Journal of the Brazilian Computer Society, 2026, 32:1, doi: 10.5753/jbcs.2026.5625

Implementation and evaluation of the Forro stream cipher in Tofino programmable hardware for remote attestation in datacenters

Rodrigo Alexander de Andrade Pierini [**Universidade Estadual de Campinas** | ***rpierini@unicamp.br***]
Caio Teixeira [**Universidade Estadual de Campinas** | ***caio@dca.fee.unicamp.br***]
Christian Rodolfo Esteve Rothenberg [**Universidade Estadual de Campinas** | ***chesteve@unicamp.br***]
Marco Aurélio Amaral Henriques [**Universidade Estadual de Campinas** | ***maah@unicamp.br***]

School of Electrical and Computer Engineering, Universidade Estadual de Campinas, Av. Albert Einstein, 400 Cidade Universitária, CEP 13083-852, Campinas – SP, Brazil

Received: 01 March 2025 • **Accepted:** 28 June 2025 • **Published:** 24 February 2026

Abstract. The software-defined networking (SDN) paradigm has enabled several innovations in computer networking, specially in programmable packet processing. This paper shows the feasibility and impact on computing resources of the Forro stream cipher algorithm in the Tofino programmable hardware switch. For comparison purposes, the ChaCha algorithm was also analyzed in terms of its performance and impact on the same device. It was observed that the Forro algorithm performs better and uses fewer resources than ChaCha in sequential implementations. However, when parallelization techniques are adopted, ChaCha performs better for higher data rates, but uses more ternary matching resources than Forro. For the use case of remote attestation in programmable data planes, the Forro cipher seems more promising, as it uses less limited resources and can achieve sufficient throughput rates for this scenario. We then propose P4DRA, a distributed remote attestation solution based in the programmable data plane that can offload the verification process of remote devices to the data plane, freeing resources from a central verifier based on a x86 server and improving the attestation proof verification speed by around 150 times.

Keywords: software-defined networking, stream ciphers, cryptography, programmable data planes, remote attestation, Forro stream cipher, Tofino, P4 language

1 Introduction

In the last decades, computer networks have been through significant paradigm shifts in scenarios where network management and operation are financially more relevant and need higher performance, such as Metropolitan Area Networks (MAN) and data centers. One of the major changes is the separation between the circuit for packet forwarding (data plane) and the intelligence that defines how to forward packets (Control plane), defining the computer network paradigm called Software-defined Networking (SDN) [Kreutz *et al.*, 2014]. SDN enables centralizing the control of a network and using control software to define innovative ways of forwarding packets, which are made possible through the programmability of the data plane with technologies like OpenFlow [Fernandes and Rothenberg, 2014] or, more recently, P4 [Bosshart *et al.*, 2014].

P4 is a domain-specific language that allows high-level definition of a data plane, bringing programmability to several different devices (*targets*) with versatility. The P4 language is widely used for developing in-network solutions that allow for high performance packet processing and forwarding in programmable data planes.

SDN-enabling technologies bring several new possibilities of innovation, not only in traffic engineering, packet processing or resource optimization, but also in network security. New ways of providing security guarantees with these technologies have been explored in several research areas, such as intrusion detection [Mahrach *et al.*, 2018], firewall [Datta *et al.*, 2018], DDoS attack detection [Sivaraman *et al.*, 2017], IP Spoofing [Li *et al.*, 2019] and even hash [Yoo and Chen, 2021] and cryptographic [Chen, 2020] operations directly in the data plane. However, implementing such functions impacts performance and resource usage in programmable switches; therefore, an active topic is the investigation of more efficient solutions to provide those security guarantees with low impact in resources and performance.

In such context, one of this work's objectives is the implementation and evaluation of a new stream cipher called Forro [Coutinho *et al.*, 2023b] in SDN, investigating its feasibility in terms of network throughput and resource usage when compared to other confidentiality solutions. To the best of our knowledge, this is the first initiative to implement this stream cipher in programmable switches based on the Tofino hardware using the P4 language.

Running these algorithms directly in packet forwarding devices can reduce server resource usage, allow the distribution of the processing throughout the network and take advantage of the physical and topological location of devices for data processing separation. Such sectorization can be interesting in certain scenarios, as in the remote attestation use case.

Remote attestation enables the verification of operational conformity of managed devices in the network through a remote protocol. This technique is specially interesting in scenarios with several managed devices, like data centers and IoT. This approach requires a management device to request and verify integrity proofs provided by managed devices, centralizing the verification processing cost into one single device and reducing its scalability. Therefore, offloading the verification task to programmable network devices could improve scalability while reducing the resource usage at the management device.

This introduction brings us to the following research questions for this work: *In the context of SDN networks based on the P4 language, is Forro stream cipher a better alternative than ChaCha stream cipher for remote attestation, considering resource usage and network throughput?* And *how to enable remote attestation proof processing in the data plane to free verifier resources?*

To evaluate Forro, we implemented the algorithm on a Tofino programmable switch using P4 and evaluated its performance using controlled network traffic generation and resource usage monitoring on the switch. This evaluation was then compared with the implementation of the well-known ChaCha stream cipher, which was the base for the proposal of Forro.

In Section 2, we provide some background in programmable data planes using the P4 language. Following up, Section 3 provides background about remote attestation, and states the main contributions of this work. Section 4 then provides some background in stream ciphers, as required for our investigation and proposal.

Starting at Section 5, we present a first glance at our proposal of an approach for remote attestation in programmable data planes and introduce the limitations of enabling it in this environment. Section 6 discusses the implementation of Forro in Tofino, then we evaluate and compare its performance to a ChaCha implementation in Section 7.

In Section 8 we present the P4DRA protocol and implementation for remote attestation in the programmable data plane. Finally, in Section 9, we conclude our work and present some possible future works.

2 P4 programmable data plane

A programmable data plane (PDP) is a technology that can be referred to as a specialization of SDN. In the first works with SDN, the approach was based on the OpenFlow Protocol [Kreutz *et al.*, 2014]. This protocol allows network switch manufacturers to design their devices' forwarding chips in a way that some predefined features in protocols could be programmed by a central controller.

Even though it allows several switch features to be controlled in a centralized way, the processing of packets is still limited to the switch's forwarding chip definitions, which are built upon predefined protocols in the networking context, such as Ethernet, IP, and others.

To accelerate innovation and free the operator from the limitations imposed by switch manufacturers, the concept of PDP has emerged. In this technology, the manufacturer designs a forwarding chip that can be programmed like an FPGA (Field Programmable Gate Array) and process packets at the device's defined line rate. The operator is then responsible for defining the protocols and operations that the switch can perform before its deployment. The programming model adopted for this approach is the use of *Match-Action tables*, defining the switch as a table lookup and data manipulation device on predefined structures, directing these manipulations according to the results of the lookup.

The operator has to use a domain-specific language called P4 (Programming Protocol-independent Packet Processors) [Bosshart *et al.*, 2014] to program the forwarding chip. The high-level code defined by the operator and an architecture definition supplied by the switch manufacturer are used as inputs for a compiler that generates code compatible with the chip, usually in JSON (JavaScript Object Notation) or bytecode.

Figure 1 shows a reference architecture called PISA (Protocol-Independent Switch Architecture) that can be programmed using the P4 language. The PISA architecture composes a packet processing as a device with two processing pipelines, called *Ingress* and *Egress* pipelines. In addition to these pipelines, Static RAM (SRAM) stores the packet payload and the packet processing configuration and the *Traffic Manager* performs the internal packet switching in the device. In each pipeline, there is a *Parser* that analyze the packet headers data and stores it in structures called *Headers*. These *Headers* are processed by the pipeline together with the packet *Metadata* that contains extra information, like packet ingress port, ingress timestamp, defined output port, among others. The portion of the packet header that is not parsed is saved as payload and is not processed by the pipelines.

Finally, in the *Deparser*, the packet is rebuilt using the extracted and processed headers together with the payload and sent to the next component of the architecture.

The P4 language has become the standard for programming packet processors and is adopted in several types of data planes, such as programmable switches [Intel, 2024], SmartNICs [Scholz *et al.*, 2019] and even in-kernel data planes like eBPF [Vieira *et al.*, 2020].

With the use of programmable data planes, several more complex applications can be implemented inside the data forwarding infrastructure, enabling a new paradigm called *In-Network Computing* (INC) [Tokusashi *et al.*, 2018], in which several computing operations can be performed directly in the data plane, such as distributed systems consensus algorithms [Dang *et al.*, 2020], in-network caching [Jin *et al.*, 2017], machine learning [Zheng *et al.*, 2023], and many others related to network performance, monitoring, telemetry and application support [Kfoury *et al.*, 2021; Hauser *et al.*, 2023].

However, the P4 language has several limitations that make it challenging to implement algorithms not directly related to packet forwarding. A common limitation is the fact that the P4 language does not have support for iterative statements (loops). A new iteration for processing data is done by reintroducing a packet containing partially processed data back into the packet queues (namely, packet recirculation), so that the same processing steps can be re-executed. Therefore, implementation of ciphers like AES and ChaCha is more

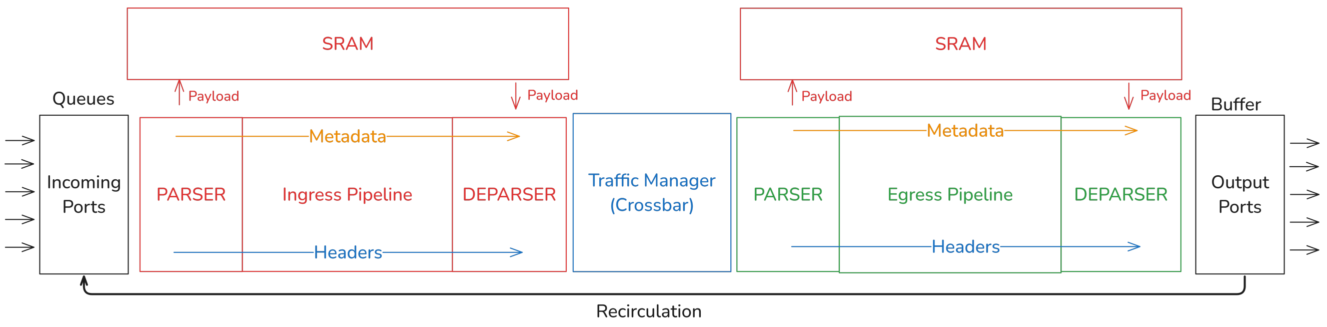

Figure 1. Reference architecture of a P4-programmable switch. Source: adapted from Systems Approach. [Peterson *et al.*, 2021].

complex, as they rely on such statements throughout their execution, and as such, requires architecture-dependent techniques where the operation will be performed, which are defined in the compilation of the P4 code.

3 Remote attestation in data planes

Remote attestation is a technique where a *Verifier* requests from a remote entity (called *Prover*) a proof of integrity of its current state, and then compares it to an expected, well-known correct state. This proof of integrity can be calculated as a *hash* of certain files, parameters and configurations of the device.

A common approach to attest a remote device is requesting the result of the *measured boot* process, where the boot process of a device is measured through a cumulative hash of the components loaded and executed in each step [Ling *et al.*, 2021].

To identify if the measured boot of a Prover is in a correct state, the Verifier stores the expected result of a known correct configuration, to be later compared with the requested proof.

For the proof generation, the Prover can collect the measured boot result hash stored in a secure memory (such as a *Trusted Platform Module*, TPM [TCG, 2024]), sign it and then send both hash and signature to the Verifier. The Verifier then verifies if the signature is valid and compares the hash to the expected result it possesses. If the hash from the proof and the hash of the expected value match, then the remote device is considered to be in a valid state. Otherwise, the remote device is considered to be in an unexpected state and actions may be taken by external processes or personnel.

Figure 2 illustrates the process of a measured boot. In this process, each component of the boot process stores the hash of the next boot component. The current component verifies if the next component has the same hash as the expected value. Before loading the next component, another hash, that starts with the hash of the first component, is stored in the TPM. This hash is then extended (by performing the hash operation on the current value concatenated with the previous hash of the current component) during the load of each component of the boot process, and results in a value that represents the final state of the whole chain. This value can be requested by the operating system as a proof of the currently loaded boot configuration to be compared with a well known correct value by the remote Verifier, attesting the current state's correctness.

In a remote attestation scenario, the goal of an adversary is to perform modifications in the Prover state without being noticed by the Verifier. This can be achieved by undoing any modifications before the attestation process begins or by forging a valid state proof and answering it to the Verifier. Since a central Verifier can be overwhelmed by a large amount of proof verifications in a larger scenario, the attestation request for each device is performed either on-demand or with a large time window between requests, enabling an adversary to compromise the Prover between attestation rounds. This kind of attack between attestations is called *Time-of-check-to-time-of-use* (TOCTTOU) attack and is illustrated in Figure 3 [Tan *et al.*, 2011].

In this illustration, the attestation is requested to the device in regular intervals. When a attacker compromises the device, it changes its operational context (i.e. through configuration hijacking) and keeps a copy of the valid operational context. When the attackers notices that a attestation will be requested, it changes the operational context back to the correct state and hides his presence in the system (as illustrated by the ghost in the picture). Once the attestation process is finished, the attacker changes the context again.

To prevent this kind of attack, the generated proof must be authentic, relevant and current. To achieve these protections, the proof must be signed or generated by the prover using a secret key (source authenticity), generated using relevant information about the prover state (integrity) and be based in a probabilistic value that prevents proof re-use (like a random nonce) and proof requests forgery by an adversary to store the expected proofs in advance.

The main drawback of remote attestation is its scalability, since the central Verifier needs to verify each signature and calculate the expected configuration for every nonce sent to a device. A common approach to scale remote attestation to hundreds of thousand devices (in scenarios like IoT) is the use of Collective Remote Attestation (CRA) [Ambrosin *et al.*, 2020]. In CRA, the state of all or a cluster a devices is verified collectively, informing the Verifier if all devices are in their expected states or if at least one device is not. This approach prevents the Verifier of having a quick granular view of the state of its managed devices, since it will need to trace the invalid device in case of an invalid attestation.

In this regard, this work also proposes a distributed and granular remote attestation scheme that allows the verification of servers integrity in the context of data centers without burdening a central Verifier based in a x86 server, in a way that programmable switches in the network are capable of ver-

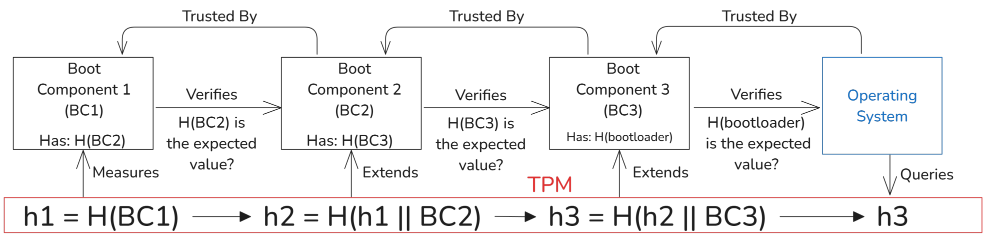

Figure 2. The measured boot process

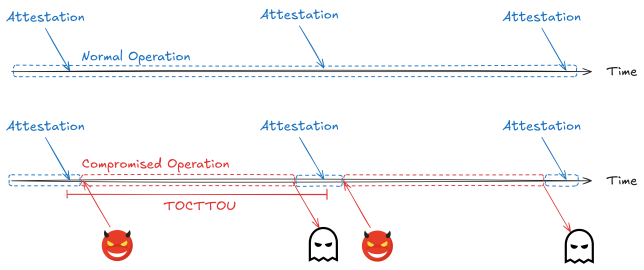

Figure 3. Illustration of a TOCTTOU attack scenario

ifying the integrity of devices connected to them. This would require the switch to be capable of calculating hashes and digital signatures without significant impact in its forwarding operation capabilities.

As described in the previous section, the P4 language has several limitations that makes it challenging to implement hash and digital signature functions directly in the data plane of a programmable switch. Therefore, it is necessary to investigate other approaches to make it feasible to verify a proof directly in a data plane. One possible approach is to use some type of *Pseudorandom Function* (PRF) to generate a value that can be authenticated through a shared key and be dependent on a relevant input information: the configuration hash. This is where stream ciphers can be an interesting approach, as described in Section 4.

4 Stream ciphers

Aiming to provide confidentiality in a communication, cryptographic algorithms can be used to ensure that only holders of a cryptographic key can have access to the original content of a encrypted message, making it computationally unfeasible to decipher a ciphertext without the knowledge of the key.

Among the existing symmetric cryptographic algorithms, two major groups can be outlined: *block cipher algorithms*, in which data blocks of fixed size are encrypted, and *stream cipher algorithms*, in which data is encrypted bit-by-bit by a pseudorandom bitstream generated from a secret key.

In the context of block ciphers used in programmable networks, the implementation of AES [Dworkin *et al.*, 2001] stands out, with satisfactory performance for several applications. The algorithm was implemented in programmable networks using a technique called *scrambled lookup tables*, where basic processing stages are previously calculated and stored in lookup tables, simplifying encryption and decryption [Chen, 2020].

In the context of stream ciphers, a commonly used algorithm is the ChaCha20 stream cipher [Bernstein *et al.*, 2008]. This algorithm uses a key and a nonce to generate a 4x4 state matrix of 32 bits elements, totalizing 512 bits of state, which is then used encrypting (or decrypting) a message (or a ciphertext) through a bitwise XOR operation. It is required that each calculated state matrix is unique for the algorithm to be secure, preventing the same state matrix to be used to encrypt two sets of 512 bits from a message. Counters and nonces are used to prevent the generation of the same state matrix, with the nonce varying by message and the counter varying for every set of 512 bits of the message.

The ChaCha20 algorithm uses 20 rounds of operations divided in functions called QR (Quarter Round), where 4 QRs compose one round of the algorithm. QRs performs operations of addition modulo 2^{32}, XORs and circular bit shifts (a set of operations known as ARX, *Add-Rotate-Xor*) to the matrix, causing diffusion of modifications in the bits through all the state matrix. The QRs are applied in the columns of the matrix in odd rounds and in the diagonals in even rounds (counting from round 1 up to round 20). Figure 4 illustrates the operations on each QR and how the 32 bits elements of the state matrix are collected as parameters for each QR. The elements C_i, K_i, T_i and N_i of the state matrix are, respectively, the *constants*, *key*, *counter* and *nonce*. The final state matrix after 20 rounds is added element-by-element to the initial state matrix to produce the serialized *keystream*, which is then XORed with the set of 512 bits of the message for encryption or decryption. For the next set of 512 bits of the message, a new state matrix must be calculated, but with an increment in the counter, generating different keystreams for every 512-bit message chunk.

In the SDN context, ChaCha was implemented in programmable switches with the parallelization of the QRs, being capable to encrypt messages ranging from 512 bits to 3072 bits, with throughput 50 to 100% higher in comparison to the implementation of AES with a 256-bit key [Yoshinaka *et al.*, 2022].

Forro has a very similar structure to ChaCha, but with the addition of a technique called “pollination”, which increases the diffusion of modifications in the calculated state matrix. This increases the algorithm resistance to differential analysis attacks. However, this technique makes algorithm necessarily sequential, since pollination creates a dependency between each QR by using an element modified in the previous QR. This limits the speed of round calculations by preventing the utilization of low-level parallelization available in some devices. Figure 5 shows Forro14 QR operations and how the elements of the state matrix are collected as parameters in each QR. A reorganization of the elements C_i, K_i, T_i and N_i in the initial state matrix of Forro can be observed in comparison to the elements in the initial state matrix of

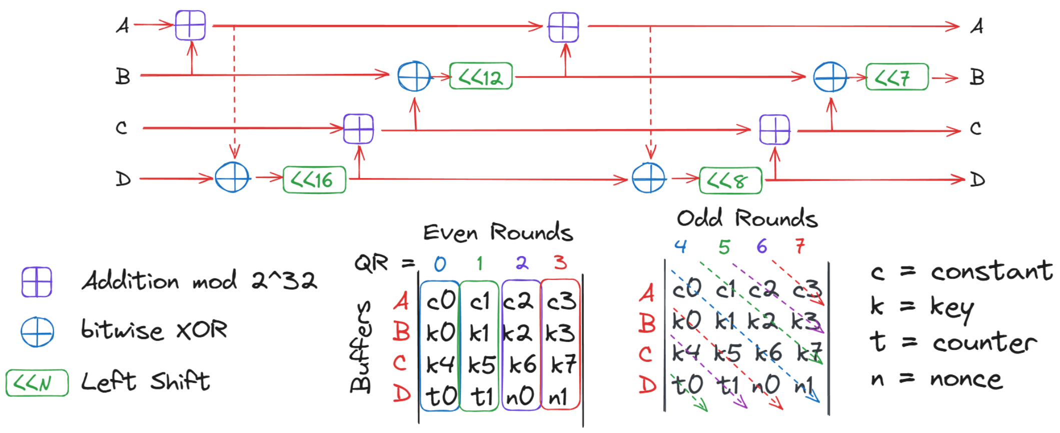

Figure 4. Scheme of the QR and use of the 32 bits elements on ChaCha's state matrix.

ChaCha.

The "E" buffer (called "pollen") is collected in a circular manner from the first line of the matrix, starting by the element K_3. Due to this technique, Forro uses less rounds to deliver the same security level of ChaCha, with a reduction of about 30% in the number of rounds. Furthermore, Forro has more *addition modulo* 2^{32} operations and less *shift* and *XOR* operations, while the *shift* parameters are chosen through experiments to maximize resistance against differential analysis attacks.

Recently, an improved version of a differential analysis attack was proposed, reducing the security level of an incomplete Forro cipher with only 5.5 rounds (that is, 22 QRs before the finalization) to around $2^{225.04}$ and the security of Forro with 5 rounds to $2^{144.14}$ [Kumar Sharma *et al.*, 2024]. That said, the Forro6 cipher (Forro with 6 complete rounds, or 24 QRs, before finalization) still has a security level of 2^{256}, since the best known attack for key discovery with this number of rounds is brute-force.

Although Forro6 currently provides the same level of security as an exhaustive key search, this does not preclude the emergence of more sophisticated attacks that may reduce or break the security of the 6-round algorithm. Therefore, the adoption of more rounds is recommended. The ChaCha algorithm is standardized with 8, 12 and 20 rounds, while the recommended number of rounds for Forro are 6, 10 and 14 to achieve the equivalent security levels of ChaCha.

5 Stream ciphers for remote attestation based in programmable data planes

Remote attestation in programmable data planes is a way to enable the periodic verification of the status of network devices managed by a Verifier. The idea is to divide the network devices in a way that a programmable data plane can perform the verification on behalf of a central Verifier, turning it into a *distributed* remote attestation.

Since programmable data planes is a resource-constrained environment, the use of algorithms that requires high usage of memory (such as digital signature) and many iterative processing rounds (such as hash algorithms) are not efficient in terms of resource usage and throughput [Ben-Basat *et al.*, 2018].

Considering that digital signatures provide authenticity to an attestation proof and that hashing algorithms produce a pseudorandom digest based on relevant information about a device, the use of a keyed hash could provide the same security services expected in both cases by using a shared secret key.

In this context, the stream cipher's round function could be used to securely generate a key-based pseudorandom value. The expected state about a device should not change frequently, so a hash of the expected configuration of the device could be used together with a 256 bit key to generate a secure proof of the device's state using the pseudorandom function.

Therefore, only one 512 bit keystream would be enough as the attestation proof. This work investigated a way of making this approach viable by testing which stream cipher algorithm would perform better in programmable data planes for encrypting 512 bits of data, which is essentially the same as calculating the verification of a proof and XORing it with the provided value to verify if they match.

The next sections will cover how Forro is implemented in a programmable data plane of a Tofino switch and its performance in resource usage and network throughput. Later, we will describe how to use Forro round functions to generate a valid integrity proof and verify it in the programmable data plane.

6 Forro implementation and analysis

The Forro algorithm was implemented[1] and evaluated on a programmable switch, and compared to both parallel and sequential implementations of the ChaCha stream cipher. The chosen SDN platform was the Intel Tofino ASIC. This ASIC is composed by two pipelines (Ingress and Egress) with 12 processing stages each, for a total of 24 processing stages. The programmable switch chosen for the experiments was the Edgecore Wedge 100BF-32X model DCS800 with 32 100Gbps QSFP28 ports. [Edgecore-Networks, 2024] The Tofino platform has several limitations on its pipeline programming to ensure operation at line rate without bottlenecks. As a switch architecture premise, it is not possible to use

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

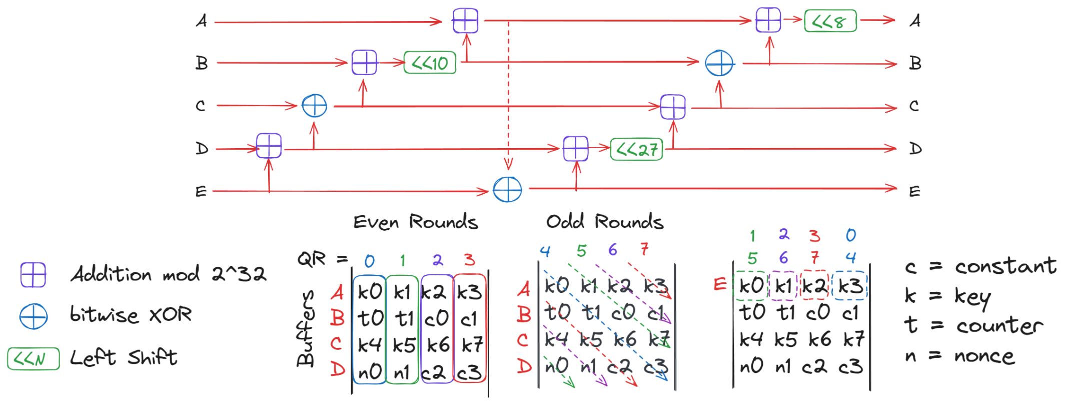

Figure 5. Scheme of the QR and use of the 32 bits elements on Forro state matrix.

loops in packet processing, making it necessary to recirculate packets to achieve iterative processing. Therefore, the switch offers two internal packet recirculation ports. It is also possible to use physical ports as loopback ports for recirculation.

The parallelized implementation of ChaCha on this architecture processes 4 QRs per pipeline (Ingress or Egress) traversal, since there is no dependency between each QR result for each round. Therefore, one Ingress pipeline and one Egress pipeline traversal (a switch traversal) can process 2 rounds of the algorithm. With only 10 packet recirculations, it is possible to process all 20 rounds of ChaCha20. To manage the intermediary states during data processing, a new header is inserted after the Ethernet header containing three fields: the operation mode (encrypt or decrypt), the index of the current round, and the index (varying from 0 to 5) of each set of 512 bits of the message. This header structure is shown in Figure 6.

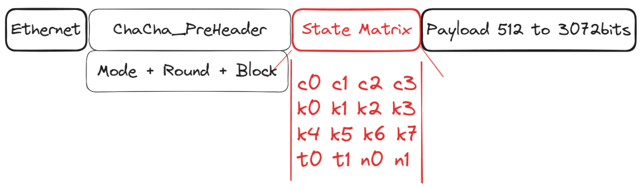

Figure 6. Packet header structure for the processing of parallel ChaCha20 in programmable data plane.

When in encryption mode, the implementation generates a nonce in the Tofino ASIC, while in decryption mode, it uses a nonce provided alongside the ciphertext. For the sake of simplicity, the performance tests of all algorithms use a fixed nonce to perform decryption of a randomly generated message (Payload).

On a direct implementation, the algorithm uses 12 basic operations to process each QR, using all the stages of the pipeline for processing the round. This pipeline occupation prevents its use for other operations, such as initialization and finalization of the state matrix and encryption and decryption (XOR). Nonetheless, there is a special instruction in Tofino called *Identity Hash* that can be used to perform the *shift* operation along with a XOR or add operation in the same stage, requiring only a few adaptations to the P4 code. By doing so, it is possible to perform the initialization together with the first round of the algorithm and the finalization together with the last round, requiring only one extra recirculation to perform the final XOR operation between the keystream and the payload. Figure 7 shows the packet traversal for the calculation of the state matrix.

The same technique was used for the implementation of Forro. However, due to Forro's sequential nature, it was not possible to process more than one QR per pipeline traversal, requiring four pipeline traversals to process each of the four QRs in one round of the algorithm. With that, it is necessary to perform a total of 28 packet recirculations to process all 14 rounds, resulting in 56 pipeline traversals for the calculation of the state matrix, and one extra recirculation and traversal for the final XOR operation between keystream and message. The amount of recirculations affects the algorithm's maximum throughput, since the same packet needs to traverse the switch several times before it's sent to its destination. Furthermore, the recirculations ports have packet-in queues for concurrent processing of the packets in the pipeline with other switch ports. This sequential implementation performs encryption of only 512 bits of data, working with the minimal payload size. Figure 8 shows the design of the Forro implementation in this programmable data plane.

To process only 512 bits, the Forro packet header was reduced compared to the one used in ChaCha's implementation, as shown in Figure 9.

For a more straightforward resource usage comparison for the Forro implementation, a sequential version of ChaCha was also implemented; that is, a version without parallelizing QR calculations in each stage. This implementation was based on Forro's implementation, only changing which operations are performed in each QR and the number of recirculations needed to perform all 20 ChaCha rounds. As with Forro, the packet header was also reduced for this sequential ChaCha implementation. In this case, 40 recirculations and 81 pipeline traversals were required to perform all state matrix calculations and the final XOR operation.

As ChaCha is parallel by design, this sequential version was implemented just for comparison with Forro and is not expected to be used in production environments.

The test vectors available in RFC7539 [Nir and Langley, 2015] and the official repository of the Forro14 [Coutinho, 2023] were used to verify the correctness of the implementations.

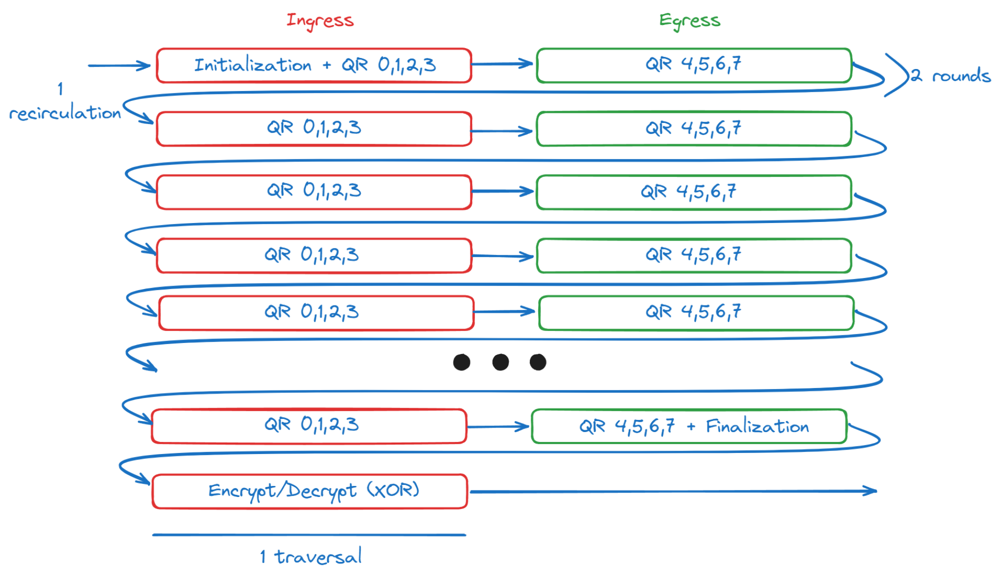

Figure 7. Parallel Chacha20 implementation in decryption mode on programmable data plane.

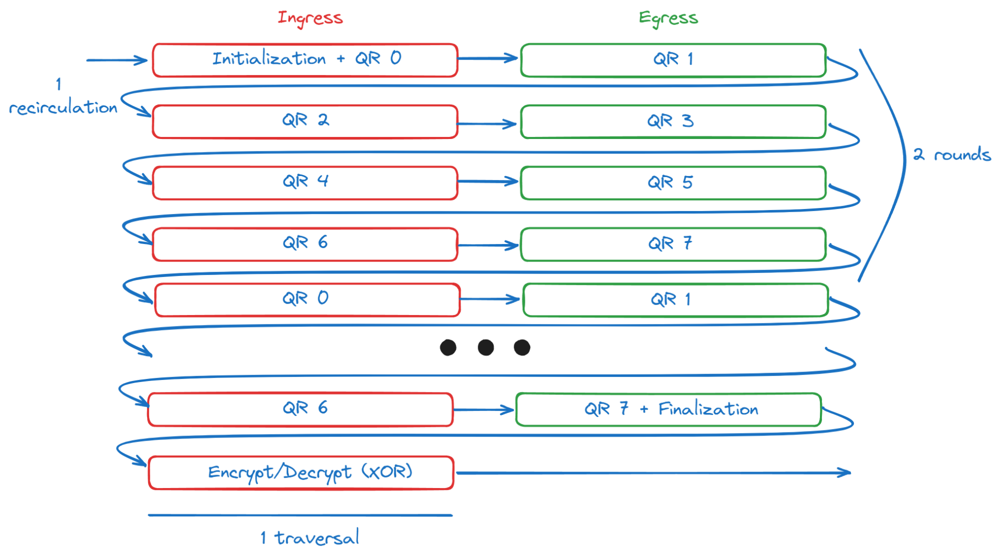

Figure 8. Design of the Forro algorithm implementation on programmable data plane.

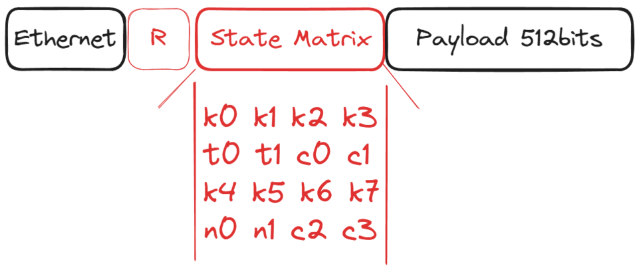

Figure 9. Packet header structure for the processing of Forro in the programmable data plane. "R" stands for Round counter.

7 Experimental evaluation of Forro in PDP

7.1 Testbed

To evaluate the algorithm's performance in programmable data planes, a testbed was built using two Tofino-based switches: one was a *Traffic Generator* (TG) running PIPO-TG [Costa, 2023] to generate traffic with predefined headers with up to 100 Gbps of throughput, while the other switch ran the implemented *Stream Cipher* algorithm (SC).

The generated traffic is forwarded to the SC switch through a 100 Gbps DAC (Direct Attached Copper) cable with QSFP28 connectors, and then forwarded back to the TG switch through another cable of the same type. The link is configured to run on 100Gbps without FEC (Forward Error Correction) and disabled auto-negotiation. For packet recirculation, SC has 2 internal ports running on 100Gbps. Those ports were used in the experiments to reduce the packet loss due to queue saturation without occupying the switch's physical ports.

The algorithms have both nonce and key statically defined in their code. Therefore, the random values generated by the TG switch as payload are always decrypted using the same key and nonce. It must be noted that the switch does not have any cache structure that would accelerate the decryption by using the same parameters.

The size of the encrypted payload used in the tests is always 512 bits. Therefore, all algorithms run with the same amount of encrypted data. Since only 512 bits are decrypted, only one state matrix is calculated per packet, so the counters are always fixed to zero. This payload size can be used in a real world scenario when performing the remote attestation of devices, as the payload of the packet is the proof provided by the Prover.

7.2 Throughput

To collect the maximum throughput after decryption, traffics ranging from 1 to 40 Gbps were generated and the versions with 6, 10 and 14 rounds of Forro and 8, 12 and 20 rounds of ChaCha were evaluated. The traffic is measured by the maximum value of the moving average from the last 10 seconds at the SC's output port, producing the results shown at Figure 10.

As shown in the graph, the implementations reach a maximum throughput for an optimal input rate, but after this point, the output rate starts decreasing in function of the input rate. The amount of recirculations required by each implementation directly impacts the maximum throughput achieved by each implementation, since more recirculation implies in more packets being queued in the packet-in queues of the internal recirculation ports. Once this queue is saturated, the switch starts dropping packets, and the more packets are dropped due to queue saturation, the lower the output rate becomes.

The parallel implementation of ChaCha shows a difference in the input-to-output rate when compared to the sequential implementations of the same algorithm. This is due the need of more input information for the switch (the mode, number of data blocks and the initial nonce) to start the payload decryption. Nonetheless, the parallel implementation keeps the same packet rate in input and output until it reaches its maximum throughput, exhibiting the same behavior as the other implementations despite these differences.

It is important to note that this difference in slope does not imply a lower encryption rate for ChaCha (Par). Rather, it is a consequence of the output packet size being smaller than the input packet size, which affects the observed output rate relative to the input rate.

As expected and shown in the graph, the ChaCha (Par) implementations exhibit consistently superior output rates compared to their sequential counterparts. When comparing the sequential implementations, Forro is also consistently better than ChaCha (Seq) in each security level. Considering this, we should evaluate the resource usage of each implementation, as parallel implementations are typically expected to consume more resources than sequential ones.

7.3 Resource usage

Another evaluation that must be made is the computing resource usage for each implementation in the SC switch. For most use cases, it is not interesting that the switch's resources are used exclusively for the encryption or decryption of packets. Therefore, it's relevant to evaluate the resource usage by those algorithms, considering that other functions such as L2/L3 forwarding will also require resources when included in the switch.

Table 1. Stream cipher algorithm's resource usage

Component	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%

To collect the resource usage, Intel provides a tool called *P4 Insight* that provides an usage estimation of the switch's resources based on the result of the P4 code compilation. Several resources composes the ASIC's architecture, and this usage can be seen on Table 1 and Figure 11.

Since the implementation uses the same amount of resources for any number of algorithm rounds and the same defined pipeline is used repeatedly for each round, the results in the table and graph can be extrapolated to any number of rounds up to 20 in ChaCha and up to 14 in Forro. The number of rounds only slightly affects the SRAM allocated in each implementation, because more table entries will be required to point out which QR should be processed in each traversal.

That is, the same amount of resources will be used considering the highest number of rounds an algorithm can perform, as the resources are allocated for the defined number of rounds during the compilation of the code. Therefore, any number of rounds of ChaCha below 20 rounds will use the same amount of resources as the implementation of ChaCha20. The same applies to Forro.

The *Action Data Bus Bytes* (ADBB) can be highlighted in this comparison, since it represents the amount of operations that can be executed in the pipeline's stages. All implementations use very little of this resource, which is positive, since switches that have to deal with many different protocols or specialized operations may quickly run out of ADBB as more concurrent operations are placed in the pipeline.

Ternary Content Addressable Memory (TCAM), *Ternary Result Bus* (TRB) and *Ternary Match Input Crossbar* (TMIX) are resources used for ternary and longest-prefix matches in the switch's tables, while *Static RAM* (SRAM), *Exact Match Result Bus* (EMRB), *Exact Match Search Bus* (EMSB) and *Exact Match Input Crossbar* (EMIX) are used for exact matches. As ternary matches resources are more scarce than exact matches resources, the implementations of Forro and the sequential implementations of ChaCha use only exact matches, freeing resources for applications that require ternary matches, like subnet-based forwarding, multicast and packet filtering. Considering that, Forro could be a better option in scenarios where operations that uses ternary matches are required.

A curious case is the usage of the *Packet Header Vec-*

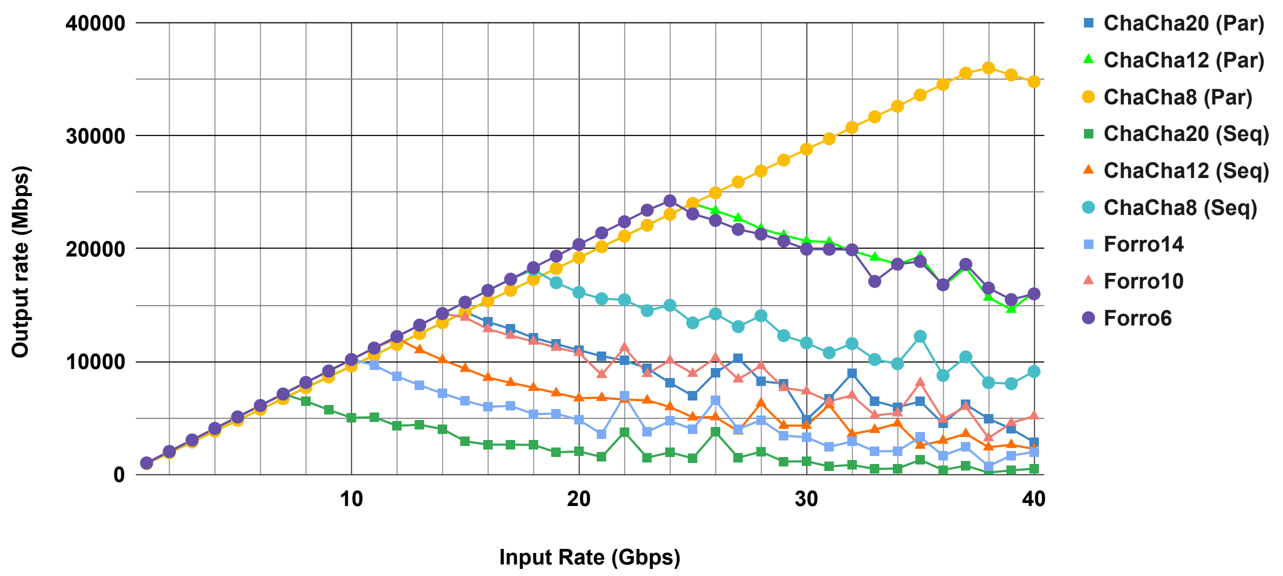

Figure 10. Tofino's output rate for each stream cipher algorithm implementation and input rate.

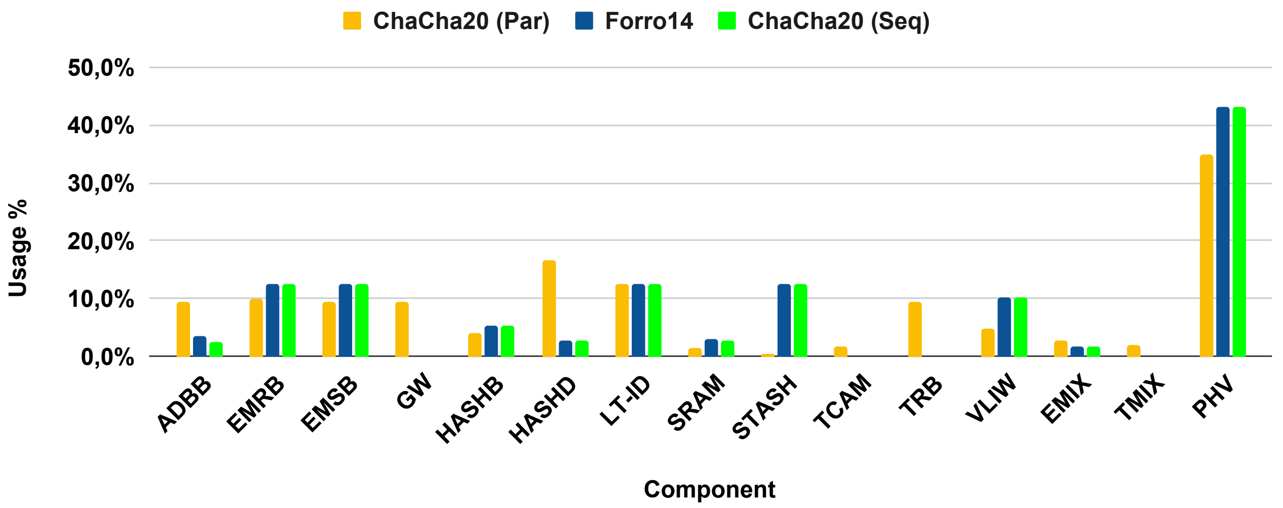

Figure 11. Tofino's resource usage for each stream cipher algorithm implementation.

tors (PHV) in each case. PHVs can be understood as CPU registers allocated in each processing stage of the pipeline to perform the arithmetic operations on the values extracted from the headers or from metadata. As the parallelized implementation of ChaCha performs more operations per stage, it is expected that the PHV allocation would be higher than the sequential algorithms, which perform only one operation per stage. However, the observed behavior is the opposite. One possible explanation for this counterintuitive PHV allocation is that each pipeline has to handle operations involving columns and diagonals of the state matrix, in contrast to ChaCha (Par), that only deals with columns in the Ingress pipeline and only with diagonals in the Egress pipeline. This prevents the flexibility of reusing the same PHV for different operations in the sequential implementations, and forces the compiler to use different PHVs for additions and XORs, including the additions in the finalization of the state matrix (Egress pipeline) and the final XOR in the encrypt/decrypt operation (Ingress pipeline). Therefore, a deeper evaluation of the allocation is required for future implementations of Forro so that the same optimized allocations obtained by ChaCha (Par) can be achieved, which would reinforce the advantage of Forro in comparison to ChaCha in terms of usage of the switch's scarce resources.

Considering that ChaCha (Par) can achieve higher throughput rates, when compared to Forro (with equivalent security level) and ChaCha (Seq), and shows a higher use of TCAM, TRB and TMIX resources, it is more efficient in scenarios where the switch is not intended to perform operations that requires ternary matches, like subnet-based forwarding, multicast and packet filtering.

Meanwhile, in scenarios where the switch is intended to perform ternary match-intensive operations, ChaCha (Par) is not the best choice. One alternative, Forro, performs better in throughput when compared to the other alternative, ChaCha (Seq). In this regard, Forro is a better alternative when the switch is expected to encrypt up to 10Gbps of its inbound traffic.

8 P4DRA: an approach for distributed remote attestation in data plane

We propose the P4-based Distributed Remote Attestation[2] (P4DRA) approach to enable proof verification in programmable data plane using the stream cipher pseudorandom function as a proof generator. In Subsection 8.1 we present some modifications in Forro's state matrix for generating proofs based on relevant information. In Subsection 8.2 is described the protocol for requesting and verifying integrity proofs using the programmable data plane. Then, we describe the P4DRA implementation in a Tofino switch in Subsection 8.3 and evaluates it in Subsection 8.4.

8.1 Forro's PRF as an attestation proof generator

Forro PRF generates a 512 bits keystream based on an input of 512 bits containing a 256 bits secret key (k). Since the key is shared between Verifier and Prover, it can provide source authenticity and prevent the generation of a proof by an adversary without knowledge of the key.

As a stream cipher basic requirement, the same keystream cannot be used to encrypt two different chunks of data, so a nonce is used to differ each calculated keystream. Likewise, to prevent proof re-use by the prover, the proof must be generated using a random and single-use value. We propose to use two different values: a 64 bits counter incremented with each request (r) to provide single-use and a 64 bit random nonce generated every request (n) to prevent proof forgery in advance. Finally, to identify each Prover, we propose a 128 bits device ID (d) unique for each device. These values are XORed with a 256 bits hash (h) of the Prover's current configuration (c), producing the hash of the Prover's configuration in the current request ($\hat{h}$). In this scenario, c could be the resulting hash of the measured boot process of the Prover or a hash of a set of configuration parameters and binaries concatenated with the measured boot's hash.

The modified version of the Forro state matrix is composed by k and $\hat{h}$ divided in 16 elements of 32 bits each. Since k is common in the original Forro state matrix and the modified version, it takes the same positions as in the original matrix, while $\hat{h}$ takes the positions of the counter, nonce and constants of the original matrix. Since the counter, nonce and constants are public parameters in Forro, we argue that their substitution does not affects the security of the keystream generation. The modified matrix is illustrated in Figure 12.

The generated keystream with the modified matrix is the attestation proof (P). If the same parameters are used as an input at the Verifier, the verification proof (V) generated will be equal to P.

[2] `https://github.com/regras/p4dra`

$$\begin{vmatrix} k0 & k1 & k2 & k3 \\ (r1\oplus h7) & (r0\oplus h6) & (d3\oplus h5) & (d2\oplus h4) \\ k4 & k5 & k6 & k7 \\ (n3\oplus h3) & (n2\oplus h2) & (d1\oplus h1) & (d0\oplus h0) \end{vmatrix}$$

r = counter d = device ID
n = nonce h = config. hash
k = key

Figure 12. Modified Forro state matrix for attestation proof generation.

8.2 Proof request and verification

In this approach, the programmable switch will perform the Verifier role, a host connected to the switch will perform the Prover role and a SDN Controller will be responsible for orchestrating the requests and keeping the state of requests. The first step is to setup the attestation environment: the Controller sends a setup packet to every host that joins the network through a secure channel, informing its device ID (d) and shared key (k). This secure channel is expected to provide confidentiality and mutual authentication in the communication between Controller and host during the *setup* phase only. The secure channel can be achieved using a manufacturer-provided key or certificate and prevents the leakage of k shared between Controller and host. The device calculates a hash of its current configuration and sends it to the Controller. Finally, the Controller sends the device's ID, key and configuration hash to the programmable switch through a dedicated out-of-band link, so it can perform the verification of proofs in behalf of the Controller.

When an attestation is required, the Controller generates and sends the request to the switch informing the request counter (r) and the random nonce (n), which then forwards it to the connected host. If the Controller requires the attestation of all hosts at once, the switch could broadcast this request to all hosts connected to it.

The host generates a proof (P) using the hash of its current configuration (h), r and n provided by the switch in the request, and d and k provided by the controller in the setup phase. The host replies the request informing r, n, d and P. The switch uses the stored h value and the received parameters to generate $\hat{h}$ and combines it with k stored to generate the verification (V).

At last, the switch checks if $P \oplus V = 0$. If true, the switch forwards the proof to the controller, so it can confirm the device state. If false, the switch silently drops the proof and the controller assumes the host is not in a correct state and can react to this information. This flow is illustrated in Figure 13.

One may argue that the switch should only forward failed attestations, as it is expected that fewer attestations should fail. However, in case of forwarding only failed attestations, an adversary could arbitrary choose r and n and re-use previous proofs to fool the switch. In this case, the switch would need to store in its tables the state of the current attestation requests. The process of a controller writing the current request's r and

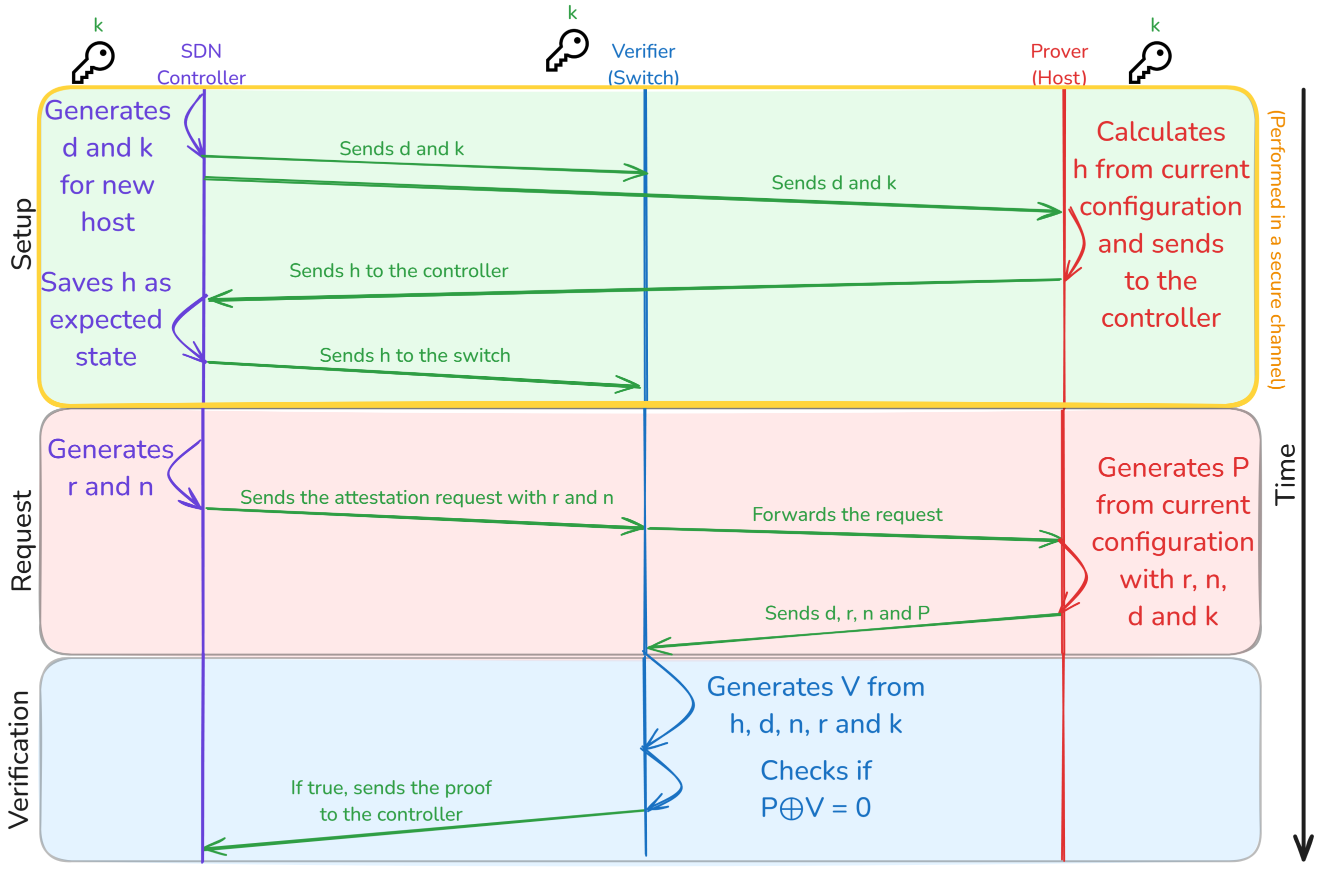
k
SDN Controller
k
Verifier (Switch)
Prover (Host)
k
Setup
Generates d and k for new host
Sends d and k
Sends d and k
Calculates h from current configuration and sends to the controller
Sends h to the controller
Saves h as expected state
Sends h to the switch
(Performed in a secure channel)
Request
Generates r and n
Sends the attestation request with r and n
Forwards the request
Generates P from current configuration with r, n, d and k
Sends d, r, n and P
Time
Verification
Generates V from h, d, n, r and k
Checks if P⊕V = 0
If true, sends the proof to the controller

Figure 13. Remote attestation flow with a programmable switch as verifier.

n in the switch tables is considerably slower (and therefore does not scale) than the controller keeping the current state and only confirming each received proof and verification.

Even so, reusing r and n values would not benefit the attacker, as the controller is not expecting those values to confirm the current state of the device.

P4DRA also has to provide the same security assumptions as the common remote attestation approach provides under a threat model. In this scenario, an attacker wants to modify the operation of a remote server to take advantage or exfiltrate sensitive data without being notice by a remote Verifier. To evade detection, an attacker can hide the changes in a non-negligible time before the attestation occurs or he can forge a valid proof and answer the attestation request. The first kind of evasion can be avoided by reducing the time frame that the attacker has to hide its presence, as proposed by this work.

To avoid the second kind of evasion, the system must *authenticate* that the proof was provided by a trusted device and that the proof was generated in a way that it represents the current state of the system (*integrity*). The first requirement can be achieved by imposing a secret that can be verified by the Verifier, like a signature or a shared secret key. The second requirement can be achieved by making the proof dependent of an unpredictable value that changes every request (the nonce n), avoiding the reuse of past proofs and the precalculation for future requests.

We argue that P4DRA is secure as long as the underlying secret key and algorithm (in this case, Forro's PRF) are secure. Since discovering the secret key implies in breaking Forro's security, the attacker cannot forge a valid proof without knowledge of the key. The attacker also should not be able to request the calculation of a proof with a self supplied configuration hash (h), as the attacker could keep a copy of the expected h value before making any modifications in the system. These requirements can be achieved by using a Trusted Execution Environment (TEE) with a signed and trusted application for obtaining the system current configuration and performing the proof calculation with safeguarded d and k.

8.3 P4DRA implementation

To perform the proposed operations, a protocol header is defined for P4DRA. this header is inserted after the Ethernet header with ethertype 0x1234 and has an initial *operation* field (p4dra.oper) that defines the format of the header and the processing to be performed by each device. All possible operations have r and n in the header, but in operation 0x5 the values of n and r are null.

The operation 0x0 is used by the controller to inform a switch to forward the attestation request to a host identified by the destination MAC address. This operation carries only r and n of the request. This operation value also allows the switch to forward the request to another switch or to decide to broadcast this request to all the hosts connected to it. Operation 0x1 is used by the switch to forward the request to the prover. The value 0x2 in the operation field is used by the prover to reply the calculated proof to the switch, informing the original r and n and concatenating the Prover's d and the generated P.

During operation 0x3, the switch calculates the verification value with the provided and stored parameters. During this operation, the switch adds a Forro round counter and the state matrix to be modified during each round. Once the state matrix is finished, the switch updates the operation field to 0x4 and recirculates the packet. Once the switch processes the recirculated packet with operation 0x4, it performs a XOR operation between the received proof and the calculated verification. If the resulting operation is not equal zero, then the switch drops the packet. Otherwise, it forwards the packet to the controller.

Operation 0x5 is used during the setup phase by the controller. This operation informs the switch to forward the packet containing encrypted d and k to the end host, so it can be securely stored by the Prover for generating proofs. After sending this packet and receiving h by the Prover, the controller inserts k and h in the switch tables indexed by d. This switch configuration is not performed using P4DRA headers, but through the switch management interface using the P4Runtime protocol. The protocol headers are illustrated in Figure 14.

The verifier pipeline execution is guided by the operation field value. The *Forward Request* operation changes the operation field to 0x1, while *Forward* does not change anything in the headers, but both forwards the packet through the specified port. The *Start Calculation* inserts the initial state matrix of the verification to the packet headers and sets the round counter (ForroRound) to start processing the matrix. *Calculates Forro14* operation performs the modifications in the state matrix as described in Section 6. At last, the *Verifies Proof* operation performs the XOR operation between the generated verification and provided proof, checking if the result is zero. Figure 15 shows the flow diagram of the Verifier in Tofino.

8.4 P4DRA evaluation

For evaluating the proposed remote attestation in a programmable data plane approach, we compare the average time to perform the verification of proofs in a x86 server running a Verifier implementation and the Tofino switch with P4DRA. In this testbed, a program sends a set of proofs to the verifier and measures the time that the Verifier uses to reply all of them. However, the Tofino Verifier is limited to 20,000 provers due to SRAM allocation in stages of the generated pipeline. Then, we evaluated scenarios from 2,500 to 20,000 provers in intervals of 2,500 provers. Figure 16 shows the average attestation time by the number of provers for a x86 Verifier and Figure 17 shows the same for a Tofino Verifier.

It can be seem that the Tofino Verifier is around 150 times faster than the x86 Verifier to verify each proof received, showing that the Tofino Verifier is a viable option for offloading the verification process to the programmable network and freeing resources in a central Verifier, as the central Verifier will not need to perform the proof verification process. This speedup enables the reduction of the attestation window that attackers could explore, making the system less susceptible to TOCTTOU attacks.

The presence of the switch as an intermediate Verifier of proofs does not pose an attack vector to the network, as the controller has a dedicated out-of-band channel with the switch. The controller is also responsible for checking if the proof forwarded by the switch follows an expected request by checking r and n provided with P.

9 Conclusions and future work

This work implemented and evaluated the performance of the Forro stream cipher in programmable data planes in the Intel Tofino architecture. The implementation was compared to two possible implementations of the ChaCha algorithm (parallel and sequential).

Even though Forro cannot achieve the same data throughput as the parallelized ChaCha, it uses less resources and is a better option for applications that requires throughput rates of up to 10Gbps, if security concerns require the use of the 14 rounds version; or up to 14Gbps, if 10 rounds are sufficient; and up to 24Gbps, if only 6 rounds suffice to provide enough security. Furthermore, it surpasses the corresponding sequential Chacha in terms of max throughput, using about the same amount of resources.

We propose an approach for enabling programmable switches to perform proof verification in behalf of a central Verifier by proposing a slight modification to Forro's PRF function. This approach enables a distributed and granular remote attestation for fast identification of compromised devices.

Finally, the proposed approach for offloading remote attestation to the programmable data plane using the Forro stream cipher can free up resources in a central verifier, with around 150 times verification speedup when compared to a x86 verifier. This speedup reduces the time window that attackers can use to perform TOCTTOU attacks, enhancing the security of the managed devices against configuration hijacking.

Regarding future works, the Tofino architecture provides two internal ports for packet recirculations. The external ports of the switch could also be used as loopback ports for recirculation, adding more in-packet queues to extend the maximum achievable output rate, as explored in ChaCha's implementation available in the literature [Yoshinaka *et al.*, 2022].

Another interesting future work would be the design of a mathematical model to characterize the relationship between these rates and the number of recirculation ports while performing the experimental evaluation for Forro and ChaCha sequential implementations for each number of recirculation ports. This model should take into consideration, among other factors, that the number of packet recirculations for each algorithm is well-defined.

When using the switch as a traffic encryption solution, other possible future work could be the implementation and evaluation of the parallelized Forro cipher (called ”Xote”) based on the calculation of more than one state matrix per pipeline traversal, highlighting its impact in throughput and resource usage. Another interesting investigation would be the use of more recirculation ports to maximize the traffic produced by encryption/decryption operations on the switch.

Another research avenue would be the implementation of

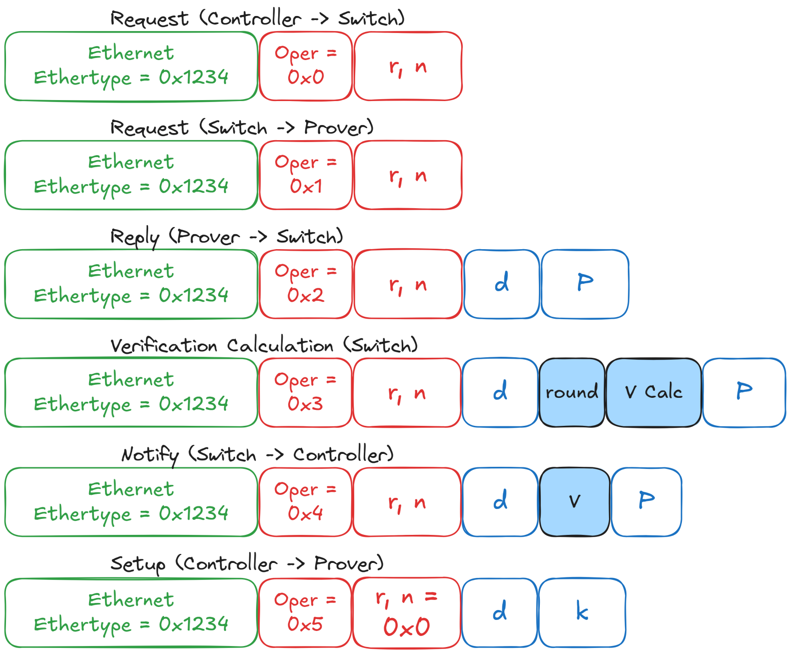
Request (Controller -> Switch)
Ethernet Ethertype = 0x1234
Oper = 0x0
r, n
Request (Switch -> Prover)
Ethernet Ethertype = 0x1234
Oper = 0x1
r, n
Reply (Prover -> Switch)
Ethernet Ethertype = 0x1234
Oper = 0x2
r, n
d
P
Verification Calculation (Switch)
Ethernet Ethertype = 0x1234
Oper = 0x3
r, n
d
round
V Calc
P
Notify (Switch -> Controller)
Ethernet Ethertype = 0x1234
Oper = 0x4
r, n
d
V
P
Setup (Controller -> Prover)
Ethernet Ethertype = 0x1234
Oper = 0x5
r, n = 0x0
d
k

Figure 14. P4DRA headers.

AEADs (*Authenticated Encryption with Associated Data*) like XChaCha20-Poly1305 [Arciszewski, 2020] and XForro14-Poly1305 [Coutinho *et al.*, 2023a] to verify the impacts of the throughput and resource usage by including integrity and authenticity guarantees into the ciphertext, which offers improved protection against some attack scenarios.

Considering the open problem of PHV allocation in the sequential algorithm, there is still work needed to refactor the code in a way that reduces the PHV allocation, which, as mentioned before, can be explored in a future work.

The proposed modification to the Forro keystream generation to generate attestation proofs modifies only public parameters. However, since these parameters are used in calculations with the key, a set of chosen parameters could potentially leak information about the secret key. A formal cryptanalysis of the proposed proof generator should be conducted in a future work to evaluate its security against chosen parameters attacks.

Declarations

Authors' Contributions

RAAP contributed to the conception of this study and is the main contributor and writer of this manuscript. CT, CRER and MAAH contributed with revisions and relevant discussions and propositions to the content. MAAH advised the development of the study and made relevant revisions to the manuscript. All authors read and approved the final manuscript.

Competing interests

The authors declare that they have no competing interests.

Acknowledgements

We thank Fabricio Rodríguez and Francisco Vogt of the Information and Networking Technologies Research and Innovation Group (INTRIG) from Unicamp's DCA for the contributions and relevant discussions for the implementation of Forro14 and sequential Chacha20 on Intel Tofino's Hardware.

Funding

This work has been performed within the framework of the FAPESP Engineering Research Center (ERC) Program under FAPESP grant agreement #2021/00199-8 (SMARTNESS).
The information in this document reflects the SMARTNESS ERC's view, but the partner institutions of SMARTNESS are not liable for any use that may be made of any of the information contained therein. The opinions, hypotheses and conclusions or recommendations expressed in this material are the responsibility of the author(s) and do not necessarily reflect FAPESP's vision or the other granting authorities. The views expressed are solely those of the authors and do not necessarily represent Ericsson's official standpoint.

Availability of data and materials

The software implementation used in this study is available at `https://github.com/regras/p4-forro`. The datasets generated and/or analyzed during the current study will be made available upon request.

References

Ambrosin, M., Conti, M., Lazzeretti, R., Rabbani, M. M., and Ranise, S. (2020). Collective remote attestation at the internet of things scale: State-of-the-art and future challenges. *IEEE Communications Surveys & Tutorials*, 22(4):2447–2461. DOI: 10.1109/comst.2020.3008879.

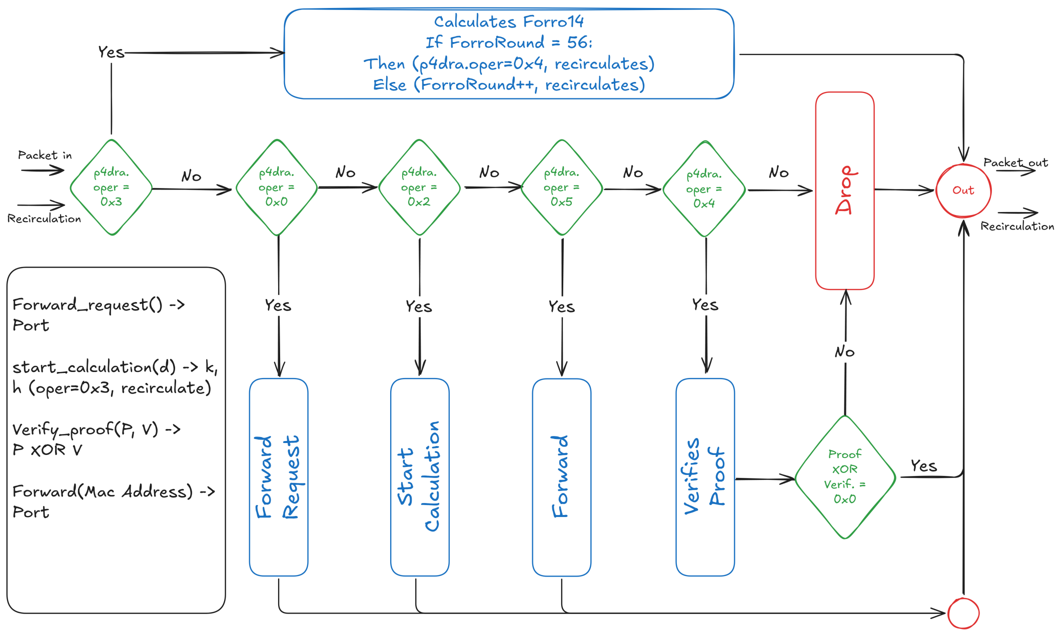

Figure 15. P4DRA implementation fluxogram.

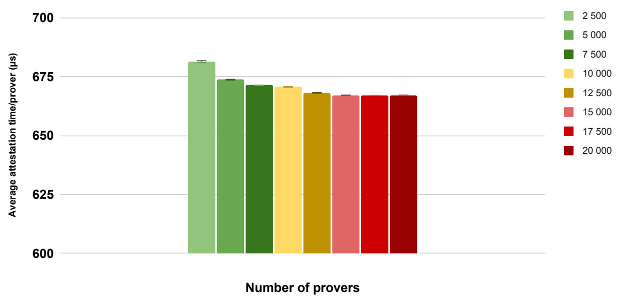

Figure 16. Average attestation time by the number of provers with a x86 verifier.

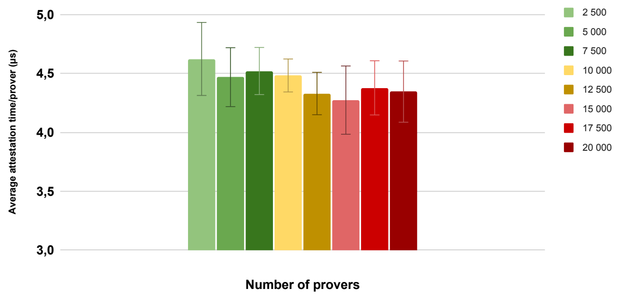

Figure 17. Average attestation time by the number of provers with a Tofino verifier.

Arciszewski, S. (2020). XChaCha: eXtended-nonce ChaCha and AEAD_XChaCha20_Poly1305. Work in Progress.

Ben-Basat, R., Chen, X., Einziger, G., and Rottenstreich, O. (2018). Efficient measurement on programmable switches using probabilistic recirculation. In *2018 IEEE 26th International Conference on Network Protocols (ICNP)*, pages 313–323. IEEE. DOI: 10.1109/ICNP.2018.00047.

Bernstein, D. J. *et al.* (2008). Chacha, a variant of salsa20. In *Workshop record of SASC*, volume 8, pages 3–5. Citeseer. Available at:`https://cr.yp.to/chacha/chacha-20080120.pdf`.

Bosshart, P., Daly, D., Gibb, G., Izzard, M., McKeown, N., Rexford, J., Schlesinger, C., Talayco, D., Vahdat, A., Varghese, G., *et al.* (2014). P4: Programming protocol-independent packet processors. *ACM SIGCOMM Computer Communication Review*, 44(3):87–95. DOI: 10.1145/2656877.2656890.

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. DOI: 10.1145/3405669.3405819.

Costa, F. G. (2023). Pipo-tg: parameterizable high performance traffic generation. DOI: https://repositorio.unipampa.edu.br/jspui/handle/riu/8840.

Coutinho, M. (2023). forro_cipher. Available at:`https://github.com/murcoutinho/forro_cipher` Online: Acesso em 28-05-2024.

Coutinho, M., Passos, I., and Borges, F. (2023a). 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*, pages 456–469, Porto Alegre, RS, Brasil. SBC. DOI: 10.5753/sbseg.2023.232879.

Coutinho, M., Passos, I., Vásquez, J. C. G., Sarkar, S., de Mendonça, F. L., de Sousa Jr, R. T., and Borges, F. (2023b). Latin dances reloaded: Improved cryptanalysis against salsa and chacha, and the proposal of forró. *Journal of Cryptology*, 36(3):18. DOI: 10.1007/s00145-023-09455-5.

Dang, H. T., Bressana, P., Wang, H., Lee, K. S., Zilberman, N., Weatherspoon, H., Canini, M., Pedone, F., and Soulé, R. (2020). P4xos: Consensus as a network service. *IEEE/ACM Transactions on Networking*, 28(4):1726–1738. DOI: 10.1109/TNET.2020.2992106.

Datta, R., Choi, S., Chowdhary, A., and Park, Y. (2018). P4guard: Designing p4 based firewall. In *MILCOM 2018-2018 IEEE Military Communications Conference (MILCOM)*, pages 1–6. IEEE. DOI: 10.1109/MILCOM.2018.8599726.

Dworkin, M., Barker, E., Nechvatal, J., Foti, J., Bassham, L., Roback, E., and Dray, J. (2001). Advanced encryption standard (aes). DOI: 10.6028/NIST.FIPS.197.

Edgecore-Networks (2024). Dcs800. Available at:`https://www.edge-core.com/product/dcs800/` Online: Acesso em 28-05-2024.

Fernandes, E. L. and Rothenberg, C. E. (2014). Openflow 1.3 software switch. *Salao de Ferramentas do XXXII Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuıdos SBRC*, pages 1021–1028. Available at:`https://intrig.dca.fee.unicamp.br/wp-content/papercite-data/pdf/fernandes2014openflow.pdf`.

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. DOI: 10.1016/j.jnca.2022.103561.

Intel (2024). Intel® tofino™. Available at:`https://www.intel.com.br/content/www/br/pt/products/details/network-io/intelligent-fabric-processors/tofino.html` Online: Acesso em 28-05-2024.

Jin, X., Li, X., Zhang, H., Soulé, R., Lee, J., Foster, N., Kim, C., and Stoica, I. (2017). Netcache: Balancing key-value stores with fast in-network caching. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 121–136. DOI: 10.1145/3132747.3132764.

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. DOI: 10.1109/ACCESS.2021.3086704.

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. DOI: 10.1109/JPROC.2014.2371999.

Kumar Sharma, N., Garai, H. K., and Dey, S. (2024). Breaching forró's security with differential-linear foray. *IEEE Access*, 12:99175–99182. DOI: 10.1109/ACCESS.2024.3429140.

Li, G., Zhang, M., Liu, C., Kong, X., Chen, A., Gu, G., and Duan, H. (2019). Nethcf: Enabling line-rate and adaptive spoofed ip traffic filtering. In *2019 IEEE 27th international conference on network protocols (ICNP)*, pages 1–12. IEEE. DOI: 10.1109/ICNP.2019.8888057.

Ling, Z., Yan, H., Shao, X., Luo, J., Xu, Y., Pearson, B., and Fu, X. (2021). Secure boot, trusted boot and remote attestation for arm trustzone-based iot nodes. *Journal of Systems Architecture*, 119:102240. DOI: 10.1016/j.sysarc.2021.102240.

Mahrach, S., Mjihil, O., and Haqiq, A. (2018). Scalable and dynamic network intrusion detection and prevention system. In *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*, pages 318–328. Springer. DOI: 10.1007/978-3-319-76354-5_29.

Nir, Y. and Langley, A. (2015). ChaCha20 and Poly1305 for IETF Protocols. (7539). DOI: 10.17487/RFC7539.

Peterson, L., Cascone, C., and Davie, B. (2021). *Software-Defined Networks: A Systems Approach*. Systems Approach LLC. Available at:`https://sdn.systemsapproach.org/switch.html`.

Scholz, D., Oeldemann, A., Geyer, F., Gallenmüller, S., Stubbe, H., Wild, T., Herkersdorf, A., and Carle, G. (2019). Cryptographic hashing in p4 data planes. In *2019 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*, pages 1–6. IEEE. DOI: 10.1109/ANCS.2019.8901886.

Sivaraman, V., Narayana, S., Rottenstreich, O., Muthukrishnan, S., and Rexford, J. (2017). Heavy-hitter detection entirely in the data plane. In *Proceedings of the Symposium on SDN Research*, pages 164–176. DOI: 10.1145/3050220.3063772.

Tan, H., Hu, W., and Jha, S. (2011). A tpm-enabled remote attestation protocol (trap) in wireless sensor networks. In *Proceedings of the 6th ACM Workshop on Performance Monitoring and Measurement of Heterogeneous Wireless and Wired Networks*, PM2HW2N '11, page 9–16, New York, NY, USA. Association for Computing Machinery. DOI: 10.1145/2069087.2069090.

TCG (2024). What is a Trusted Platform Module (TPM)? Available at:`https://trustedcomputinggroup.org/about/what-is-a-trusted-platform-module-tpm/` Acesso em: 27/09/2024.

Tokusashi, Y., Matsutani, H., and Zilberman, N. (2018). Lake: the power of in-network computing. In *2018 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, pages 1–8. IEEE. DOI: 10.1109/RECONFIG.2018.8641696.

Vieira, M. A., Castanho, M. S., Pacífico, R. D., Santos, E. R., Júnior, E. P. C., and Vieira, L. F. (2020). Fast packet processing with ebpf and xdp: Concepts, code, challenges, and applications. *ACM Computing Surveys (CSUR)*, 53(1):1–36. DOI: 10.1145/3371038.

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. DOI: 10.1145/3472873.3472881.

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. DOI: 10.1145/3565475.3569073.

Zheng, C., Rienecker, B., and Zilberman, N. (2023). Qcmp: Load balancing via in-network reinforcement learning. In *Proceedings of the 2nd ACM SIGCOMM Workshop on Future of Internet Routing & Addressing*, pages 35–40. DOI: 10.1145/3607504.3609291.