Implementação e avaliação da cifra de fluxo Xote em plano de dados de hardware programável Tofino

Rodrigo A. de A. Pierini[1], **Caio Teixeira**[1],
Christian Esteve Rothenberg[1], **Marco Amaral Henriques**[1]

[1] Faculdade de Engenharia Elétrica e de Computação
Universidade Estadual de Campinas
Campinas - SP - Brasil

{rpierini,chesteve,maah}@unicamp.br

caio@dca.fee.unicamp.br

Abstract. *This work introduces a novel element reordering technique applied to the Intel Tofino architecture's packet interpreter, enabling the first-time implementation of the Xote algorithm on programmable data planes, and presents a comprehensive comparative performance analysis of the ChaCha, Forró, and Xote algorithms on this architecture. The results demonstrate that the ChaCha algorithm, when implemented with the proposed reordering technique, achieves higher maximum packet throughput and similar resource utilization compared to the other algorithms, surpassing previously reported results in the literature with a 1.65x increase in maximum throughput.*

Resumo. *Este trabalho introduz uma técnica de reordenação de elementos aplicada no interpretador de pacotes da arquitetura Intel Tofino, viabilizando a implementação inédita do algoritmo Xote em plano de dados programáveis, e apresenta uma análise comparativa abrangente do desempenho dos algoritmos ChaCha, Forró e Xote nesta arquitetura. Os resultados demonstram que o algoritmo ChaCha, quando implementado com a técnica de reordenação proposta, alcança uma maior vazão máxima de pacotes e uma utilização de recursos similar em comparação com os demais algoritmos, superando os resultados previamente reportados na literatura com um aumento da vazão máxima em 1,65 vezes.*

1. Introdução

As tecnologias de redes de computadores passaram por diversas mudanças de paradigmas nos últimos anos, avançando das redes tradicionais que possuem foco no encaminhamento de pacotes e engenharia de tráfego, às redes programáveis com foco na distribuição de processamento das aplicações através de todo o núcleo da rede. Essas mudanças de paradigmas trazem novas oportunidades e desafios para diversas áreas, como a área de segurança da informação.

Garantir a segurança de uma rede de computadores é uma necessidade crescente em cenários com ataques cibernéticos cada vez mais elaborados. Portanto, busca-se prezar por sigilo, integridade e disponibilidade dos dados que estão em trânsito ou sendo

processados pela rede. Com o processamento de pacotes ocorrendo independente de protocolos, novas tecnologias podem ser agilmente criadas para vencer os atuais desafios de segurança em redes, viabilizando novas abordagens para proteção de dados em trânsito.

Pesquisas desenvolvidas no contexto de redes programáveis abordam formas de se prover sigilo às comunicações através da implementação e avaliação de algoritmos criptográficos diretamente no *chip* de processamento de pacotes em *switches* programáveis. Algoritmos de criptografia como AES [Dworkin et al. 2001], ChaCha [Bernstein et al. 2008] e Forró [Coutinho et al. 2023] já foram implementados e avaliados quanto à sua vazão e uso de recursos na plataforma "Intel Tofino" com o objetivo de prover sigilo em comunicações e verificar a integridade de dispositivos em uma rede gerenciada. Como demonstrado na literatura, algoritmos de cifra de fluxo implementados em plano de dados programável podem alcançar vazões de dados maiores com menor uso de recursos quando comparadas à cifra de bloco AES [Yoshinaka et al. 2022] implementada na mesma arquitetura devido a não utilizarem tabelas de substituição (*S-Boxes*) e processarem a entrada e a chave conjuntamente, sem o uso de uma *key schedule*.

Este trabalho apresenta a implementação do algoritmo de cifra de fluxo "Xote" [Coutinho et al. 2023] e conduz uma comparação com outros algoritmos de criptografia em termos de vazão máxima e uso de recursos. Portanto, uma pergunta de pesquisa que este artigo permite responder é: "Com o objetivo de prover sigilo de dados que trafegam e são processados no núcleo de uma rede programável, como os algoritmos de cifra de fluxo 'Xote', 'Forró' e 'ChaCha' implementados na plataforma Tofino se comparam em vazão máxima e uso de recursos?"

Três contribuições relevantes são apresentadas ao longo deste trabalho: (i) uma nova técnica de reordenação de elementos de matriz no interpretador de pacotes (parser) para otimizar uso de recursos; (ii) a primeira implementação do algoritmo Xote em plano de dados programável na arquitetura Intel Tofino; (iii) uma avaliação detalhada de desempenho dos algoritmos Chacha, Forró e Xote em três níveis diferentes de segurança.

Para melhor contextualizar este trabalho, na Seção 2 são apresentados alguns paradigmas de redes que surgiram nos últimos anos. Já a Seção 3 apresenta as cifras de fluxo de interesse exploradas na literatura e sua implementação em plano de dados programável. A Seção 4 apresenta algumas restrições para implementação de algoritmos na plataforma Tofino. As implementações existentes na literatura são introduzidas na Seção 5 onde descrevemos a técnica desenvolvida para viabilizar a implementação da cifra de fluxo "Xote" nessa plataforma. A seguir, a Seção 7 apresenta e discute os dados coletados nos experimentos conduzidos com as implementações, com o objetivo de comparar seu desempenho em vazão e uso de recursos. Por fim, o trabalho é concluído na Seção 8 com a apresentação de possíveis trabalhos futuros.

2. Paradigmas de redes de computadores

Nas redes de computadores, os dados são transportados entre hospedeiros através do núcleo de rede na forma de pacotes de dados. O objetivo da rede é realizar a transmissão desses pacotes com a menor latência e perda possível. Para isso, os equipamentos de núcleo de rede são divididos em duas partes: o plano de controle (*control plane*, CP), responsável por definir como encaminhar pacotes, e o plano de dados (*data plane*, DP), responsável por realizar o encaminhamento dos pacotes. O DP é constituído por um

ASIC (*Application-Specific Integrated Circuit*) que interpreta, modifica e encaminha os pacotes a partir de parâmetros providos por um software executado no CP.

Em redes de computadores tradicionais, cada equipamento de comutação de pacotes (seja um *switch* ou um roteador) possui o CP e DP embarcados, realizando uma operação encapsulada e colaborativa com outros equipamentos.

Já por volta de 2011, um novo paradigma de redes chamado "redes definidas por software" (*software-defined networking*, SDN) surgiu com o objetivo de facilitar o gerenciamento de redes através da centralização do CP em um controlador com visão de todos os dispositivos da rede. nesse paradigma, o DP nos dispositivos é configurado por softwares em execução no controlador utilizando uma API, permitindo a rede operar de forma integrada [Kreutz et al. 2014].

No paradigma SDN, os DPs são limitados a operarem de acordo com a arquitetura do ASIC feita pelo fabricante do equipamento e seguindo protocolos de rede existentes. Portanto, por volta de 2014, um novo paradigma chamado de "redes programáveis" (*Programmable Networks*) surgiu com o objetivo de viabilizar a definição da operação do ASIC de forma programável, permitindo que desenvolvedores de planos de dados programáveis (*programmable data plane*, PDP) definam a forma que o DP deve interpretar, modificar e encaminhar pacotes sem depender de protocolos predefinidos. Esta definição de operação do PDP é feita através de uma linguagem de domínio específico chamada P4 (*Programming Protocol-independent Packet Processors*) [Kfoury et al. 2021] [Hauser et al. 2023].

Redes programáveis viabilizam que o núcleo da rede realize o processamento de dados diretamente no DP. Com isso, uma nova tendência chamada "computação em rede" (*In-Network Computing, INC*) adota o descarregamento (*offloading*) do processamento de dados tradicionalmente realizado nos hospedeiros para equipamentos de rede programáveis, liberando recursos dos processadores em hospedeiros para realizarem outras operações.

Essa abordagem também permite que o processamento de dados seja realizado durante a transmissão de dados na rede [Gherari et al. 2023]. Portanto, o paradigma de redes programáveis também viabiliza o processamento de funções de segurança de dados na rede, permitindo abordagens inovadoras para a proteção de redes e dados. Dentre estas abordagens, destaca-se a aplicação de criptografia diretamente no PDP com algoritmos de cifração como AES [Chen 2020], ChaCha [Yoshinaka et al. 2022] e Forró [Pierini et al. 2024], bem como algoritmos de autenticação de mensagem baseados em resumo criptográfico (*hash-based message authentication code*, HMAC) como Half SipHash [Yoo and Chen 2021] e Chaskey [Francisco et al. 2024].

Nesse contexto, esse artigo busca expandir o estado da arte com a avaliação de desempenho de uma nova implementação de um algoritmo de cifra de fluxo em PDP. A próxima seção apresenta conceitos fundamentais dos algoritmos de cifra de fluxo "ChaCha", "Forró" e "Xote".

3. Cifras de fluxo

Algoritmos de cifra de fluxo são utilizados para prover sigilo a dados através de uma operação de ou-exclusivo bit-a-bit (*bitwise XOR*) com um conjunto de bits (*keys-*

tream) gerado a partir de uma função pseudoaleatória (*Pseudorandom Function*, PRF) utilizando uma chave secreta compartilhada. Para a segurança desse tipo de cifra, o conjunto de bits utilizado para cifrar os dados não pode ser repetido, pois essa repetição facilita a recuperação da chave utilizada.

Para evitar essa repetição, os algoritmos ChaCha e Forró utilizam um *nonce* (número de uso único) de 64 bits que varia a cada mensagem cifrada e um contador de 64 bits incrementado a cada conjunto de 512 bits gerado, alterando os parâmetros de entrada da PRF do algoritmo. Juntamente com uma constante de 128 bits e uma chave secreta de 256 bits, esses parâmetros são estruturados em uma matriz (chamada "matriz de estado") com 16 elementos de 32 bits organizados em 4 linhas e 4 colunas.

Estes algoritmos são chamados de cifras ARX (*Add, Rotate, XOR*), pois os elementos dessa matriz são processados em rodadas utilizando operações de adição módulo 2^{32}, ou-exclusivo e rotação circular de bits. Quanto maior o número de rodadas, mais complexa é a reversão das operações para recuperar a chave utilizada, chegando à segurança de 2^{256} tentativas, isto é, força-bruta.

O algoritmo ChaCha é uma variação do algoritmo Salsa20, vencedor da competição eStream do NIST (*National Institute of Standards and Technology*) para padronizar um algoritmo de cifra de fluxo [Bernstein 2008]. Atualmente, não há um ataque de recuperação de chave para 8 rodadas de processamento desses algoritmos que seja mais eficiente que força-bruta.

Portanto, recomenda-se que a PRF utilizada no algoritmo ChaCha realize pelo menos 8 rodadas para geração do *keystream*. A quantidade de rodadas é indicada ao final do nome do algoritmo, sendo uma versão do ChaCha com 8 rodadas chamado de ChaCha8 e do Salsa20 de Salsa20/8. Assim como o Salsa20, são padronizadas as versões ChaCha8, ChaCha12 e ChaCha20. [Bernstein et al. 2008]

A matriz de 16 elementos utilizada como entrada da PRF do algoritmo ChaCha é processada de forma alternada a cada rodada entre colunas e diagonais da matriz. Na primeira rodada, os elementos das colunas da matriz são inseridos em uma função chamada "*Quarter Round Function*" (QR), onde o elemento de cada linha representa os *buffers* A, B, C e D, respectivamente. Esta função realiza 12 operações ARX nos elementos extraídos de cada coluna da matriz, conforme ilustrado na Figura 1.

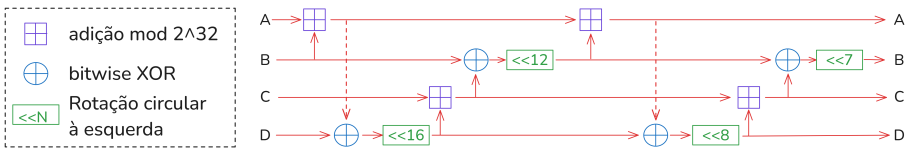

Figura 1. Ilustração da *Quarter Round Function* da cifra de fluxo ChaCha

Visto que as colunas representam elementos independentes, os QRs são executados de forma paralelizada. Uma rodada é concluída após aplicar a função QR em cada coluna. Na próxima rodada, as mesmas funções QR são aplicadas nas diagonais da matriz. Os QRs das diagonais também são calculadas de forma paralelizada.

Ao fim do total de rodadas, os elementos da matriz resultante são somados com os elementos da matriz inicial, produzindo um conjunto de bits pseudoaleatórios para

cifração dos dados. O mesmo processo é utilizado para a decifração, de forma que o mesmo conjunto de bits pseudoaleatórios sejam gerados. A Figura 2 ilustra esse processo para os algoritmos ChaCha e Forró.

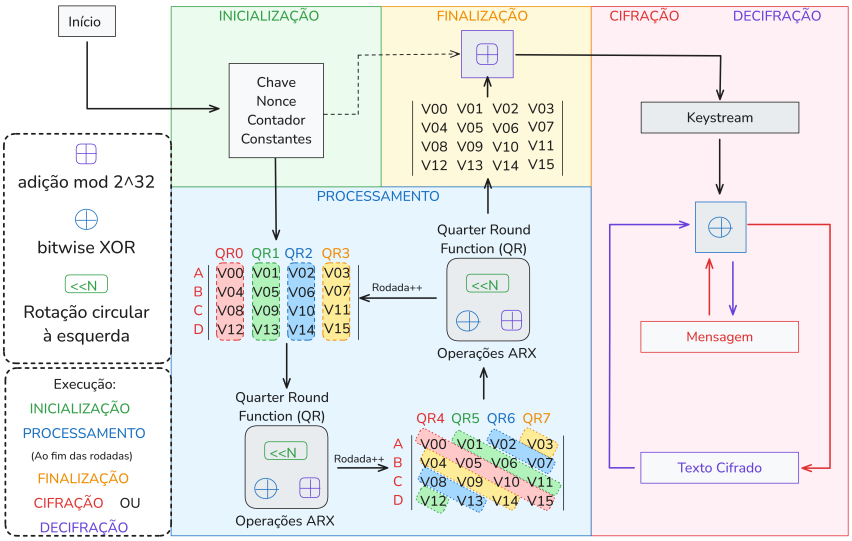

Figura 2. Ilustração dos algoritmos de cifra de fluxo ChaCha e Forró

É importante destacar que esses algoritmos não garantem a autenticidade de origem, visto que um atacante pode obter o mesmo *keystream* realizando uma operação XOR entre mensagem e texto cifrado, e então gerar um novo texto cifrado válido. Para garantir a autenticidade de origem, um código de autenticação de mensagem (MAC) deve ser utilizado junto ao algoritmo de cifra de fluxo, como o algoritmo "Poly-1305" também proposto por Bernstein [Bernstein 2005].

O algoritmo Forró, uma variação do algoritmo ChaCha, foi proposto por Coutinho de forma a torná-lo mais resistente a ataques de criptoanálise diferencial. Este tipo de ataque busca correlações entre os bits dos *keystreams* gerados a partir de cada chave utilizada, reduzindo a busca de prováveis chaves utilizadas [Coutinho 2023]. O algoritmo Forró alcança o mesmo nível de segurança a esses ataques utilizando menos rodadas que o algoritmo ChaCha, tendo uma segurança de 2^{256} a partir de 6 rodadas. Com isso, são recomendadas as versões Forró6, Forró10 e Forró14.

O algoritmo Forró é mais resistente a ataques de criptoanálise diferencial com menos rodadas devido a uma técnica chamada "polinização". Nessa técnica, o QR atual recebe um parâmetro adicional (E) que é obtido do QR anterior (o *buffer* A), tornando a execução de cada QR dependente do resultado do QR anterior. Embora isso aumente a difusão das modificações através da matriz e dificulte ataques de análise diferencial, essa técnica também torna a execução do algoritmo sequencial, visto que não é possível executar um QR sem o resultado do anterior. a Figura 3 ilustra o QR do algoritmo Forró.

Para reduzir o impacto no desempenho do algoritmo em decorrência da polinização, Coutinho propôs uma variação do algoritmo chamada "Xote". Nessa variação, dois *keystreams* são calculados paralelamente, de forma a cifrar conjuntos de 1 Kib ao invés de apenas 512 bits [Coutinho et al. 2023].

A próxima seção aborda a implementação destes algoritmos em plano de dados

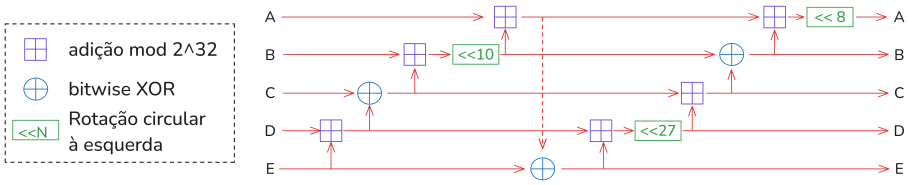

Figura 3. Ilustração da *Quarter Round Function* da cifra de fluxo Forró

programável baseado na plataforma Tofino para processamento de pacotes.

4. Restrições de implementação na plataforma Tofino

A linguagem P4, desenvolvida para programação de processadores de pacotes, possui diferenças em comparação a linguagens de programação desenvolvidas para processadores de propósito geral, trazendo novos desafios e novas técnicas para realizar o processamento de dados em PDP. Para implementar essas cifras de fluxo em plano de dados programável, algumas restrições da plataforma devem ser consideradas. A plataforma Intel Tofino utilizada define uma arquitetura de processador de pacotes chamada "*Tofino Native Architecture*" (TNA), ilustrada na Figura 4.

Essa arquitetura conta com 2 *pipelines* de processamento de pacotes (chamados "*Ingress*" e "*Egress*"), onde é realizada a interpretação (*parser*), modificação e reconstrução (*deparser*) dos cabeçalhos dos pacotes para processamento. Cada *pipeline* possui 12 estágios de processamento que utilizam uma estrutura de *match/action tables*, onde buscas em tabelas (*matches*) decidem a operação (*action*) realizada nos cabeçalhos interpretados. Nessa estrutura, os registros das tabelas são definidos pelo CP.

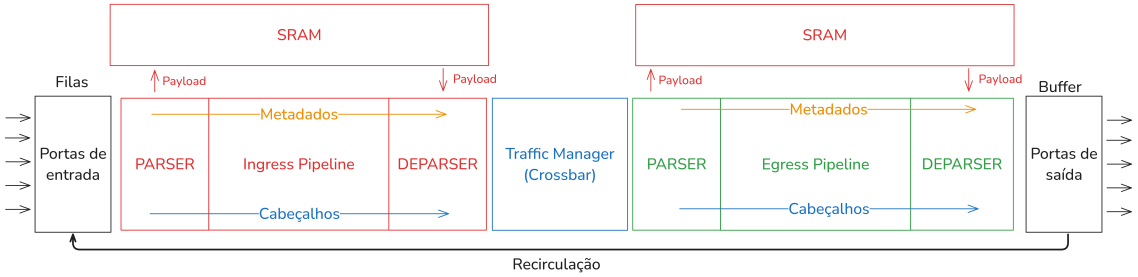

Figura 4. ilustração da arquitetura TNA utilizada em *switches* Intel Tofino

A linguagem P4, utilizada para programar o PDP da arquitetura TNA, não possui suporte a laços de repetição, devendo os dados serem reinseridos no *switch* através da recirculação do pacote para continuar seu processamento. No entanto, essa recirculação reduz a vazão máxima alcançável pelo pacote. Para recirculação de pacotes sem necessidade de uso das portas externas, a arquitetura TNA disponibiliza 2 portas internas no *switch* para recirculação.

Para realizar o *parser* dos cabeçalhos, a arquitetura utiliza uma máquina de estados finitos (*finite state machine*, FSM) que decide o próximo cabeçalho a ser interpretado a partir de valores dos cabeçalhos já coletados. O *parser* do *Ingress pipeline* pode extrair até 4 Kib, enquanto o *parser* do *Egress pipeline* pode extrair apenas 1280 bits, sendo que 192 bits são utilizados em ambos *pipelines* para extrair metadados da arquitetura.

Os dados que são interpretados no *parser* são inseridos no circuito de processamento do PDP em *Packet Header Vectors* (PHVs): registradores de 32, 16 e 8 bits presentes em cada estágio de processamento do *pipeline*. PHVs possuem restrições de alocação de acordo com as possíveis operações realizadas com eles.

Por exemplo, dados podem ser particionados entre PHVs menores para alocação (por exemplo, um dado de 32 bits pode ser dividido em 2 PHVs de 16 bits) e dados que não são operados na mesma travessia de um *pipeline* podem ser alocados no mesmo PHV. No entanto, dados que serão rotacionados ou gravados com o resultado de operações aritméticas não podem ser particionados entre PHVs, como as operações ARX.

5. Trabalhos relacionados: algoritmos de cifra de fluxo em plano de dados programável

Para todas as implementações citadas nessa seção, um cabeçalho de controle é incluído após o cabeçalho *ethernet* com seu conteúdo variando de acordo com a implementação específica, mas com ao menos um contador de rodadas processadas do algoritmo de cifra de fluxo. O conteúdo a ser cifrado é adicionado após o cabeçalho de controle, devendo o comprimento do *payload* ser definido em tempo de compilação pelo desenvolvedor.

O algoritmo ChaCha foi implementado em plano de dados programável (PDP) por Yoshinaka *et al.*. Nessa implementação, o *switch* realiza a cifração ou decifração dos dados enviados em um quadro *ethernet* sem *ethertype* específico. Para a cifração, o *switch* gera um *nonce* aleatório e o adiciona após o cabeçalho de controle. Já para a decifração, o *nonce* deve ser informado pelo emissor do pacote. O *switch* decide através de uma *flag* no cabeçalho de controle se o *nonce* deve ser gerado ou extraído [Yoshinaka et al. 2022].

Essa implementação exige que a chave de cifração e a quantidade de rodadas sejam definidas estaticamente no código P4 compilado para o *switch*, não sendo possível sua alteração durante a operação nem a definição de uma chave para cada porta ou origem. Além disso, o comprimento de 512 a 3072 bits (em intervalos de 512 bits) de dados a serem cifrados pelo *switch* também é definido em tempo de compilação, devendo inserir os dados no *payload* do pacote com o *padding* realizado.

O algoritmo Forró14 foi implementado por Pierini et al. [Pierini et al. 2024]. Esta implementação é focada na cifração de 512 bits e é aplicável para o contexto de atestação remota de dispositivos em redes de *data centers*. Portanto, não são utilizados outros tamanhos de conteúdo para cifração.

A implementação requer que a chave, a quantidade de rodadas e o *nonce* sejam definidos estaticamente no código. Portanto, não é feita a geração do *nonce* para cifração, mas isso torna a implementação insegura para um cenário real.

6. Implementação do algoritmo Xote em PDP

Nesse trabalho é apresentada a implementação do algoritmo "Xote" em PDP[1]. Para essa implementação, a chave é inserida em tempo de execução no *switch*, podendo ser definida uma chave por endereço MAC de origem. O *nonce* a ser utilizado deve ser

[1]https://github.com/RPierini/p4-forro-regras/tree/xote

informado entre o cabeçalho *ethernet* e o conteúdo a ser cifrado/decifrado, ficando a cargo do emissor a geração do *nonce* utilizado. O comprimento dos dados a serem cifrados é fixado em 1 Kib, devendo ser inseridos pelo emissor com o *padding* realizado.

Para a implementação, o algoritmo foi dividido em 3 etapas: inicialização, processamento e finalização. Na inicialização, as matrizes de estado iniciais referentes ao primeiro e segundo conjunto de 512 bits são carregadas com os parâmetros obtidos a partir de uma tabela indexada pelo endereço MAC de origem.

No processamento, um QR é executado em cada matriz a cada travessia de pipeline. Considerando que cada rodada (**r**) do algoritmo é composto de 4 QRs, são necessárias **4r** travessias para realizar o processamento de todas as rodadas, sendo o pacote recirculado a cada execução de um QR par (contado de 0 a 7) e ao final do processamento das rodadas. Na finalização, cada matriz é somada elemento a elemento com seus respectivos valores iniciais.

Após essa soma, o primeiro conjunto de 512 bits do *payload* é cifrado utilizando a operação de XOR com o resultado obtido. Devido à limitação de alocação de PHVs na arquitetura, a cifração do segundo conjunto de 512 bits precisa ser realizada no *Ingress pipeline*, implicando em outra recirculação de pacote.

Com as recirculações para finalização e cifrações, são necessárias **4r+3** travessias e **2r+2** recirculações de pacotes para toda execução do algoritmo, o que impacta a vazão máxima obtida. A Figura 5 ilustra a disposição do processamento do algoritmo nos *pipelines* da arquitetura TNA.

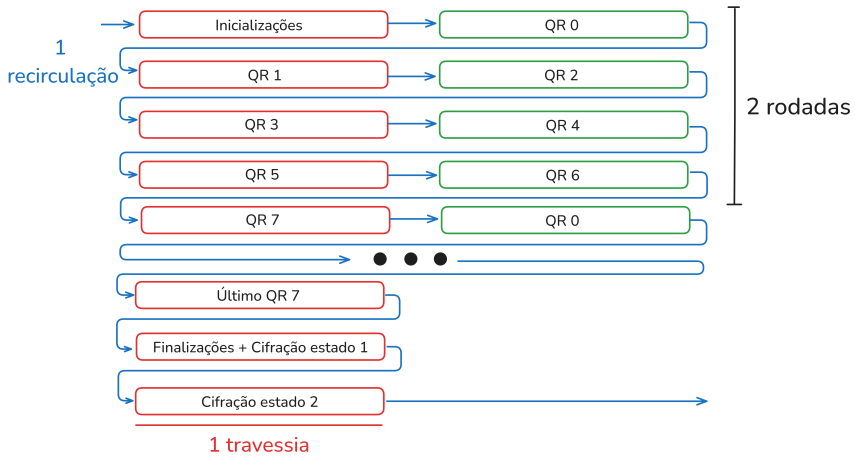

Figura 5. Disposição do processamento do algoritmo Xote na arquitetura TNA

Não é possível fazer a finalização das matrizes e a primeira cifração do *payload* no *Egress pipeline* devido à limitação de extração de 1280 bits no *Egress parser*. Além disso, considerando os tamanhos de cabeçalhos:

- 192 bits em metadados da arquitetura TNA;
- 112 bits do cabeçalho *Ethernet*;
- 8 bits do cabeçalho de controle; e
- 64 bits do *nonce* escolhido pelo emissor do pacote;

totalizando 376 bits, restam apenas 904 bits no *Egress parser* para realizar a extração de cabeçalhos, impedindo que ambas matrizes de 512 bits sejam totalmente interpretadas

devido ao deficit de 120 bits para extração. Isso impede que a última linha da segunda matriz seja processada no *Egress pipeline*, inviabilizando o processamento de qualquer QR nesse *pipeline*.

Para lidar com essa limitação, foi desenvolvida uma técnica de reordenação dos elementos das matrizes de estado, de forma que o *parser* sempre realize a interpretação dos parâmetros do QR atual na primeira linha da matriz e os elementos do QR anterior na segunda linha. Esta abordagem traz complexidade à estruturação do *parser*, mas viabiliza a implementação do algoritmo e otimiza o uso de PHVs, visto que sempre são processadas as mesmas posições de elementos da matriz interpretada. Para clarificar, chamaremos a primeira matriz de estado de matriz "**M**" e a segunda matriz de estado de matriz "**N**".

Primeiro, as matrizes são transpostas para que as colunas processadas nos QRs 0 a 3 se tornem linhas. As novas linhas serão nomeadas de **M0** a **M3** e **N0** a **N3**. Para possibilitar a extração de 3 linhas de ambas matrizes no *Egress parser*, as matrizes são mescladas linha-a-linha. Por fim, as linhas são reordenadas de forma que a primeira linha e o primeiro elemento da segunda linha representem os parâmetros utilizados como entrada no QR0. A Figura 6 ilustra essa organização das matrizes para iniciar o processamento dos QRs.

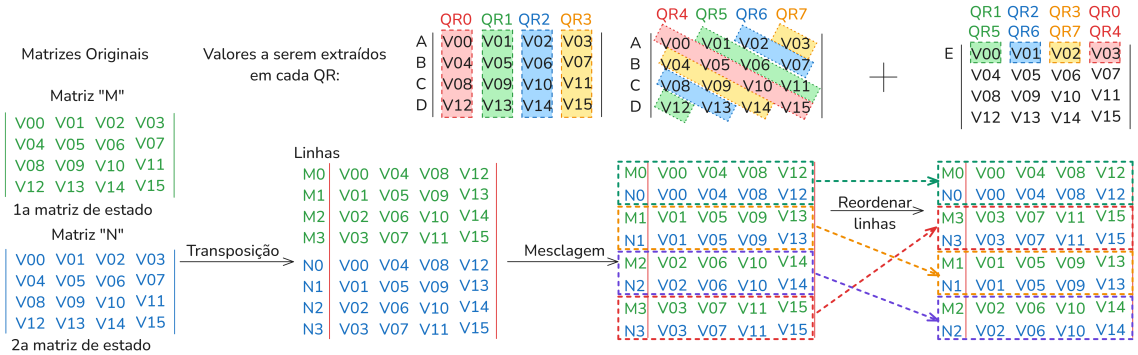

Figura 6. Organização inicial das matrizes de estado para implementação do algoritmo Xote em P4.

Ao fim de cada QR, as linhas são reorganizadas para que as mesmas posições de elementos sejam extraídos no *parser* da próxima travessia. No caso do *Egress pipeline*, a última linha de cada matriz não é extraída, sendo copiadas na mesma posição para o próximo QR e as demais linhas **rotacionadas**. A exemplo, ao fim do QR0 executado no *Egress pipeline*, as linhas **M02** e **N02**, são mantidas no fim da matriz, enquanto as linhas **M0**, **N0**, **M3** e **N3** são movidas para o meio da matriz de forma que as linhas **M1** e **N1** a serem processadas no QR1 fiquem no início da matriz.

No caso dos QRs 1 e 5 no *Ingress pipeline*, **as linhas** são extraídas e **reordenadas** para serem preparadas para a próxima travessia no *Egress pipeline*. Já no caso dos QRs 3 e 7, **os elementos** são extraídos individualmente para serem inteiramente **reordenados**, de forma que as linhas representem as diagonais (no caso do QR7) ou colunas (no caso do QR3) das matrizes originais. Para clarificar, as diagonais das matrizes originais transpostas em linhas são nomeadas **M4** a **M7** e **N4** a **N7**. A Figura 7 ilustra as reordenações realizadas em cada QR para processamento das duas primeiras linhas extraídas de cada matriz em cada travessia.

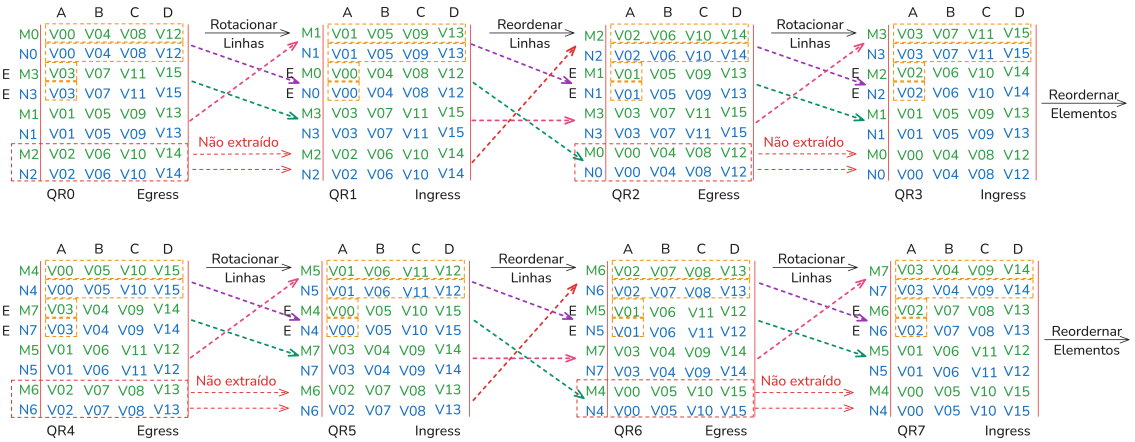

Figura 7. Esquema de reordenação de elementos de matrizes para processamento dos QRs no algoritmo Xote em P4.

Ao final do QR7 da última rodada, os cabeçalhos são reconstruídos com o formato esperado para o QR0. Ao recircular o pacote para a finalização e cifração, os elementos das matrizes resultantes são interpretados na ordem esperada pelo QR0 e somados com os elementos correspondentes das matrizes iniciais para produzir 1 Kib de *keystream*.

Para a cifração com *keystream*, uma técnica de rotação de conjuntos de 512 bits do *payload* proposta no trabalho de Yoshinaka et al. é adaptada, onde os conjuntos de 512 bits são rotacionados no *parser* para fazer a cifração dos conjuntos de dados a cada conjunto de *keystream* calculado [Yoshinaka et al. 2022]. A nossa implementação do algoritmo Xote realiza apenas a cifração de 1 Kib, não sendo possível aumentar a quantidade de dados cifrados devido a limitações do *deparser* do *Ingress pipeline*.

Esta seção descreveu as implementações de cifra de fluxo em PDP com destaque para a implementação do algoritmo Xote. Na próxima seção é descrita a análise de vazão máxima e uso de recursos para cada um dos algoritmos de cifra de fluxo elencados, bem como os impactos da técnica desenvolvida em comparação à implementação do algoritmo ChaCha disponível na literatura.

7. Avaliação experimental

Para verificar o desempenho dos algoritmos de cifra de fluxo implementados em PDP em condições semelhantes, os algoritmos Forró e ChaCha foram implementados utilizando a mesma técnica de rotação de matriz de estado, mas utilizando apenas uma matriz por vez. Essas implementações fazem a cifração de 1Kib calculando 2 matrizes de estado de forma sequencial. Não foi possível aplicar a mesma técnica de cálculo de duas matrizes de estados juntas para o algoritmo ChaCha devido a limitações de PHVs no *pipeline Ingress*.

Com o objetivo de medir a vazão alcançada pelos algoritmos em função da taxa de tráfego recebida para cifração, utilizou-se outro *switch* programável executando o gerador de tráfego PIPO-TG [Costa 2023] com tráfegos gerados na faixa de 1 Gbps a 25 Gbps, em intervalos de 1 Gbps, composto por pacotes com cabeçalho *ethernet*, 64 bits do *nonce* inicial e 1024 bits de *payload* aleatório.

Para os experimentos, foram utilizadas as 2 portas internas de recirculação de pacotes disponíveis na arquitetura TNA para execução dos algoritmos. Tanto a taxa de entrada quanto a vazão de saída representam o tamanho do pacote composto por seus cabeçalhos e *payload*, não representando apenas o tamanho da mensagem de entrada e o texto cifrado de saída.

O número de portas de recirculação possui um impacto na vazão máxima alcançada, pois o limitante para a vazão máxima é o tamanho das filas de pacotes nas portas de recirculação utilizadas [Pierini et al. 2024]. Os resultados coletados podem ser visualizados na Figura 8.

Como pode ser observado, as versões do algoritmo ChaCha obtiveram uma vazão máxima maior do que as versões correlatas dos algoritmos Forró e Xote, indicando que o algoritmo ChaCha é mais indicado para casos de uso que requerem até 19Gbps de vazão com um nível mais baixo de segurança, ou 10Gbps com requisitos de segurança mais restritivos.

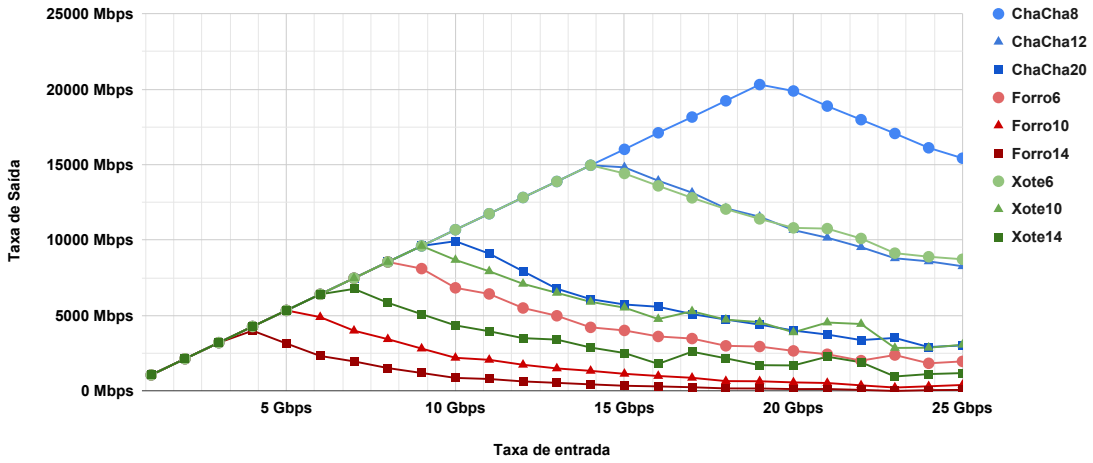

Figura 8. Vazão de saída dos algoritmos de cifra de fluxo no *switch* TNA em função da vazão de entrada para cifração de 1024 bits.

Quanto à técnica de reorganização de elementos das matrizes no *parser* desenvolvida para implementação do algoritmo Xote, ela permitiu que o algoritmo Xote se compare à implementação do algoritmo ChaCha realizada por Yoshinaka et al. quanto à vazão máxima alcançada para a cifração de 1024 bits de *payload*.

No entanto, ao aplicar a mesma técnica para implementação do algoritmo ChaCha, foi observado um aumento da vazão máxima obtida em comparação à implementação de Yoshinaka et al. em 1,65 vezes para a execução com 20 rodadas. A Figura 9 ilustra as vazões máximas obtidas por algoritmo, onde o sufixo "-R" indica a implementação com a técnica de reorganização de elementos.

Para a comparação de uso de recursos, extraiu-se relatórios de ocupação dos recursos da arquitetura TNA através da ferramenta *P4 Insight*. A arquitetura TNA utiliza diversos componentes para operação, sendo 15 deles utilizados pelas implementações comparadas. A Figura 10 mostra o consumo de recursos para cada implementação.

Quanto à utilização de recursos, a quantidade de rodadas não influencia a

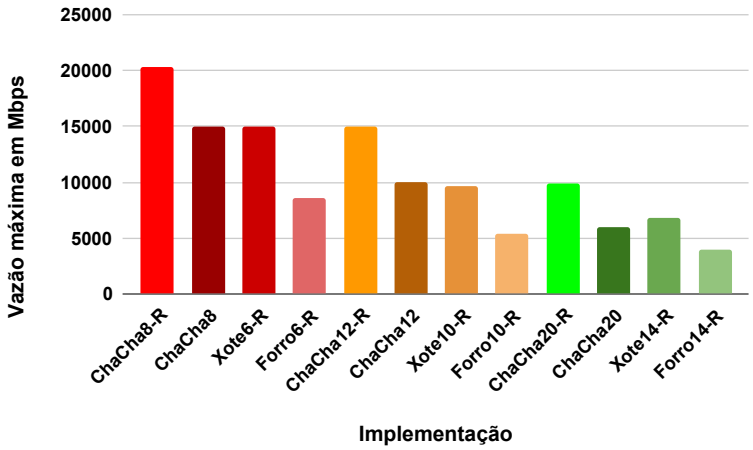

Figura 9. Comparação de vazão máxima obtida entre os algoritmos implementados e a implementação do algoritmo ChaCha por Yoshinaka et al.

utilização, visto que esse parâmetro é definido nas tabelas do *switch* e a alocação de recursos para as tabelas é feita pelo máximo de travessias definida em código. Os valores extraídos referem-se a um *switch* realizando apenas a cifração de tráfego. No entanto, outras funções de encaminhamento e processamento de pacotes podem ser adicionadas ao *switch* com pouco impacto nos recursos utilizados, pois pode-se utilizar os mesmos recursos alocados para finalidades diferentes a depender do cabeçalho do pacote.

Como pode ser observado, a implementação do algoritmo ChaCha utiliza menos PHVs do que do que a implementação do algoritmo Xote e menos VLIW do que as implementações do Forró e Xote. Os usos de outros recursos são relativamente próximos, não indicando uma vantagem significante quando considerada a diferença na vazão máxima obtida por cada algoritmo.

Quanto às implementações do algoritmo ChaCha, a implementação com reordenação de elementos possui um maior uso de PHVs em comparação à

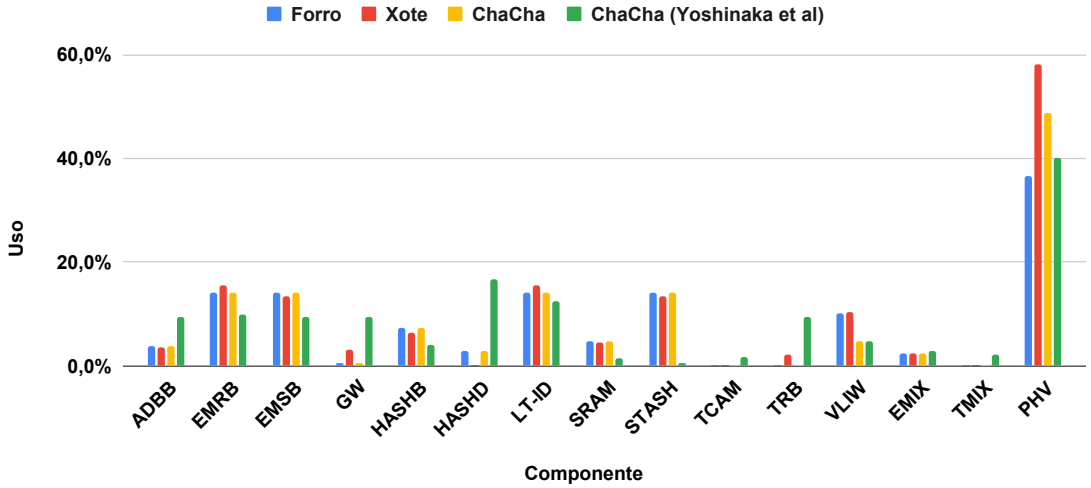

Figura 10. Utilização de recursos da arquitetura TNA por cada implementação de cifra de fluxo para cifração de 1024 bits.

implementação de Yoshinaka et al., mas utiliza menos recursos HASHD, ADBB e GW, influenciados pelo fluxo de execução do *pipeline* e o fluxo de dados entre PHVs.

Além disso, a implementação com reodernação utiliza menos recursos referentes à memória TCAM (TCAM, TRB e TMIX), escassa no *switch*, ao utilizar mais recursos referentes à memória SRAM e *matches* exatos (EMRB, EMSB, SRAM e EMIX), abundantes no *switch*. A opção pelo maior uso de SRAM é interessante para *pipelines* que incluem funções de encaminhamento de pacotes baseado em sub-rede e filtragem de pacotes baseado em ACLs, que requerem a memória TCAM para serem implementados.

8. Conclusão e trabalhos futuros

Essa primeira implementação do Xote foi viabilizada pela adoção da nova técnica de reordenação de elementos no switch Tofino, a qual teve impacto nos algoritmos avaliados. Constatou-se que ela permitiu que a implementação do Xote alcançasse taxas maiores que a implementação do Forró para cifração de dados de 1024 bits. No entanto, a implementação do Xote utiliza mais recursos e obtém taxas menores em comparação ao ChaCha implementado com a mesma técnica de reordenação. Além disso, a implementação do ChaCha com a técnica de reordenação de elementos permitiu alcançar uma vazão máxima 1,65x maior em comparação à implementação do mesmo algoritmo disponível na literatura.

Como trabalho futuro, cita-se a avaliação de algoritmos de cifra de fluxo padronizados com *nonces* de 96 bits (XChaCha e XForró) e possivelmente 128 bits, este último realizando a substituição do contador de conjuntos de 512 bits incrementado por pacote por um contador de todos os conjuntos de 512 bits cifrados pelo *switch*, similar ao *Packet Numbering* utilizado no protocolo MACSec.

Outro possível trabalho futuro é investigar o impacto de acrescentar algoritmos de código de autenticação de mensagem (MAC), que possam ser implementados em plano de dados programáveis, de forma a prover autenticação de origem para os pacotes cifrados. Para este fim, poderiam ser comparados os algoritmos Poly-1305, comumente utilizado em conjunto com implementações do algoritmo ChaCha, e os algoritmos Half SipHash e Chaskey, já implementados em P4.

Tendo em vista que a quantidade de portas do *switch* alocadas para recirculação impacta a vazão máxima obtida sem perda de pacotes, um possível trabalho futuro é a investigação de quantas portas de recirculação são necessárias para se alcançar vazões máximas condizentes com taxas de transferência padronizados em enlaces *ethernet*.

Por fim, a investigação desses algoritmos poderia ser expandida para outros planos de dados programáveis, como SmartNICs e eBPF, para avaliar a melhor abordagem para prover sigilo em comunicações utilizando os novos paradigmas de redes.

Agradecimentos

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

Referências

Bernstein, D. J. (2005). The poly1305-aes message-authentication code. In *International workshop on fast software encryption*, pages 32–49. Springer.

Bernstein, D. J. (2008). *The Salsa20 Family of Stream Ciphers*, page 84–97. Springer-Verlag, Berlin, Heidelberg.

Bernstein, D. J. et al. (2008). Chacha, a variant of salsa20. In *Workshop record of SASC*, volume 8, pages 3–5. Citeseer.

Chen, X. (2020). Implementing aes encryption on programmable switches via scrambled lookup tables. In *Proceedings of the Workshop on Secure Programmable Network Infrastructure*, pages 8–14.

Costa, F. G. (2023). Pipo-tg: parameterizable high performance traffic generation.

Coutinho, M. (2023). Design, diffusion, and cryptanalysis of symmetric primitive.

Coutinho, M., Passos, I., Vásquez, J. C. G., Sarkar, S., de Mendonça, F. L., de Sousa Jr, R. T., and Borges, F. (2023). Latin dances reloaded: Improved cryptanalysis against salsa and chacha, and the proposal of forró. *Journal of Cryptology*, 36(3):18.

Dworkin, M., Barker, E., Nechvatal, J., Foti, J., Bassham, L., Roback, E., and Dray, J. (2001). Advanced encryption standard (aes).

Francisco, M., Ferreira, B., Ramos, F. M. V., Marin, E., and Signorello, S. (2024). P4chaskey: An efficient mac algorithm for pisa switches. In *7th European P4 Workshop (EuroP4'24)*, pages –.

Gherari, M., Akbari, F. A., Habibi, S., Ali, S. O., Hmitti, Z. A., Kardjadja, Y., Saqib, M., Maia, A. M., Rayani, M., Soyak, E. G., Elbiaze, H., Ercetin, O., Ghamri-Doudane, Y., Glitho, R., and Ajib, W. (2023). A review of the in-network computing and its role in the edge-cloud continuum.

Hauser, F., Häberle, M., Merling, D., Lindner, S., Gurevich, V., Zeiger, F., Frank, R., and Menth, M. (2023). A survey on data plane programming with p4: Fundamentals, advances, and applied research. *Journal of Network and Computer Applications*, 212:103561.

Kfoury, E. F., Crichigno, J., and Bou-Harb, E. (2021). An exhaustive survey on p4 programmable data plane switches: Taxonomy, applications, challenges, and future trends. *IEEE Access*, 9:87094–87155.

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

Pierini, R., Teixeira, C., Rothenberg, C., and Henriques, M. (2024). Implementação e avaliação da cifra de fluxo forro14 em hardware programável tofino usando a linguagem p4. In *Anais do XXIV Simpósio Brasileiro de Segurança da Informação e de Sistemas Computacionais*, pages 399–414, Porto Alegre, RS, Brasil. SBC.

Yoo, S. and Chen, X. (2021). Secure keyed hashing on programmable switches. In *Proceedings of the ACM SIGCOMM 2021 Workshop on Secure Programmable network INfrastructure*, pages 16–22.

Yoshinaka, Y., Takemasa, J., Koizumi, Y., and Hasegawa, T. (2022). On implementing chacha on a programmable switch. In *Proceedings of the 5th International Workshop on P4 in Europe*, pages 15–18.