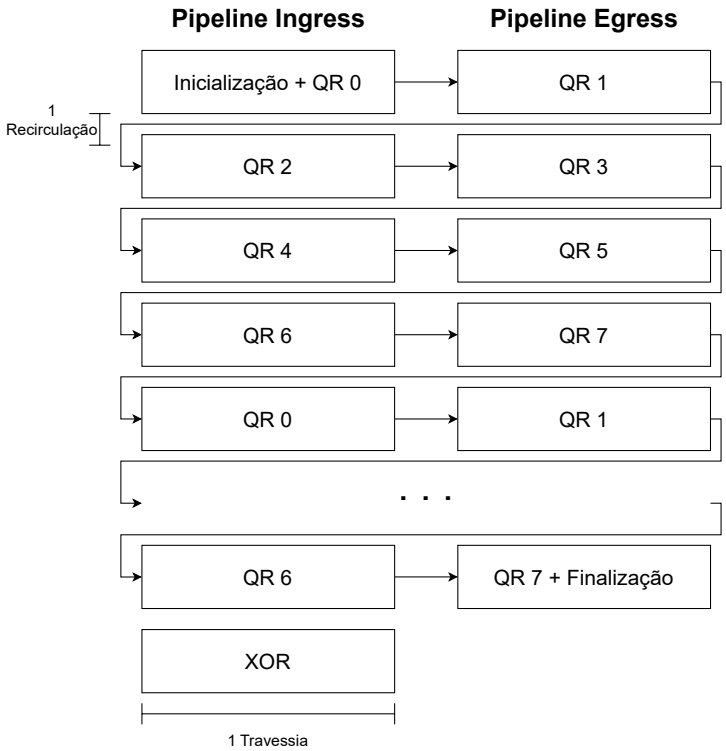

Figura A.6 – *Design* do Algoritmo Forro14 em Plano de Dados Programável.

Com o objetivo de processar apenas 512 bits, o cabeçalho do pacote para o algoritmo Forro14 foi reduzido em relação ao utilizado pelo ChaCha20 paralelo, conforme Figura A.7.

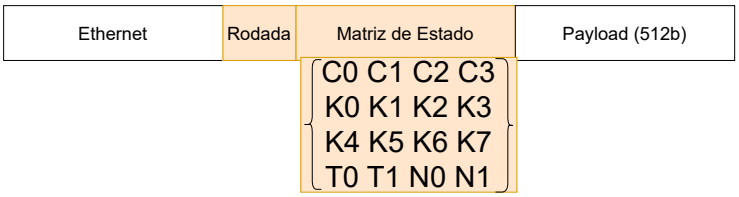

Figura A.7 – Estrutura de cabeçalho do pacote de processamento do Forro14 em Plano de Dados Programável.

Para uma comparação mais direta com o Forro14 quanto ao uso dos recursos do *switch*, foi feita uma implementação do algoritmo ChaCha20 sequencial, isto é, sem a paralelização dos QRs. Essa implementação foi baseada na implementação do Forro14, alterando apenas as operações realizadas nos QRs e a quantidade de recirculações feitas para cumprir as 20 rodadas do ChaCha20. Assim como ocorreu no Forro14, o cabeçalho dos pacotes também foi reduzido para o ChaCha20 sequencial. Nesse caso, foram necessárias 40 recirculações com 81 travessias do pipeline para realizar o cálculo da matriz de estados e a cifração.

Para verificar a implementação correta dos algoritmos, foram utilizados os vetores de teste disponíveis na RFC 7539 (NIR; LANGLEY, 2015) para o algoritmo ChaCha20 e no repositório de implementação do Forro14 (COUTINHO, 2023b).

A.7 Resultados e discussão

A.7.1 Configuração de testes

Para avaliação do desempenho do algoritmo em plano de dados programável, foi elaborado um cenário com dois *switches* programáveis Tofino: um dos *switches* (TG) está com um gerador de tráfego PIPO-TG (COSTA, 2023) para gerar tráfegos com cabeçalhos pré-definidos com até 100 Gbps de vazão, enquanto o outro *switch* (SC) está com o algoritmo de cifra de fluxo implementado. A Figura A.8 ilustra a configuração de testes.

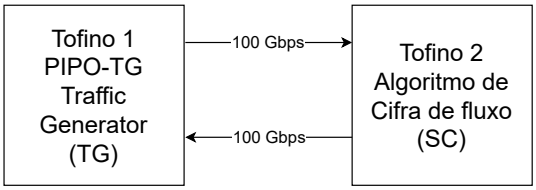

Figura A.8 – Configuração de testes para avaliação de desempenho.

O tráfego gerado é encaminhado para o *switch* SC (*Stream Cipher)* através de um cabo DAC (*Direct Attached Copper*) de 100 Gbps com conectores QSFP28 e devolvido para o *switch* TG (*Traffic Generator)* através de outro cabo igual. O enlace está

Tabela A.1 – Taxas de vazão dos algoritmos de cifra de fluxo

	Taxa de entrada											
Algoritmo	**1Gbps**	**2Gbps**	**3Gbps**	**4Gbps**	**5Gbps**	**6Gbps**	**7Gbps**	**8Gbps**	**9Gbps**	**10Gbps**	**11Gbps**	**12Gbps**
Forro14	1019	2038	3055	4076	5093	6110	7127	8152	9161	10186	9678	8701
ChaCha20 (Seq)	1019	2038	3055	4076	5093	6110	7127	6418	5581	4932	4310	3893
ChaCha20 (Par)	961	1922	2881	3844	4803	5763	6722	7689	8640	9607	10559	11523
	Taxa de entrada											
Algoritmo	**13Gbps**	**14Gbps**	**15Gbps**	**16Gbps**	**17Gbps**	**18Gbps**	**19Gbps**	**20Gbps**	**25Gbps**	**40Gbps**	**50Gbps**	**100Gbps**
Forro14	7904	7184	6520	5994	6083	5349	5388	4842	3594	1313	912	46
ChaCha20 (Seq)	3606	3145	2747	2233	2787	2336	2009	1860	1163	268	155	2,6
ChaCha20 (Par)	12477	13442	14410	13511	12901	12109	11565	11013	6971	2862	2789	476

configurado para funcionar a 100 Gbps sem FEC (*Forward Error Correction*) e com autonegociação de taxa desabilitada. Para recirculação, o SC possui 2 portas internas que trabalham em taxas de 100 Gbps. Essas duas portas foram utilizadas nos testes para reduzir a perda de pacotes devido à saturação de suas filas, sem ocupar portas externas do *switch*.

Os algoritmos estão com *nonce* e chave definidos de forma estática no código, portanto os valores aleatórios gerados pelo TG como *payload* são decifrados sempre utilizando a mesma chave e *nonce*. Deve ser observado que o *switch* não possui nenhuma estrutura de *cache* para acelerar a decifração com os mesmos parâmetros.

O tamanho do *payload* cifrado nos testes foi sempre de 512 bits. Portanto, todos os algoritmos trabalharam com a mesma quantidade de dados cifrados. Como é feita a decifração de apenas 512 bits, uma única matriz de estado é calculada para cada pacote, sendo os valores dos contadores da matriz sempre mantidos em 0.

A.7.2 Vazão de tráfego

Para coleta da vazão alcançada após a decifração, foram gerados tráfegos de pacotes de 1 a 20 Gbps e de outros valores comuns de velocidades de enlace de fibra óptica, como 25 Gbps, 40 Gbps, 50 Gbps e 100 Gbps. O tráfego é medido como o valor inteiro máximo da média móvel de vazão dos últimos 10 segundos na porta de retorno do SC para o TG, produzindo os resultados mostrados na Tabela A.1 e na Figura A.9.

Como demonstrado no gráfico, o algoritmo ChaCha20 paralelo manteve a taxa de transmissão até 15Gbps sem perdas significativas de vazão, enquanto o algoritmo Forro14 chegou até 10 Gbps e o ChaCha20 implementado de forma sequencial até 7 Gbps. Isso se deve principalmente à quantidade de recirculações necessárias para cada algoritmo, visto que mais recirculações implicam em mais pacotes sendo enfileirados nas filas de entrada das portas internas de recirculação.

Nesse ponto, é possível afirmar que o algoritmo ChaCha20 paralelo suporta taxas de tráfego maiores sem perda de pacotes para aplicações focadas no uso de todo

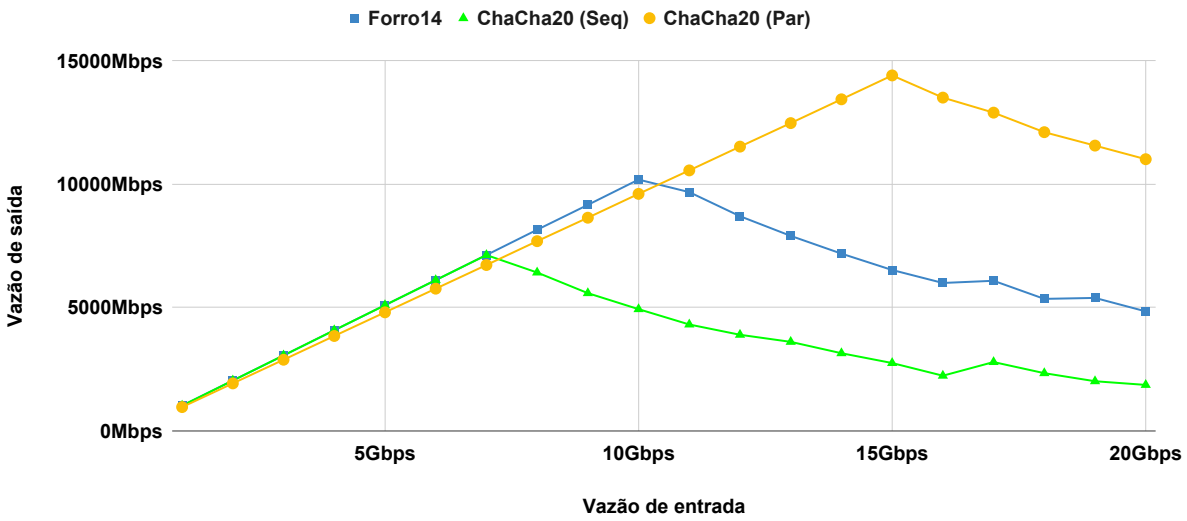

Figura A.9 – Vazão de saída dos algoritmos de cifra de fluxo no *switch* Tofino para diversas vazões de entrada.

o *switch* apenas para cifração/decifração do tráfego. Já o algoritmo Forro14 pode ser utilizado para aplicações que precisem de tráfego de até 10 Gbps sem perda de pacotes com o uso do *switch* apenas para cifração/decifração.

A.7.3 Ocupação de recursos

Outro ponto a ser verificado é a ocupação de recursos computacionais que cada algoritmo tem no SC. Para a maioria dos cenários de uso, não é interessante que os recursos do *switch* sejam destinados exclusivamente para a cifração de pacotes. Portanto, é relevante avaliar a ocupação de recursos pelos algoritmos.

Para coleta da ocupação de recursos, a Intel disponibiliza a ferramenta *P4 Insight*, que provê uma estimativa do uso dos recursos do *switch* de acordo com o resultado da compilação do código P4 para a ASIC. Diversos recursos compõem a arquitetura da ASIC e sua ocupação pode ser vista na Tabela A.2 e no gráfico da Figura A.10.

Como destaque da comparação, deve ser citado o *Action Data Bus Bytes* (ADBB) que representa a quantidade de operações que podem ser executadas nos estágios de um pipeline. Todos os algoritmos utilizaram pouco esse recurso, o que é positivo, pois, em *switches* que precisam lidar com muitos protocolos diferentes ou operações especializadas, esse recurso pode se esgotar rapidamente com mais operações concorrentes no *pipeline*.

Ternary Content Addressable Memory (TCAM), *Ternary Result Bus* (TRB) e *Ternary Match Input Crossbar* (TMIX) são recursos utilizados para *matches* ternários e de *longest prefix match* (LPM) em tabelas do *switch*, enquanto *Static RAM* (SRAM), *Exact Match Result Bus* (EMRB), *Exact Match Search Bus* (EMSB) e *Exact Match In-*

Tabela A.2 – Ocupação de recursos dos algoritmos de cifra de fluxo

Componente	ChaCha20 (Par)	Forro14	ChaCha20 (Seq)
ADBB	9,4%	3,6%	2,6%
EMRB	9,9%	12,5%	12,5%
EMSB	9,4%	12,5%	12,5%
GW	9,4%	0,0%	0,0%
HASHB	4,0%	5,4%	5,4%
HASHD	16,7%	2,8%	2,8%
LT-ID	12,5%	12,5%	12,5%
SRAM	1,4%	2,9%	2,7%
STASH	0,5%	12,5%	12,5%
TCAM	1,7%	0,0%	0,0%
TRB	9,4%	0,0%	0,0%
VLIW	4,7%	10,2%	10,2%
EMIX	2,8%	1,8%	1,8%
TMIX	2,1%	0,0%	0,0%
PHV	34,9%	43,1%	43,1%

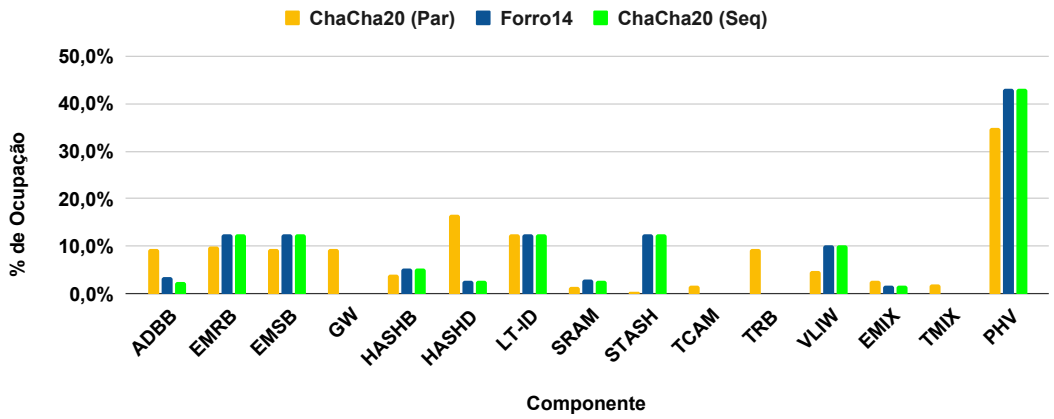

Figura A.10 – Ocupação de recursos do Tofino pelos algoritmos de cifra de fluxo.

put Crossbar (EMIX) são utilizados para *matches* exatos em tabelas. Como a memória TCAM é escassa e a SRAM é abundante em switches, as implementações do Forro14 e do ChaCha20 sequencial utilizaram apenas matches exatos. Portanto, não usam a memória TCAM, liberando este importante e escasso recurso para aplicações que necessitam de *match* ternário, como encaminhamento baseado em sub-rede, *multicast* e filtragem de pacotes.

Um ponto curioso é a ocupação dos *Packet Header Vectors* (PHV) do *switch* em cada caso. PHVs são memórias similares a registradores de CPU alocadas em cada estágio de processamento do pipeline para realizar operações aritméticas nos valores coletados dos cabeçalhos. Como o ChaCha20 paralelo realiza mais operações por estágio, espera-se que a alocação de PHVs seja maior que em algoritmos sequenciais que realizam apenas uma operação por estágio. O comportamento observado foi o oposto e uma possível explicação para essa alocação contraintuitiva de PHVs pode estar em eventuais otimizações feitas no

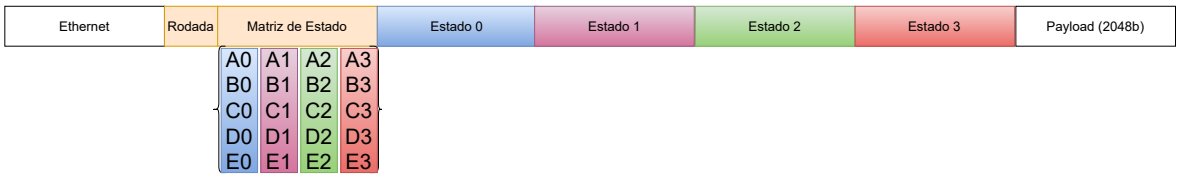

Figura A.11 – Esquema de cabeçalhos para cálculo de quatro estados do Forro14.

algoritmo ChaCha20 paralelo, mas não nos algoritmos sequenciais, tais como: (1) o uso de mais recursos de instrução de *Hash* (*Hash Bit* e *Hash Distribution*, devido ao maior uso da instrução *Hash Identity*) e (2) a forma de alocação dos dados da matriz de estado e do *payload* no momento da cifração/decifração pelo compilador, não utilizando o mesmo registrador para ambas as operações. Logo, é possível que consigamos, em implementações futuras, obter as mesmas alocações otimizadas que foram obtidas pela implementação do ChaCha20 paralelo, o que reforçaria a vantagem do Forro14 em relação ao ChaCha20 no tocante à alocação de escassos recursos do *switch*.

Sendo assim, vemos que a pergunta de pesquisa pode ser respondida da seguinte maneira: em comparação ao ChaCha20 paralelo, o algoritmo Forro14 é a melhor opção para aplicações que demandem até 10 Gbps sem perda de pacotes, visto que possui uma menor utilização de recursos do *switch*, liberando mais recursos para outras funcionalidades. Em cenários onde a ocupação de recursos do *switch* não é relevante, o algoritmo ChaCha20 paralelo apresenta um desempenho melhor em comparação ao Forro14. O ChaCha20 sequencial tem desempenho inferior ao Forro14 para taxas de entrada acima de 7 Gbps. Com relação ao consumo de recursos, vemos (pela Tab. A.2) que há um empate entre os dois, com ligeira vantagem para o ChaCha20 sequencial nos recursos ADBB e SRAM. Logo, torna-se mais interessante usar o Forro14 para aplicações cujas taxas de entrada estiverem entre 8 e 10 Gbps.

A.7.4 Paralelização do algoritmo Forro14

Na proposta do algoritmo Forro14 existe uma sugestão de uma forma de paralelização do algoritmo por meio do cálculo de mais de uma matriz de estado para a cifração de mais de um conjunto de 512 bits de dados por vez (COUTINHO, 2023a). Devido a algumas limitações da arquitetura do Tofino quanto à quantidade de dados que podem ser extraídos no *parser* do *pipeline Egress*, essa paralelização não pôde ser implementada ainda, necessitando investigações mais aprofundadas para tentar viabilizá-la. Uma abordagem possível é buscar uma alocação dos valores de estado de forma que sempre se calcule os mesmos elementos de matrizes diferentes em paralelo. A Figura A.11 exemplifica uma possível abordagem para essa implementação.

A.7.5 Impactos da quantidade de portas de recirculação

Em outra frente, investigou-se as taxas que poderiam ser alcançadas se mais portas de recirculações fossem utilizadas. O gráfico da Figura A.12 mostra uma relação de vazão alcançada utilizando-se 1, 2 e 4 portas de recirculação para o algoritmo Forro14. Como é possível perceber, a alocação de mais portas de recirculação não tem uma relação linear com a vazão máxima obtida após a cifração.

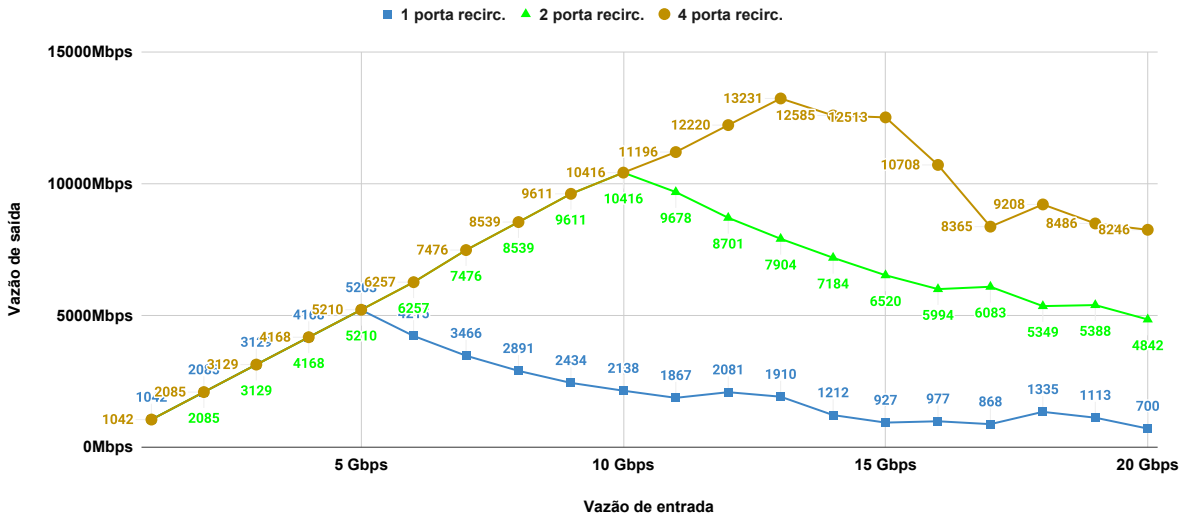

Figura A.12 – Vazão de saída do Forro14 em função da vazão de entrada e do número de portas de recirculação utilizadas.

Com isso, em uma aplicação na qual o *switch* esteja focado apenas na cifração de tráfego, o uso de até 32 portas de recirculação (2 internas mais 30 externas, para o *switch* utilizado) pode prover aumento expressivo na vazão.

Considerando uma vazão de entrada de 10 Gbps e que o *switch* Tofino trabalha a 100 Gbps, um pacote pode atravessar 10 vezes os *pipelines* antes da chegada do próximo. A partir do momento que o segundo pacote começou a ser processado, ele concorrerá com a travessia do primeiro, fazendo com que ambos atravessem os *pipelines* apenas 5 vezes até a chegada do próximo pacote. Entre as chegadas do terceiro e quarto pacotes, o primeiro pacote atravessará o pipeline 4 vezes, o segundo 3 vezes e o terceiro outras 3 vezes. À medida que mais pacotes vão chegando, essa competição pelos recursos aumenta, reduzindo o número de travessias de cada um durante os intervalos entre as chegadas de novos pacotes. Isso satura as filas de entrada das portas de recirculação, causando a queda da vazão de saída com taxas de entrada maiores.

A.8 Trabalhos futuros

Para entender melhor as relações entre os fluxos de entrada e saída discutidos acima, seria interessante buscar um modelo matemático que consiga caracterizar a relação entre tais fluxos e o número de portas de recirculação. Tal modelo deveria levar em consideração, entre outros pontos, que o número de recirculações de cada pacote é bem definido para cada algoritmo.

Alguns outros possíveis trabalhos futuros podem ser destacados como, por exemplo, a investigação de uma implementação paralela do Forro14 a partir do cálculo de mais de uma matriz de estado por *pipeline*, investigando-se os impactos em alocação de recursos e vazão. Outro trabalho seria a avaliação do uso de mais portas de recirculação de pacotes para maximizar a operação de cifração e decifração do tráfego pelo *switch*.

Uma outra investigação seria a implementação dos AEADs (*Authenticated Encryption with Associated Data*) XChaCha20-Poly1305 (ARCISZEWSKI, 2020) e XForro14-Poly1305 (COUTINHO *et al.*, 2023a) para verificar os impactos na vazão e na ocupação do *switch* causados pela inclusão de uma garantia de integridade e autenticidade do conteúdo cifrado, a qual é de grande interesse em alguns cenários de ataque.

Considerando o problema em aberto da alocação de PHVs nos algoritmos sequenciais, há ainda o trabalho de reestruturação do código de forma que uma mesma alocação de PHV seja utilizada para o cálculo da matriz de estado e para a operação de cifração e decifração.

A.9 Conclusão

Este trabalho implementou e avaliou o desempenho do algoritmo de cifra de fluxo Forro14 em plano de dados programável da arquitetura Intel Tofino.

A implementação foi comparada com duas versões do algoritmo ChaCha20 (paralela e sequencial).

Apesar de não conseguir alcançar as mesmas vazões de dados que a versão paralela do ChaCha20, o Forro14 usa menos recursos do *switch* e é a melhor opção para taxas de até 10Gbps. Além disso, ele supera a versão sequencial do ChaCha20 em termos de capacidade de vazão, utilizando aproximadamente os mesmos recursos que este.

Como o Forro14 é mais econômico em termos de demandas por recursos e oferece uma taxa de vazão satisfatória para vários casos de uso, ele se mostra uma opção interessante em aplicações que necessitam da implementação de outras funções no mesmo *switch*, como roteamento e filtragem de pacotes, por exemplo.

A.10 Agradecimentos

Agradecemos ao Fabricio Rodríguez e ao Francisco Vogt do Information and Networking Technologies Research and Innovation Group (INTRIG) do DCA da UNICAMP pelas contribuições e discussões relevantes para a implementação dos algoritmos Forro14 e ChaCha20 sequencial no hardware Intel Tofino.

Este trabalho foi parcialmente financiado pela Fundação de Amparo à Pesquisa do Estado de São Paulo (FAPESP), processo 2021/00199-8, CPE SMARTNESS.