NET-SM4: A High-performance Secure Encryption Mechanism Based on In-Network Computing

Wen Wang[†‡], Shuyong Zhu[*†§], Tianyu Zuo[‡], Zhiyuan Wu[†‡], Yujun Zhang[*†‡§]
[†]Institute of Computing Technology, Chinese Academy of Sciences, China
[‡]University of Chinese Academy of Sciences, China
[§]Nanjing Institute of InforSuperBahn, China
{wangwen22s, zhushuyong, zuotianyu22s, wuzhiyuan22s, nrcyujun}@ict.ac.cn

***Abstract*—Encryption is crucial for securing critical network infrastructures, including datacenter networks, 5G networks, and the Internet of Things (IoT). In-network encryption (INE) offers a promising solution by enabling direct encryption of data on the network's data plane during transmission, thereby eliminating the need for host-side hardware encryption. However, existing INE solutions fail to fully leverage the processing capabilities of programmable switches, leading to low throughput, high resource overhead, and limited key flexibility. These limitations hinder their compatibility with other network functions and restrict their real-world deployment. To address these challenges, we introduce NET-SM4, a high-performance secure encryption mechanism based on in-network computing. NET-SM4 offloads the highly secure and pipeline-optimized SM4 encryption algorithm to programmable switches. By employing a hardware-friendly table lookup approach, NET-SM4 reduces computation dependency chains and supports parallel encryption inherently, thereby achieving high throughput and low resource overhead for in-network encryption. We implement a prototype of NET-SM4 on a commercial Tofino switch and evaluate its performance through testbed experiments and a real-world RDMA-based case study. The results demonstrate that NET-SM4 (1) outperforms state-of-the-art in-network encryption solutions in throughput by up to 293.85%, and (2) ensures link-speed data transmission with less than 20 μs overhead in real-world scenarios.**

***Index Terms*—Secure Encryption, In-Network Computing, Programmable Switch**

I. Introduction

The growing concern for network security and privacy protection has significantly increased the adoption of network encryption in various network infrastructures, including datacenter networks, 5G networks, and the Internet of Things (IoTs) [1]–[5]. Secure encryption serves as a fundamental mechanism to protect sensitive information from unauthorized access and interception, thus ensuring data confidentiality and integrity. As cyber threats evolve, high-performance encryption mechanisms are essential to conserve the Quality of Service (QoS) of the networked systems.

Traditionally, encryption operations are executed as software on end-host CPUs. However, as network bandwidth continues to scale, CPU-based encryption introduces substantial computational overhead, and has become a bottleneck for network transmission. To mitigate this issue, offloading encryption operations to dedicated hardware accelerators, such as FPGA [6]–[8], GPU [9]–[11] and SmartNIC [12], [13], has been widely explored to improve the encryption throughput. However, offloading-based solutions also lead to deployment costs for dedicated hardware and processing delay for additional intra-host communication. This delay is particularly pronounced for small data blocks, where intra-host data movement from memory to accelerators often far exceeds the calculation time. Our preliminary experiments using NVIDIA GTX 3090 GPU as the accelerator showed that even if data block size increases up to 64 MB, there remains more than 90% of the total processing time spent on essential PCIe communication between GPU and memory.

The emergence of in-network computing has opened up new possibilities for network secure encryption. In-network encryption (INE) based on programmable switches [14] has been widely explored recently. These INE solutions, include P4-AES [15], P4-Chacha [16], and P4EAD [3], directly perform secure encryption on data packets as they go through the data plane of switches, thereby eliminating the burden on end-host devices.

However, existing INE solutions still fall short of the performance required for real-world deployment. Firstly, existing solutions exhibit long computation dependency chain that significantly increase the hardware resource consumption, such as Packet Header Vector (PHV). Given the reality that hardware resources are the primary performance bottleneck for PISA architecture programmable switch [17], [18], excessive resource consumption also means constrained encryption throughput. Secondly, existing solutions have limited support of dynamic encryption keys and only support fixed keys, which severely diminishes their security guarantee. The performance bottlenecks and incomplete functionality make existing INE solutions difficult to integrate with other network functions, thus preventing their real-world deployment.

Widely used block encryption algorithm SM4 [19] offers a promising solution to address these issues and achieve high-performance in-network encryption. Firstly, it provides security guarantees comparable to prevailing algorithms such as AES [20] and ChaCha [21]. Secondly, the inherent modularity and in-place computation style of SM4 make it well compatible with the pipelines of programmable switch.

[*] Corresponding author.

In this paper, we propose NET-SM4, a high-performance secure encryption mechanism based on In-Network Computing, to achieve seamless integration of the SM4 algorithm and the data plane of high-speed programmable switch. NET-SM4 is implemented using a switch hardware-friendly table lookup method to avoid long dependency chain and improve encryption efficiency. When a plaintext data packet arrives at the switch, NET-SM4 employs a header rotation parser to extract plaintext blocks into fixed PHV, eliminating variable dependencies and mitigating PHV limitations. Additionally, NET-SM4 fully utilizes the parallelism of the switch MAU (Match-Action Unit) to process multiple plaintext blocks simultaneously, further improving the throughput. Through the compact layout in the data plane, NET-SM4 achieves high encryption throughput while ensuring key scalability and application compatibility with other network functions.

We implement a prototype of NET-SM4 with commercial Tofino [14] programmable switches and evaluated its performance through testbed experiments. We compared NET-SM4 with state-of-the-art in-network encryption solutions, i.e., P4-AES and P4-ChaCha. The results demonstrate that NET-SM4 improves throughput by up to 293.85% over these baselines. To evaluate NET-SM4 in real-world scenarios, we conducted a case study on high-speed RDMA transmission with commercial RNICs. The RDMA experiments further confirm NET-SM4's high performance and low overhead: it achieves full bandwidth utilization with a 23.4 μs total additional latency for encryption and decryption.

This paper makes the following major contributions:

- We analyze the limitation of existing INE solutions and advocate the SM4 algorithm as a promising solution (§II).
- We propose NET-SM4, a high-performance secure encryption mechanism based on in-network computing through switch hardware-friendly table lookup and pipeline-tailored parallel encryption design (§III).
- We implement a NET-SM4 prototype on commercial programmable switch and confirm its performance through testbed experiments and real-world case study (§IV and §V).

II. Background and Motivation

A. Programmable Switches

Programmable switch provides high-throughput data plane forwarding and user-defined packet processing for a wide range of applications, such as distributed machine learning [22], [23], deep packet inspection [24], [25], and DDoS attack mitigation [26]. Prevailing programmable switches follow the Protocol-Independent Switch Architecture (PISA) that comprise three core components: Parser, Reconfigurable Match-Action Table (RMT) pipeline, and Deparser. The Parser performs header extraction that converts packet headers into PHV. The RMT pipeline, which integrates multiple Match-Action Units (MAUs, a.k.a. "stages"), each supporting limited parallel table lookup and arithmetic operations for user-defined logic. The Deparser reconstructs the packet after processing logic according to their predefined headers.

However, programmable switches also have various architectural limitations. Here we introduce two of the most significant ones. The first is the strict pipeline processing mode. Although a single MAU (stage) supports parallel execution of arithmetic operations, multiple MAUs must maintain a strict serial pipeline computing mode, which limits the flexibility of packet processing. The second is the limited data-plane resources. Taking the PHV as an example, its capacity is usually only a few hundred bytes, limiting the scale and performance of switch functions.

B. SM4 Block Cipher Preliminary

SM4 is a block encryption algorithm with a block size and key length of 16 bytes. The encryption process of SM4 can be divided to 3 parts: (1) Firstly, the 16-byte *encryption key* is transformed into 32 16-byte *round keys* via predefined key expansion algorithm. (2) Secondly, in the encryption phase, an unbalanced Feistel structure is employed to iterate the *Round Function* F 32 times with different *round keys*. (3) Finally, the ciphertext is output in reverse order. Decryption reverses this process, using round keys in reverse order and outputting plaintext in reverse order as well. The core of the SM4 encryption algorithm is the **Round Function** F, which consists of **Mixed Substitution** T, **Nonlinear Transformation** τ and **Linear Transformation** L, shown as follows:

Suppose the input to the Round Function F is a tuple of four 32-bit words: $\mathbf{X} = (X_0, X_1, X_2, X_3) \in (Z_2^{32})^4$, and the round key is a 32-bit word $rk \in Z_2^{32}$, then F could be defined as:

$$F(\mathbf{X}) = X_0 \oplus T\left(X_1 \oplus X_2 \oplus X_3 \oplus rk\right) \tag{1}$$

where T represents the Mixed Substitution, which is further broken down into a combination of the Nonlinear Transformation τ and the Linear Transformation L, i.e., $T(\cdot) = L(\tau(\cdot))$.

The Nonlinear Transformation $\tau(\cdot)$ made up of four S-boxes operating in parallel. If the input to $\tau(\cdot)$ is a set of four 8-bit bytes $\mathbf{A} = (a_0, a_1, a_2, a_3) \in (Z_2^8)^4$, the corresponding output is another set of four 8-bit bytes $\mathbf{B} = (b_0, b_1, b_2, b_3) \in (z_2^8)^4$, as shown below:

$$\mathbf{B} = \tau(\mathbf{A}) = (\mathit{Sbox}(a_0), \mathit{Sbox}(a_1), \mathit{Sbox}(a_2), \mathit{Sbox}(a_3)) \tag{2}$$

The 32-bit output from the Nonlinear Transformation $\tau(\cdot)$ then serves as the input to the Linear Transformation $L(\cdot)$. For the input $B \in Z_2^{32}$, the corresponding output $C \in Z_2^{32}$ is calculated as:

$$\begin{aligned} C = L(B) = & B \oplus (B <<< 2) \oplus (B <<< 10) \\ & \oplus (B <<< 18) \oplus (B <<< 24) \end{aligned} \tag{3}$$

where "<<<" denotes a circular left shift operation.

Lastly, the assignment operation updates the state variables as shown in the following equation:

$$X_0, X_1, X_2, X_3 = X_1, X_2, X_3, C \tag{4}$$

C. Motivation

Previous studies have explored deploying encryption within the switch data plane, yet challenges such as low encryption throughput, high resource utilization, and lack of support for dynamic key management persist.

P4-AES [15] combines the AddRoundKey and SubBytes steps of the AES algorithm into 16 scrambled lookup tables, aiming to shorten the computational dependency chain. However, the static binding of the scrambled lookup table and key necessitates issuing a large number of matching tables for different keys. Moreover, encrypting a 16-byte plaintext requires a 96-byte PHV for intermediate variables, leading to significant PHV waste and preventing the encryption of multiple data blocks in a single pipeline, thus reducing throughput.

P4-ChaCha [16] distributes each encryption round across 12 pipeline stages, leaving no extra stages for other network functions. It also uses 64 bytes as intermediate variables in both the PHV and packet header, causing PHV and switch recirculation bandwidth waste. Additionally, P4-ChaCha requires extra resubmit operations.

In contrast, the SM4 algorithm offers security guarantee comparable to AES and ChaCha, with greater modularity and in-place computation conceptions. These features of SM4 allow it to have a much shorter computational dependency chain than AES and ChaCha, resulting in fewer hardware resources for intermediate variables storage and encryption calculation. These advantages inspire us to implement an in-network encryption mechanism based on the SM4 algorithm.

III. NET-SM4 Design and Implemantation

In this section, we detail the challenges and corresponding optimization measures for deploying the SM4 algorithm on Tofino programmable switches. We then propose NET-SM4, a high-performance encryption mechanism based on In-Network Computing.

A. Challenges and Optimization

Although SM4 is relatively modular and straightforward compared to AES and ChaCha, its direct implementation on a programmable switch presents unique challenges. In this section, we explore these challenges in detail and introduce the corresponding optimization implementations.

Challenge 1: How to shorten the dependency chain as far as possible? Simply implementing the SM4 algorithm on the switch will result in long computational dependency chain and a large waste of PHV. Firstly, the SM4 algorithm needs to calculate the XOR value tmp of the three plaintext values and the round key which requires at least two stages and two intermediate variables. Secondly, there is the nonlinear transformation, which requires a table lookup based on each byte of the 4-byte value tmp. This can be done in 1 stage using 4 intermediate variables. Finally, the intermediate variable result is recorded as B. Then B needs to be circularly shifted left by 0, 2, 10, 18 and 24 times respectively, which will consume 1 stage and 4 intermediate variables, and then 3 stages and two intermediate variables are used to XOR the five values. The initial acquisition and final assignment of the round key also require one stage each. Considering reuse, 4 intermediate variables, i.e. 16 bytes, are still required. This means that for every 16 bytes encrypted, an additional 16 bytes PHV are consumed to save the intermediate state. From the stage point of view, at least 8 stages are required, and the Tofino programmable switch widely used today has only 12 stages, which means that a single pipeline can only encrypt one round.

Optimization 1: Table lookup implementation for SM4. Compared with the limited stage and PHV resources, programmable switches have relatively abundant SRAM for storing matching action tables, so we hope to use more storage occupation in exchange for shorter computational dependency chains. We note that the nonlinear transformation and linear transformation can be integrated into a lookup table to reduce computational dependence. The lookup table method was first proposed by Daemen and Rijmen [27] when they implemented Rijndael on a 32-bit processor. The mixed substitution T in the SM4 encryption and decryption round function consists of a nonlinear transformation τ and a linear transformation L. The input of the linear transformation L is recorded as $B = (b_0, b_1, ..., b_{n-1}) \in (Z_2^m)^n$, and the output is recorded as $C = (c_0, c_1, ..., c_{n-1}) \in (Z_2^m)^n$. The size of m must be a multiple of the size of the S-box used by SM4, and the relationship between m and n satisfies $n = 32/m$. Since the linear transformation L only contains circular shift and XOR operation, it can be generalized as:

$$\begin{aligned} C =& L(B) \\ =& L(b_0 << (n-1)m) \oplus L(b_1 << (n-2)m) \oplus L(b_{n-1}). \end{aligned} \tag{5}$$

And mixed substitution T can be generalized as

$$\begin{aligned} C =& T(A) \\ =& L(Sbox(a_0) \ll (n-1)m) \oplus L(Sbox(a_1) \\ & \ll (n-2)m) \oplus L(Sbox(a_{n-1})). \end{aligned} \tag{6}$$

According to Eq. 6, the operation of nonlinear change T can be equivalent to a matching table of n m-bit inputs to 32-bit outputs. First, when m is equal to 8, the matching table for 4 8-bit inputs to 32-bit outputs is defined as

$$T_0[a_0] = L(Sbox(a_0) \ll 24), \tag{7}$$

$$T_1[a_1] = L(Sbox(a_1) \ll 16), \tag{8}$$

$$T_2[a_2] = L(Sbox(a_2) \ll 8), \tag{9}$$

$$T_3[a_3] = L(Sbox(a_3)), \tag{10}$$

where $a_i \in [0, 255], 0 \leq i < 4$. Therefore, the mixed substitution T can be implemented by table lookup method. The pseudo code implementation of substitution T is defined as Algorithm 1.

The size of the table is a critical factor. The size of each table is $Size_T = 2^m$ entries. When $m = 8$, each lookup table has $2^8 = 256$ entries, and one SM4 round requires

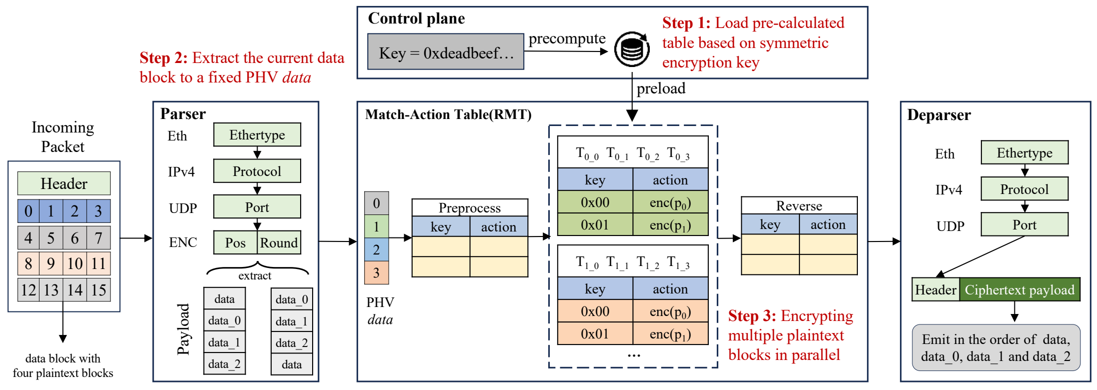


Fig. 1. The overview of NET-SM4.

```
Algorithm 1: Table Lookup Method for Mixed Substitution of $T$ in SM4
  Data: a 32-bit value A
  Result: The result of mixed subsitition $T$ in SM4
          block cipher
1 $n = 32/m$
2 $MASKm = 2 \ll m - 1$
3 $t_0 = T_0[A\&MASKm]$;
4 $t_1 = T_1[(A \gg m)\&MASKm]$;
5 $\cdots$
6 $t_{n-1} = T_{n-1}[(A \gg (n-1) * m)\&MASKm]$;
7 $T(A) = t_0 \oplus t_1 \oplus ... \oplus t_{n-1}$
```

four such tables, totaling 1024 entries. For $m = 16$, each table expands to 65,536 entries (131,072 in total), exhausting most of the data-plane SRAM, so we adopt $m = 8$. To support efficient access, we define two intermediate variables and organize the four lookups into two parallel pairs, enabling two-stage pipelined execution. Each result is XORed directly in the hit action. As shown in Fig. 2, 16-byte plaintext can be encrypted with just 5 stages and 8 bytes of PHV for intermediate variables.

Challenge 2: How to reduce the PHV utilization? Since there are multiple plaintext blocks in the data packet, if we simply use if-else logic to determine which block needs to be encrypted at runtime, all blocks will need to be extracted into the dynamic PHV, and this could easily exceed the hardware resource limitaion such as dynamic PHV, which is only 512 bytes in prevailing Tofino 1 switch. We noticed that the switch extracts packets into PHVs in the parser, so extracting encrypted blocks into fixed PHV will reduce the use of dynamic PHV.

Optimization 2: Extracting encrypted blocks to fixed PHV. We define it as d, d_0, d_1, d_2 in the header. When the parser extracts, if the first block needs to be encrypted, it is extracted in the order of d, d_0, d_1, d_2, otherwise it is extracted in the order of d_0, d_1, d_2, d. Since the data packets are saved in the order defined in the header, the rotation extraction in the parser will be saved, so that the plaintext block can be extracted to a fixed PHV d to reduce the use of dynamic PHV.

Challenge 3: How to fully unleash the encryption throughput? In the table lookup implementation, it can be optimized to use 5 stages to complete the encryption of a plaintext block, but the switch has only 12 stages and can only encrypt 2 plaintext blocks at most. More plaintext blocks carried in the data packet require the data packet to be recycled for encryption, resulting in lower encryption throughput and higher latency.

Optimization 3: Parallel encryption via Match-Action Units. We note that the RMT programmable switch can perform many arithmetic operations in parallel at each stage, so as long as the encryption of each plaintext block does not have mutual dependencies, parallel encryption can be performed in each stage. We implement NET-SM4 in CTR mode or ECB mode to achieve independent encryption of different plaintext groups. Due to the parallel capability of the Match-Action Unit, parallel encryption can significantly improve encryption throughput, but it will also bring additional table resource waste, because each lookup table in the pipeline can only be used once, which we will analyze in detail in the experimental section.

B. Design Overview

To implement a secure and efficient cryptographic primitive, we designed a high-performance encryption mechanism NET-SM4 based on In-network computing. The overall architecture and high-level design are shown in Fig. 1. We address the problems of scarce switch stage resources, limited on-chip storage, and insufficient programming flexibility from the data plane, and propose a parallel encryption mechanism to maximize encryption throughput. The packet format shown in Fig. 3 includes an ENC header with a $switch_id$ field for per-switch-pair key selection, with each encrypted data block sized at 64 bytes.

The data plane of the programmable switch is responsible for directly processing data packets. It can be divided into three components: Parser, RMT pipeline and Deparser. We

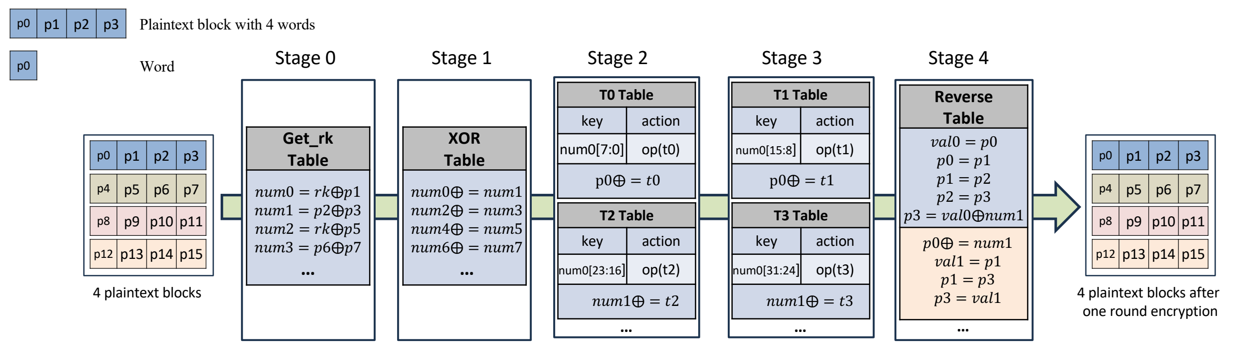


Fig. 2. Encrypt 4 plaintext blocks in parallel using 5 stages.

design different processing algorithms for each of them to alleviate the resource consumption of NET-SM4 and maximize the encryption throughput, which will be introduced in Section III-C1 and Section III-C2. The core goal of the control plane is to pre-calculate the lookup table based on the key and send it to the data plane for encryption based on the $switch\_id$. At the same time, in order to ensure that the data packet can circulate correctly, we also need to carefully design different export table entries, which will be introduced in Section III-D. Key negotiation can be securely performed within the control plane, although its implementation details are beyond the scope of this paper.

C. Data Plane Design

1) Implementation within A Single Pipeline: In the previous section, we analyzed in detail that the conventional implementation of the SM4 algorithm is not suitable for switch data plane programming, so we decided to implement the SM4 algorithm on the switch using a table lookup method.

In the table lookup implementation of the SM4 algorithm, as described in Section III-A. $p0$, $p1$, $p2$, and $p3$ are four 32-bit plaintexts in the plaintext block, and rk represents the round key currently in use. First, we find the round key rk that needs to be used currently according to the $switch\_id$ in the header and the number of encryption rounds. Since we set two intermediate variables, we can get their XOR results through three XORs in two stages. According to the table lookup implementation of the SM4 algorithm, we merge the nonlinear change tau and the linear transformation L into four 8-bit input and 32-bit output lookup tables. In order to avoid complex PHV allocation, we set the key of each table to a 32-bit intermediate variable and use ternary matching, that is, by setting a mask to represent different 8-bit hits. In these four lookup tables, we divide them into two groups for parallel search, because $p0$ and another temporary variable that is not used as a key can be assigned at this time. Thus, we have completed the encryption of a single plaintext block with only 5 stages and two intermediate variables.

However, we note that the matching-action unit has parallel computing capabilities, and using 5 stages to encrypt a plaintext group only uses a few computing instructions in each stage, so we try to encrypt multiple plaintext groups in parallel, as shown in Fig. 2. XOR is performed in parallel in the first and second stages, table lookup operations are performed in parallel in the third and fourth stages, and assignment operations are performed in the fifth stage. After testing, it was found that since each stage can only perform a maximum of 8 table lookups in parallel, a maximum of 4 plaintext groups can be encrypted in parallel, which brings a significant improvement in encryption throughput.

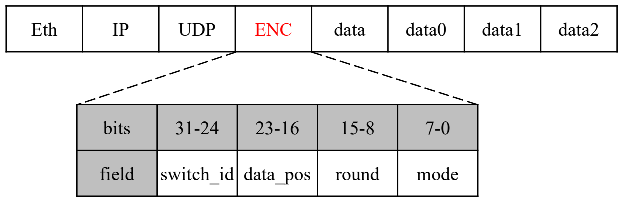


Fig. 3. Packet header with 4 data blocks.

Considering that it only takes 5 stages to encrypt each plaintext packet, we envision encrypting each packet in 2 rounds in each pipeline. In this way, in each pipeline, we first encrypt each packet once in the first 5 stages, and then encrypt each packet once in the next 5 stages. The remaining 2 idle stages and the parallel computing power of the used 10 stages can still be reserved for the regular forwarding table, which is completely sufficient for switches in data centers. This pipeline computing arrangement has doubled the throughput.

Since at most 4 plaintext blocks can be encrypted in parallel at a time, we set the data block to 64 bytes. In order to store more plaintext payloads, we save multiple data blocks in the payload. However, in switch programming, due to the lack of programming flexibility of the switch and limited PHV, we should try to reduce the branch entries in the pipeline. For example, try to avoid judging whether the first block is encrypted in this way and whether the second block is encrypted in that way, because such programming requires all blocks to be saved with the variable PHV. Therefore, we designed a flexible and effective Parser to complete the packet header rotation, that is, to move the current block to be encrypted to the front of the payload. Whenever a data packet arrives at the Parser, it will use the two variables of the current encryption block number and the number of

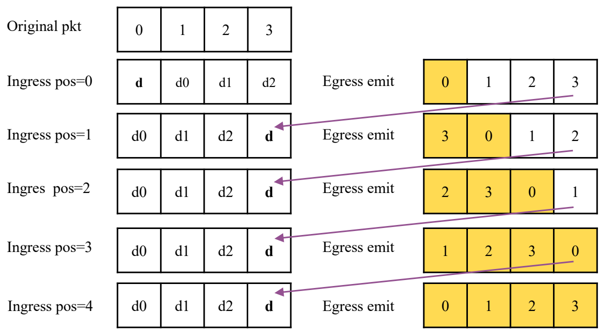


Fig. 4. Packet header rotation.

Forward Table		
Match Keys data_pos : exact, round : exact, mode : exact		
Keys	Action	Parameter
(3, 28, _)	Lpm	egress_port
else	Reciculate	recirculate_port

Fig. 5. Packet recirculate or forward match-action table.

encryption rounds to determine whether the next unencrypted packet header needs to be rotated to a fixed position, so that the pipeline is only related to fixed variables. Taking 4 data blocks as an example, the data packet header is shown in Fig. 3, and the parsing process is shown in Fig. 4. $data\_pos$ in the enc header indicates which data block is currently encrypted, and $round$ indicates the number of encryption rounds of the current data block. They all start with 0. After each encryption round, $round$ is incremented. After reaching 32, it is set to 0 and $data\_pos$ is incremented. We set the following packet header parser rules: When $data\_pos$ is equal to 1, 2, 3, and round is equal to 0, it means that this encryption is the first round of encryption for the current data block. The rotation parser is used to rotate the next unencrypted data block to the front of the payload, that is, to extract it in the order of $data0, data1, data2, data$. Otherwise, it means that the current header is the data that needs to be encrypted, and the conventional parser is used, that is, to extract it in the order of $data, data0, data1, data2$. When the Egress pipeline ends, since the packet header definition is ordered, the currently rotated data block will be saved in the order defined by the packet header, so that the block to be encrypted in the pipeline is always fixed to $data$, eliminating the variable dependency between the data plane code and the encryption block.

2) Design between Multiple Pipelines: After each pipeline encryption, round will increase. Different rounds use different round keys and forward to different output ports. When round is less than 31, it means that the current data block has not been encrypted yet. We need to specify the output port of the current data packet as a loop port in the inlet pipeline. The data packet output from the loop port will be immediately input from the current port. This is one of the characteristics of the Tofino programmable switch. Since we encrypt two rounds in the inlet and outlet pipelines, but the forwarding port can only be set in the inlet pipeline, when round is equal to 31 and the data block position reaches the threshold, that is, when round is 28 in the inlet pipeline, it means that the current data packet is about to be encrypted and forwarded to the regular port directly according to the IP longest prefix forwarding table as shown in Fig. 5. Note that before this, we still send the data packet to the loop port once more, so that we can continue to rotate the Parser once so that the data block position is completely in positive order.

D. Control Plane Design

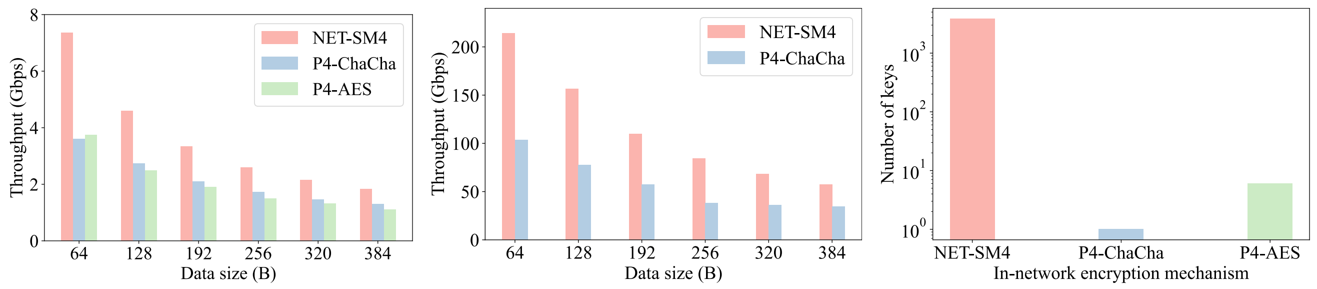
The main function of the control plane is to send down various table entries used by the data plane. Specifically, when we configure the data plane as an ingress pipeline or an egress pipeline to encrypt n rounds of packets and encrypt m packets at the same time, a total of $n * m * 4$T tables need to be sent down, each with 256 entries. The contents of the T table for each packet are consistent, and the repeated sending is because in the pipeline of the programmable switch, each matching lookup table can only be called once. The second is the rk table at the beginning of each round of encryption, which is used to obtain the round key for the current round. For the first rk table of ingress, it is also necessary to determine the forwarding port and reverse the order of the data packets encrypted in the first round of each data block. Finally, there is the operation table for each round of encryption, which is used to assign values to the 4 words in the data packet in sequence. For the operation table in egress, it is necessary to determine whether it is the 31st round so that the reversal operation can be completed after encryption.

IV. Evaluation

A. Comparison with Other In-Network Encryption Solutions

First, we compared NET-SM4 with P4-ChaCha and P4-AES. Considering the security of the encryption algorithm, P4-ChaCha uses ChaCha 20 and P4-AES uses AES-256. P4-AES only uses the inlet pipeline for encryption. Although it is believed in [15], [16] that the performance will double if the egress pipeline is used for encryption at the same time, after testing, P4-AES uses too many intermediate variables, which makes it impossible to encrypt on the egress pipeline. At the same time, P4-ChaCha extends P4-AES to support carrying multiple plaintext packets. In order to ensure the fairness of the comparison, we reuse the experimental results in P4-ChaCha. NET-SM4 uses a configuration that encrypts four plaintext blocks in parallel and two rounds of single pipeline encryption. We first explore the performance comparison of the three in the case of single recirculation port.

Setup: We connected a server to an Intel Tofino 32-port programmable switch via a 100Gbps direct-attached copper cable (DAC). The server was equipped with a Mellanox ConnectX-5 100G NIC. A DPDK-based [28] traffic generation



(a) Throughput with one recirculation port (b) Throughput with multiple recirculation ports (c) The maximum number of keys supported by NET-SM4, P4-ChaCha and P4-AES.

Fig. 6. Encryption throughput, number of supported keys of NET-SM4, P4-ChaCha and P4-AES.

system was used on the server to generate plaintext packets with payloads. Each packet carried a payload that was an integer multiple of 64 bytes, up to 384 bytes. The key used was a fixed key, that is, a pre-calculated table of fixed keys was sent to the data plane through the control plane. We gradually increase the input packet rate and observe whether the loop port is fully loaded to obtain the maximum encryption throughput under various configurations.

Results: In Fig. 6(a), we can see that under different load scales, the encryption throughput of NET-SM4 is always higher than that of P4-ChaCha and P4-AES. Compared with P4-ChaCha and P4-AES, the performance improvement of NET-SM4 reaches up to 104.58% and 293.85%. Taking 64-byte plaintext data as an example, NET-SM4 can encrypt each plaintext block for 4 rounds in a single data packet loop, and can encrypt 4 plaintext blocks at the same time, which means that 7 loops are required. The data block size of P4-ChaCha is 64 bytes, and 2 rounds of encryption per group means 10 rounds of data packet loops, and an additional resubmission operation is required. P4-AES encrypts two rounds in a single data packet loop and can only encrypt 16 bytes, and more rounds of loops are required to encrypt 64 bytes. On the other hand, P4-ChaCha needs to save an additional 64 bytes of intermediate state in the header of the data packet, which further consumes the effective utilization of the loop port bandwidth. As the plaintext load increases, the throughput of the three decreases, but NET-SM4 still achieves the best encryption throughput.

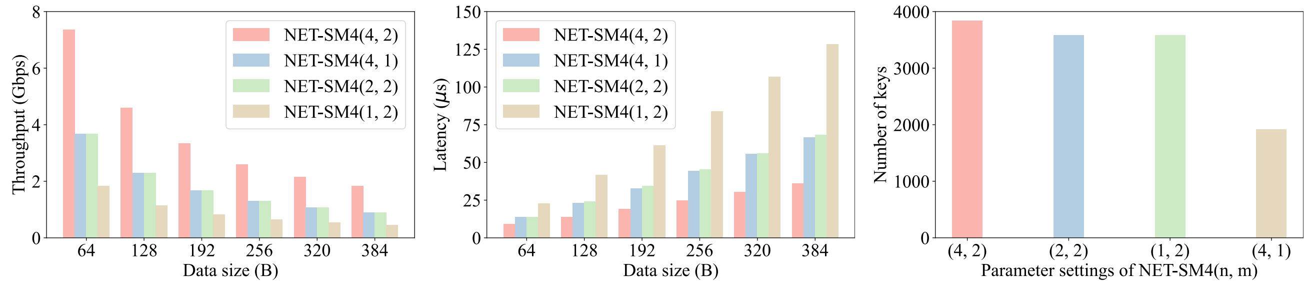
We further compare the maximum encryption throughput of each encryption implementation. Since P4-AES has a lower throughput, we only compare P4-ChaCha and NET-SM4. In order to maximize the throughput, we use a traffic generator composed of multiple computers. Specifically, for P4-ChaCha, we adopt the experimental setting in their paper, that is, the switch is configured with 5 ports for data output and the remaining 29 ports for recirculation. For NET-SM4, we use 4 ports as input and output and 30 ports for recirculation, which means there is one input port and only 6 ports can be recirculated. The traffic generator injects test traffic at a rate that saturates the recirculation port of the switch. As shown in Fig. 6(b), the maximum throughput of NET-SM4 reaches 214 Gbps, which is 106.96% higher than that of P4-ChaCha. At the same time, NET-SM4 shows significantly better performance in all cases of different plaintext payload lengths.

TABLE I
THE USE OF HARDWARE RESOURCES OF NET-SM4, P4-CHACHA AND P4-AES.

	SRAM	**TCAM**	**VLIW Instruction**	**PHV**	**Hash unit**	**Stage**
NET-SM4	10.10%	22.22%	14.58%	54.20%	0.00%	10
P4-ChaCha	1.35%	1.74%	4.69%	76.38%	16.67%	12
P4-AES	5.10%	5.56%	10.80%	68.34%	0.00%	10

Key management is also an indispensable part of the symmetric encryption process. We tested the maximum number of keys supported by NET-SM4, P4-ChaCha and P4-AES, as shown in Fig. 6(c). The pre-calculated table and key in NET-SM4 are completely decoupled, that is, different keys can reuse the same pre-calculated table, and the round key of symmetric encryption is determined by the switch ID, encryption and decryption mode, and encryption and decryption rounds. Therefore, in our experiment, up to nearly 3800 different keys are supported, which is sufficient in modern data centers. P4-ChaCha only supports 1 key because its encryption implementation completely occupies 12 stages of the data plane and there are no extra stage resources for obtaining keys. In P4-AES, pre-calculated tables need to be issued on the control plane according to different keys. Each key needs to issue 32 tables, and these tables cannot be reused. Considering that there are 12 stages in the data plane, each stage supports up to 16 table IDs, so it supports up to 6 keys. In actual use, considering the determination of the forwarding port table, the compiler automatically generates the table, and programming limitations, even 6 keys cannot be supported. Therefore, NET-SM4 is more practical than P4-ChaCha and P4-AES and supports dynamic management of multiple keys.

Table I shows the switch hardware resource usage of P4-AES, P4-ChaCha, and NET-SM4. It can be seen that NET-SM4 uses significantly less PHV resources. This is because NET-SM4 only needs to store plaintext data and very little metadata in the PHV, while P4-ChaCha and P4-AES need to store too much intermediate data in the PHV. It should be noted that since the programmable switch needs to extract the data packet to the PHV before matching operations, and the



(a) Throughput

(b) Latency

(c) The maximum number of keys supported by NET-SM4

Fig. 7. Encryption throughput, packet processing latency and number of supported keys of NET-SM4 under different parameter settings.

data plane only has a few hundred bytes of PHV, we believe that the reduction of PHV usage is very helpful for integration with other network functions. On the other hand, since NET-SM4 uses pre-calculated tables to avoid complex operations and reduce the number of stages used, it uses significantly more table and TCAM resources, but still retains more than 40% of table resources and nearly 80% of TCAM resources, which we believe is sufficient to implement other network functions.

B. Impact of configurable parameters on NET-SM4 performance

Considering that NET-SM4 has two configurable parameters, namely, the number of plaintext blocks encrypted in parallel, n, and the number of rounds of encrypting plaintext blocks in a single pipeline, m, we continue to explore how these two parameters affect the encryption performance, packet processing latency, and data plane resource consumption, and name the different configurations NET-SM4(n,m). The experimental setup is as shown in section IV-A, using a recirculation port.

Results: In Fig. 7(a), we show the relationship between encryption throughput and plaintext payload size. Overall, NET-SM4(4,2) achieves the highest encryption throughput, reaching 7.365Gbps when the payload is 64 bytes, which is roughly equivalent to a modern desktop CPU core with instruction set AES optimization. Note that only one loop port is used, and more loop ports will bring linear improvement. Under the same plaintext payload, NET-SM4(4,2) always achieves the highest encryption throughput, which is attributed to the fact that more encryption calculations can be performed in a single packet loop, thus reducing the number of packet loops. NET-SM4(2,2) and NET-SM4(4,1) achieve comparable encryption throughput under almost any plaintext payload, which is because they both incur the same number of packet loops. As the plaintext payload increases, the encryption throughput of each configuration decreases significantly because the number of packet loops increases significantly. In addition, the throughput of NET-SM4(4,2) is almost always twice that of NET-SM4(4,1), indicating that longer computation chains in a single packet pipeline do not significantly affect processing latency.

TABLE II
THE USE OF HARDWARE RESOURCES OF NET-SM4 UNDER DIFFERENT CONFIGURATIONS.

	SRAM	TCAM	VLIW Instruction	PHV	Hash unit	Stage
NET-SM4(4,2)	10.10%	22.22%	14.58%	54.20%	0.00%	10
NET-SM4(2,2)	6.77%	11.11%	9.38%	44.90%	0.00%	10
NET-SM4(1,2)	5.10%	5.56%	5.47%	40.50%	0.00%	10
NET-SM4(4,1)	5.10%	11.11%	8.59%	54.20%	0.00%	5

In Fig. 7(b), we show the latency of NET-SM4 with different configurations for packet encryption under different plaintext loads. In general, NET-SM4(4,2) achieves the lowest latency result. Under various plaintext load conditions, the encryption latency is between 10-40 microseconds, which is a very low result and has almost no impact on network transmission. Under the same plaintext load, NET-SM4(4,2) always achieves the lowest latency result, while NET-SM4(2,2) and NET-SM4(4,1) achieve almost the same latency results, which directly proves that programmable switches can forward packets at line speed. Even though NET-SM4(2,2) consumes several more stages on the data plane, the latency increase it brings is much smaller than the latency caused by the packet loop, which is about nanoseconds according to our measurements.

Fig. 7(c) shows the maximum number of keys that NET-SM4 can support. NET-SM4(4,2) supports up to 3838 keys, and NET-SM4(4,1) supports a minimum of 1918 keys. This is because NET-SM4(4,2) uses more Match-Action Units, so more SRAM can be used. In real scenarios, we think that close to 2000 keys are sufficient.

In Table II, we show the use of different configurations of NET-SM4 under the largest explicit load. We can see that NET-SM4(4,2) always uses the most hardware resources, and we will analyze them one by one.

Memory: In our prototype implementation, each match table stores 256 entries. For NET-SM4(m,n), $2 * 4 * m * n$ matching tables are required. This is due to the limitations of data plane programming. Each table can only be queried once and the tables of the ingress and egress pipelines cannot be reused. Each rule requires 12 bytes (4 bytes of key, 4 bytes of mask and 4 bytes of operation data), which is about 192KB for NET-SM4(4,2). For today's switch chips with tens of MB of

SRAM, the actual memory usage is still only a small part. NET-SM4(4,2) only uses 10.1% of SRAM and 22.22% of TCAM.

VLIW Instruction: In NET-SM4, calculation instructions are mainly used for XOR and assignment operations. According to statistics, each plaintext block encryption process uses 7 XOR operations and multiple assignments. Overall, NET-SM4 uses 5%to 15%of the arithmetic operation instructions.

PHV: In the programmable switch programming model, the packet header needs to be extracted into the PHV in the parser before it can be used for the next matching calculation in the pipeline. PHV is equivalent to the register in the CPU, with a total of about hundreds of bytes. NET-SM4 uses 40% to 54% of it, and also reserves a considerable amount of PHV resources for other network functions.

Hash unit: NET-SM4 does not use hash units, but some related works of the same type do, so we also use it as an evaluation indicator of hardware resource consumption.

Stage: NET-SM4(4, 2), NET-SM4(2, 2) and NET-SM4(1, 2) all use only 10 stages for calculation, and only NET-SM4(4, 1) uses 5 stages. In addition, in the actual compilation, the compiler allocates 12 stages for NET-SM4(4, 2), but only uses 10 of them.

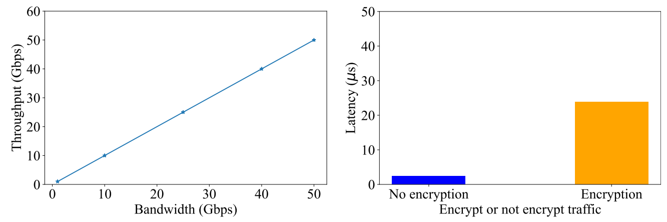
Overall, our proposed prototype implementation of NET-SM4 occupies only a small amount of specialized hardware resources on a programmable switch and should coexist with other network functions when running with other network programs, or can serve as a basic building block for other more complex data plane functions.

V. Case Study: NET-SM4 for RDMA

Remote Direct Memory Access (RDMA) has emerged as the de-facto standard for high-performance datacenter networking [29], [30]. By offloading the entire protocol stack to RDMA NICs hardware, it achieves zero-copy and kernel-bypass data transmission with high throughput, low latency, and low CPU overhead.

However, security issues have been a major concern in RDMA networks. Prevailing RDMA protocols are implemented in a plaintext communication model, which means malicious adversaries along the transmission path can intercept unprotected transport layer headers, which contain sensitive control information, as well as payload, which contain raw memory data [31], [32]. Such exposure further enables potential attacks where forged RDMA packets could directly manipulate victim memory regions. Meanwhile, recent infrastructure limitations in single-datacenter deployments for large-scale application (such as storage and LLM systems), have spurred growing interest in extending RDMA across geographically distributed datacenters to unlock the full potential of distributed systems [33]–[35]. The cross-datacenter RDMA could also introduces security challenges when operating over non-private wide-area networks (WANs).

Encryption for RDMA is a promising solution to security issues, though it's non-trival. On one hand, application-level encryption, such as TLS [36], proves incompatible with RDMA's one-side operations (i.e., read and write semantics), not mentioning it leads to heavy CPU overhead. On the other hand, although NIC-level solution like sRDMA [12] and NVIDIA ConnectX-7's IPSec offloading [37] provide packet encryption, they mandate costly hardware upgrades and remain incompatible with legacy infrastructure.



(a) Comparison of throughput between encrypted and unencrypted traffic

(b) Comparison of end-to-end latency between encrypted and unencrypted traffic

Fig. 8. Impact of NET-SM4 on cross-datacenter RDMA traffic.

We advocate NET-SM4 as a solution for cross-datacenter RDMA encryption, due to it's multi-faceted advantages. First, by deploying cryptographic processing at Data Center Interconnect (DCI) switches, NET-SM4 operates transparently to endpoint devices, requiring no modifications to existing NICs or protocol stacks. Second, NET-SM4 reduces deployment costs by processing aggregated traffic at WAN gateways and eliminating per-connection memory overheads in endpoint-based solutions. Additionally, switch-level implementation enables datacenter-wide key management through a unified control plane, simplifying security operations compared to distributed key negotiation protocols.

We evaluate the efficacy of NET-SM4 for RDMA. We build a testbed with two Tofino programmable switches, where NET-SM4 is deployed and performs packet encryption and decryption respectively, and two hosts with Mellanox ConnectX-5 RNIC. We generate RDMA traffic by OFED perftest [38], a standard performance validation tool for RDMA. We evaluate the correctness, performance, and overhead of NET-SM4 for RDMA:

Correctness. The prototype preserves RDMA's inherent reliability mechanisms – encrypted packets maintain compatibility with built-in CRC verification, ensuring data integrity without protocol modifications.

Performance. We measure RDMA throughput under under varying inter-datacenter link capacity. As shown in the Fig.8(a), the results demonstrate that NET-SM4 achieves full bandwidth utilization as well as line-rate encryption for RDMA traffic.

Overhead. We evaluate the latency overhead of NET-SM4. As shown in Fig. 8(b), we measure the round-trip time (RTT) before and after enabling NET-SM4 at switches, with NET-SM4 introducing a 23.4us overhead. We shall note that due to the pipelined nature of NET-SM4, this overhead has no influence on throughput, and it's also negligible compared to typical cross-datacenter WAN RTT which is from 1 to 100 milliseconds.

VI. Conclusion

In this paper, we propose NET-SM4, a high-performance secure encryption mechanism based on in-network computing. NET-SM4 leverages a hardware-friendly table lookup method to shorten computation dependency chain. Furthermore, it incorporates a parallel encryption mechanism, enabling the simultaneous encryption of multiple plaintext blocks to maximize throughput. We implement a NET-SM4 prototype with programmable switch and evaluate it through extensive experiments. The results show that the encryption throughput of NET-SM4 outperforms the state-of-the-art in-network encryption methods while maintaining stronger application compatibility and dynamic key management capabilities, with an overhead of less than 20 μs.

Acknowledgment

This work was supported in whole or in part, by National Natural Science Foundation of China (U24B6012, U2333201 and 62372429), the Innovation Funding of ICT, CAS under Grant No. E461040, Pilot for Major Scientific Research Facility of Jiangsu Province of China (No.BM2021800).

References

[1] C. Stransky, D. Wermke, J. Schrader, N. Huaman, Y. Acar, A. L. Fehlhaber, M. Wei, B. Ur, and S. Fahl, "On the limited impact of visualizing encryption: Perceptions of {E2E} messaging security," in *Seventeenth Symposium on Usable Privacy and Security (SOUPS 2021)*, 2021.

[2] D. Kong, Z. Zhou, Y. Shen, X. Chen, Q. Cheng, D. Zhang, and C. Wu, "In-band network telemetry manipulation attacks and countermeasures in programmable networks," in *IEEE/ACM IWQoS*. IEEE, 2023.

[3] A. Bhatnagar, X. Z. Khooi, C. H. Song, and M. C. Chan, "P4ead: Securing the in-band control channels on commodity programmable switches," in *Proceedings of the 6th on European P4 Workshop*, 2023.

[4] X. Meng, C. Lin, Y. Wang, and Y. Zhang, "Netgpt: Generative pretrained transformer for network traffic," *arXiv preprint arXiv:2304.09513*, 2023.

[5] B. Song, B. Sun, Q. Fu, and H. Li, "Flowshredder: A protocol-independent in-network security service in the cloud," in *International Conference on Service-Oriented Computing*. Springer, 2024.

[6] X. Miao, L. Li, C. Guo, M. Wang, and W. Wang, "Bit-sliced implementation of sm4 and new performance records," *IET Information Security*, vol. 2023, no. 1, 2023.

[7] S. S. Lv, B. Li, X. Chen, and Q. Zhou, "High-performance optimization of sm4-gcm based on fpga," in *2021 International Conference on Advanced Computing and Endogenous Security*. IEEE, 2022.

[8] Y. Chen, J. Song, S. Chen, Y. Cao, J. Ye, H. Li, X. Li, X. Lou, and E. Yao, "Exploring the high-throughput and low-delay hardware design of sm4 on fpga," in *2022 19th International SoC Design Conference (ISOCC)*. IEEE, 2022.

[9] O. Hajihassani, S. K. Monfared, S. H. Khasteh, and S. Gorgin, "Fast aes implementation: A high-throughput bitsliced approach," *TPDS*, vol. 30, no. 10, 2019.

[10] W. Cheng, F. Zheng, W. Pan, J. Lin, H. Li, and B. Li, "High-performance symmetric cryptography server with gpu acceleration," in *ICICS*. Springer, 2017.

[11] Z. Wang, H. Chen, and W. Cai, "A hybrid cpu/gpu scheme for optimizing chacha20 stream cipher," in *2021 IEEE Intl Conf on Parallel & Distributed Processing with Applications, Big Data & Cloud Computing, Sustainable Computing & Communications, Social Computing & Networking (ISPA/BDCloud/SocialCom/SustainCom)*. IEEE, 2021.

[12] K. Taranov, B. Rothenberger, A. Perrig, and T. Hoefler, "{sRDMA}–efficient {NIC-based} authentication and encryption for remote direct memory access," in *USENIX ATC*, 2020.

[13] D. Kim, S. Lee, and K. Park, "A case for smartnic-accelerated private communication," in *Proceedings of the 4th Asia-Pacific Workshop on Networking*, 2020.

[14] Intel tofino programmable switch. [Online]. Available: https://www.intel.com/content/www/us/en/products/details/network-io/intelligent-fabric-processors/tofino.html

[15] X. Chen, "Implementing aes encryption on programmable switches via scrambled lookup tables," in *Proceedings of the Workshop on Secure Programmable Network Infrastructure*, 2020.

[16] Y. Yoshinaka, J. Takemasa, Y. Koizumi, and T. Hasegawa, "On implementing chacha on a programmable switch," in *Proceedings of the 5th International Workshop on P4 in Europe*, 2022.

[17] X. Chen, S. Landau-Feibish, M. Braverman, and J. Rexford, "Beaucoup: Answering many network traffic queries, one memory update at a time," in *ACM SIGCOMM*, 2020, pp. 226–239.

[18] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese *et al.*, "P4: Programming protocol-independent packet processors," *ACM SIGCOMM CCR*, 2014.

[19] Z. Guan, Y. Li, T. Shang, J. Liu, M. Sun, and Y. Li, "Implementation of sm4 on fpga: Trade-off analysis between area and speed," in *2018 IEEE International Conference on Intelligence and Safety for Robotics (ISR)*. IEEE, 2018.

[20] V. Rijmen and J. Daemen, "Advanced encryption standard," *Proceedings of federal information processing standards publications, national institute of standards and technology*, vol. 19, p. 22, 2001.

[21] D. J. Bernstein *et al.*, "Chacha, a variant of salsa20," in *Workshop record of SASC*, vol. 8, no. 1. Citeseer, 2008.

[22] J. Fang, H. Xu, G. Zhao, Z. Yu, B. Shen, and L. Xie, "Accelerating distributed training with collaborative in-network aggregation," *IEEE/ACM Transactions on Networking*, 2024.

[23] Z. Wu, S. Sun, Y. Wang, M. Liu, K. Xu, W. Wang, X. Jiang, B. Gao, and J. Lu, "Fedcache: A knowledge cache-driven federated learning architecture for personalized edge intelligence," *IEEE TMC*, 2024.

[24] S. Gupta, D. Gosain, M. Kwon, and H. B. Acharya, "Deep4r: Deep packet inspection in p4 using packet recirculation," in *IEEE INFOCOM*. IEEE, 2023.

[25] C. Zhou, Q. Xiang, L. Pu, Z. Liu, Y. Zhang, X. Yuan, and J. Xu, "Netdpi: Efficient deep packet inspection via filtering-plus-verification in programmable 5g data plane for multi-access edge computing," *IEEE Transactions on Mobile Computing*, 2024.

[26] W. Wang, S. Zhu, Z. Wu, L. Lu, Z. Li, H. Yang, and Y. Zhang, "Radd: A real-time and accurate method for ddos detection based on in-network computing," in *ICC*. IEEE, 2024.

[27] J. Daemen and V. Rijmen, *The design of Rijndael*. Springer, 2002, vol. 2.

[28] I. Cerrato, M. Annarumma, and F. Risso, "Supporting fine-grained network functions through intel dpdk," in *2014 Third European Workshop on Software Defined Networks*. IEEE, 2014.

[29] T. Hoefler, D. Roweth, K. Underwood, R. Alverson, M. Griswold, V. Tabatabaee, M. Kalkunte, S. Anubolu, S. Shen, M. McLaren *et al.*, "Data center ethernet and remote direct memory access: Issues at hyperscale," *Computer*, 2023.

[30] C. Guo, H. Wu, Z. Deng, G. Soni, J. Ye, J. Padhye, and M. Lipshteyn, "Rdma over commodity ethernet at scale," in *ACM SIGCOMM*, 2016.

[31] B. Rothenberger, K. Taranov, A. Perrig, and T. Hoefler, "Redmark: Bypassing rdma security mechanisms," in *USENIX Security*, 2021.

[32] J. Xing, K.-F. Hsu, Y. Qiu, Z. Yang, H. Liu, and A. Chen, "Bedrock: Programmable network support for secure {RDMA} systems," in *USENIX Security*, 2022.

[33] P. Yu, F. Xue, C. Tian, X. Wang, Y. Chen, T. Wu, L. Han, Z. Han, B. Wang, X. Gong *et al.*, "Bifrost: Extending roce for long distance inter-dc links," in *IEEE ICNP*, 2023.

[34] Y. Chen, C. Tian, J. Dong, S. Feng, X. Zhang, C. Liu, P. Yu, N. Xia, W. Dou, and G. Chen, "Swing: Providing long-range lossless rdma via pfc-relay," *IEEE TPDS*, 2022.

[35] T. Zuo, T. Sun, S. Zhu, W. Li, L. Lu, Z. Du, and Y. Zhang, "Lowar: Enhancing rdma over lossy wans with transparent error correction," in *IEEE/ACM IWQoS*, 2024.

[36] E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.3," RFC 8446, Aug. 2018. [Online]. Available: https://www.rfc-editor.org/info/rfc8446

[37] NVIDIA, "Nvidia mlnx ofed documentation." [Online]. Available: https://docs.nvidia.com/networking/display/mlnxofedv24010331

[38] OFED, "Infiniband verbs performance tests." [Online]. Available: https://github.com/linux-rdma/perftest