Implementing ChaCha Based Crypto Primitives on Programmable SmartNICs

Shaguftha Zuveria Kottur  
IIIT Delhi, India  
shaguftha21079@iiitd.ac.in

Praveen Tammana  
IIIT Hyderabad, India  
praveent@cse.iith.ac.in

Krishna Kadiyala  
Texas Christian University, USA  
k.kadiyala@tcu.edu

Rinku Shah  
IIIT Delhi, India  
rinku@iiitd.ac.in

ABSTRACT

Control and management plane applications such as serverless functions and 4G/5G control plane functions are offloaded to smartNICs to reduce communication and processing latency. Such applications involve multiple inter-host interactions that were traditionally secured using SSL/TLS gRPC-based communication channels. Offloading the applications to smartNIC implies that we must also offload the security algorithms. Otherwise, we need to send the application messages to the host VM/container for crypto operations, negating offload benefits.

We propose crypto externs for Netronome Agilio smartNICs that implement authentication and confidentiality (encryption/decryption) using the ChaCha stream cipher algorithm. AES and ChaCha are two popular cipher suites, but we chose ChaCha since none of the smartNICs have ChaCha-based crypto accelerators. However, smartNICs have restricted instruction set, and limited memory, making it difficult to implement security algorithms. This paper identifies and addresses several challenges to implement ChaCha crypto primitives successfully. Our evaluations show that our crypto external implementation satisfies the scalability requirement of popular applications such as serverless management functions and host in-band network telemetry.

CCS CONCEPTS

• Networks → Programmable networks; In-network processing; • Security and privacy → Security protocols;

KEYWORDS

SmartNICs, in-network crypto primitives, ChaCha algorithm, programmable data planes

ACM Reference Format:

1 INTRODUCTION

Recent advancements in programmable data plane devices (e.g., programmable switches, smartNICs) have created opportunities to save precious CPU cycles and achieve low latencies by offloading applications to these devices [28, 32, 42]. Since smartNICs stay close to the message data path, processing delays in host/VM/container’s network stack can be avoided.

Many recent works leverage this opportunity and offloaded control and management applications [18, 25, 38, 39, 56] to smartNICs. By doing so, they observe a significant reduction in latency (i.e., Round Trip Times (RTTs)) and savings in CPU usage. Some example offloaded applications are distributed orchestrator for serverless applications [25, 39], failover manager [39], load balancer [25], consensus manager [38], and replication manager [38].

Most of these applications are distributed across multiple VMs/containers and exchange control messages frequently. Prior to offloading control/management applications to smartNICs, frameworks such as gRPC with built-in SSL/TLS library [12] were used to secure communication of application control messages. Thus, when offloading such applications, we need similar crypto-based security frameworks on smartNICs. Otherwise, the smartNIC will forward the control messages to the application VM/container for crypto operations, defeating the purpose of reducing latencies and saving host CPU cycles. Therefore, application offloads bring a trade-off between security and performance.

To secure offloaded application’s communication, one popular crypto algorithm available on today’s smartNICs is the AES-GCM block cipher. However, there is much less attention to alternate stream ciphers, such as the ChaCha cipher. Currently, the Nvidia BlueField NICs [10] support hardware public key accelerators using AES-GCM cipher suite. The ChaCha algorithm is an Add-Rotate-XOR (ARX) cipher with a CPU-friendly design that provides the same or better level of security as AES [48]. ChaCha is faster than the AES cipher as a result of the ARX operations, and has been designed to be resistant to side-channel cache-timing attacks [43].

In this paper, we design ChaCha-based crypto primitives (or APIs) on smartNIC to enable authentication, encryption, and decryption of application messages offloaded to the smartNIC, without the use of hardware co-processors. We program the Netronome’s pipeline using P4/micro-C to support ChaCha primitives inline, such that there are no digressions in the packet processing flow. However, Netronome smartNICs have restricted instruction set and limited memory, making it challenging to implement complex crypto operations. In our work here, we identify and address several such challenges and successfully implement ChaCha-based crypto primitives on Netronome smartNIC.

The key contributions of this paper are as follows:
(1) To motivate the need for in-network crypto primitives, we identify the applications offloaded to smartNICs that benefit from using these crypto primitives.

(2) We identify and address the challenges in implementing the ChaCha algorithm for authentication and confidentiality over Netronome smartNICs.

(3) Performance evaluation of ChaCha algorithm offloaded to Netronome smartNICs.

2 BACKGROUND & MOTIVATION

We now describe the use cases, the threat model, and motivation of this work.

Use case. In serverless computing, there are two main components: (1) Serverless Functions (SFs) that comprise several microservices that can either run serially, in parallel, or a combination of both; and (2) The management applications such as an orchestrator and a load balancer that manage the execution of these SFs. For instance, an SF consists of several microservices, and each microservice runs on individual containers. These containers communicate with an orchestrator indicating the outcome such as completion, output state, or an error message. The orchestrator uses these messages to invoke the dispatcher and the load balancer to start the next microservice in the SF’s workflow. Traditionally, these messages are communicated using a secure gRPC-based SSL/TLS channel.

Table 1 shows the list of applications that have demonstrated performance benefits via offloading processing to smartNICs. We must secure the control messages in such offloaded systems; otherwise, the system is vulnerable to attacks on confidentiality and authentication.

Threat model. Compromised VMs/containers in a cloud environment are vulnerable to posing a threat to other VMs or containers [34, 36, 37, 52, 55]. We make the following assumptions - 1. We do not trust the VMs or containers as they can run untrusted code from external users. 2. When the application (e.g. Orchestrator) is offloaded to smartNIC, the messages exchanged between the smartNIC and the VM/container (microservices) are sent in plain text. In terms of attacks, an adversary can listen to messages if a microservice (general-purpose user-written code) is compromised. Another possible attack is that if one of the servers is compromised at a hypervisor level (e.g., by installing a backdoor), the adversary can listen to all the state messages of all VMs/Containers running on the server.

Motivation for ChaCha20. We rely on encryption algorithms or “ciphers” to guarantee data privacy, security, authentication, and integrity. Ciphers that use the same key for encryption and decryption are symmetric ciphers; they can further be classified into block ciphers and stream ciphers. Data Encryption Standard (DES) and Advanced Encryption Standard (AES) are two commonly seen block ciphers, while Rivest Cipher (RC4), Salsa, and ChaCha20 are examples of stream ciphers. ChaCha is designed for high performance (as it requires few resources and inexpensive operations) and to prevent leakage of information through side-channel attacks [17]. Currently, TLS 1.3 [51] supports both the AES-GCM cipher and ChaCha20/Poly1305 ciphers. A recent work (L5o [49]) offloads TLS/AES-GCM processing (encryption/decryption/authentication) to smartNIC. This paper complements these efforts by offloading ChaCha-based algorithms to Netronome smartNICs.

3 DESIGN & IMPLEMENTATION

3.1 Design choices

When it comes to offloading a functionality such as ChaCha onto smartNICs, the available design choices can be categorized as follows:

1. Fastest path - offload chacha functionality using a hardware accelerator on the smartNIC
2. Slower path - offload chacha functionality leveraging NIC cores
3. Slowest path - offload chacha functionality relying on a combination of NIC cores + host CPU cores

As seen here, we do not want to send the crypto functionality to the host CPU cores since it is the slowest path for offloading encryption/decryption tasks. At the same time, however, no hardware accelerators are currently available that implement the ChaCha algorithm, which renders the first design choice not viable either. This leads us to the natural design choice of implementing the algorithm in the processing pipeline (without using hardware co-processors) of the on-path smartNIC. We refer interested readers to section §A.2 in the Appendix for further reading on On-path and Off-path smartNICs. In this paper, we make a case that the smartNIC cores can support the implementation of the ChaCha algorithm. We design the ChaCha algorithm and carry out experiments to support this claim.

3.2 The ChaCha algorithm

![ChaCha algorithm](image)

**Figure 1: ChaCha algorithm for encryption and decryption.**

 ChaCha [44] is a 256-bit stream cipher technique that uses the same key for encryption and decryption. Figure 1 shows the working of the ChaCha algorithm. The inputs to the ChaCha algorithm are a 256-bit key, a 96-bit nonce (pseudo-random number), and a 32-bit block counter.
### Design challenges

#### (a) Initial nonce generation

Nonces are random or pseudo-random numbers that are used by cryptography algorithms to secure communications from replay attacks. The nonce chosen for consecutive packets should be different; otherwise, the implementation is susceptible to chosen-plaintext attacks. The Netronome smartNIC has 48 micro engines (MEs), and each ME is loaded with the ChaCha program and processes packets independently. We need to ensure that each ME uses a different nonce so that the implementation is not susceptible to a chosen-plaintext attack. We address this challenge by using the intrinsic function $ME()$ [59] to initialize the value of the nonce. The function $ME()$ provides a 32-bit unique identifier for each micro engine. The ChaCha20 standards [44] recommend a 96-bit nonce. If the micro engine’s identifier is ‘x’, we generate the 96-bit unique initial nonce for each ME as the concatenation, $concat(x, x, x)$. Therefore, our implementation is resistant to chosen-plaintext vulnerability.

#### (b) Pseudo-random number generation

We resolved the previous challenge by using a unique identifier for each ME for the first time, but the nonce has to be generated for the next set of messages too. We resolve this problem based on the assumption that ChaCha is inherently a pseudo-random number generator. After encrypting a message, we use the 96 bits from the unused ChaCha keystream as the nonce for the next packet. That is, we use the initial nonce (using $ME()$) to encrypt the first packet and the unused 96-bit keystream bits as the nonce for the consecutive packets for resistance to chosen-plaintext attacks.

#### (c) Complex operations involved in ChaCha authentication algorithm

The widely used authentication algorithm with ChaCha is Poly-1305. The Netronome NIC hardware does not support modules operations used by Poly-1305. However, the NIC supports the CRC-32 [1] algorithm, but it is neither keyed nor collision-proof. That is, an attacker might be able to generate an alternative message that satisfies the checksum. We secure the computed hash using ChaCha encryption to overcome such probabilistic chosen-plaintext attacks. As part of future work, we plan to use approximation data structures to implement Poly-1305.

### Implementation of crypto primitives

We support the following crypto primitive:

#### (1) Encryption

The primitive, ENC indicates encryption of an input message.
(2) Decryption. The primitive, DEC indicates the decryption of an input message.

(3) Authentication. We support two authentication primitives, AUTH_set and AUTH_test. AUTH_set generates the secure hash for the message, and AUTH_test validates the input message for authentication and integrity.

We also support compound primitives such as AUTH_set + ENC and AUTH_test + DEC. Figure 2 shows the components involved in the implementation of ChaCha10 algorithm for authentication, encryption, and decryption primitives on the Netronome smartNIC data plane. We program the parser, match action tables, ingress/egress logic, and the deparser using P4 language, whereas the ChaCha encryption, decryption, and authentication algorithms are implemented in the micro-C language. We use micro-C language for the constructs that are either not supported in P4 language or if the P4 implementation is complex. For example, we cannot get the ME identifiers using P4.

The first step in the ChaCha algorithm is state initialization. The keystream generator has the following inputs, the ChaCha secret key, nonce, counter, and a constant. The ChaCha secret is initialized using an SSL/TLS-based secure channel as discussed in §3.3.2. The counter is set to 0 for each input message and incremented for each 512-bit message block. The initial and consecutive nonce initialization for message encryption is described in §3.3.1. That is, during the encryption process, the first message uses the ME identifier for the nonce, and after that, each message uses the unused keystream bits of the previous message. During the decryption process, the nonce is parsed from the packet header.

For encryption, we use the ChaCha10 algorithm [17, 45] (more details in §3.3.2) to generate the keystream. The keystream generator generates a 512-bit keystream for each message block (512 bits). The message and/or the hash are encrypted by XORing with the keystream. The XOR process is done in blocks of 32-bits since the Netronome NIC supports 32-bit words (32-bit ARM processor). For decryption, the nonce is parsed from the packet header, whereas the other parameters of the ChaCha10 state, viz., ChaCha secret key, counter, and constant, are initialized as discussed in the encryption process. Since ChaCha is a stream cipher, the decryption process is similar to the encryption process.

The Netronome NIC does not support cryptographic hash functions. To implement authentication, we use the non-cryptographic hash function, CRC-32, and secure it with ChaCha encryption (see §3.3.1). We use custom_crc32 accelerator for hashing due to its resistance to performance degradation at high loads as compared to other available hash functions [57]. The custom_crc32 function takes a maximum of 64 bytes of data for hashing; therefore, we incrementally take 64-byte blocks from the packet header and the message to calculate the final 32-bit hash.

We have further tested the correctness of our implementation as follows. We first encrypt the message on the SmartNIC, followed by decryption on the SmartNIC, and compare the decrypted message with the original message on the host machine. Fig. 3 shows the workflow for an incoming message. The P4 parser program parses the packet headers to derive the message and the requested crypto primitive. If AUTH_set or AUTH_test flag is set, the input message is hashed using custom_crc32. If the ENC or DEC flag is set, ChaCha10 encryption or decryption process is computed by repeating the algorithm for each 64-byte message block with the incremented counter value. Fig. 8 in Appendix shows the packet format of a message with information about the crypto primitives requested by the application and the corresponding parameter values.

4 EVALUATION

We designed our experiments to answer the following questions. (1) How does our crypto primitive implementation perform compared to the baselines? (2) Which class of applications will benefit by leveraging these crypto primitives? (3) After implementing the crypto primitives on the SmartNIC, how much memory is available to offload other applications?

Experiment setup. All our experiments are on two machines with AMD Ryzen 9 5950X (3.4 GHz, 16 cores, 32 threads) processor and 32GB RAM. The first machine ran a DPDK-based load generator application (dpdk-21.11), pktgen 21.11 [4], that generates IP packets with configurable message sizes. We ensure that the generated
was reported by the pktgen application. The ingress and egress
messages per sec) and processing latency (μsec) of the primitives ranges between 21 μsec to 170 μsec (~83% lower than processing at the host). The baseline:NIC for 64-byte, 128-byte, and 256-byte packet sizes, respectively. We observe similar trends in AUTH_get, DEC, AUTH_get+DEC, and message processing latency.

However, our in-network crypto-system demonstrates 67% lower latency compared to baseline:Host.

**Observations.** We observe that these throughput numbers would satisfy most of the offloaded applications. To be specific, the API invocation rate is 100 requests per sec [9] for serverless functions. With a maximum of 1000 containers [50] or 200 VMs [13] per server, the required crypto throughput ranges between 20K to 100K requests per second. Our authentication (AUTH_set) throughput is 6M messages per second and can support up to 60K serverless workflows, much higher than what is required. Consider the state replication use case for one of the popular key-value stores, Twitter. Only 10% of the total traffic requires state replication [15] which indicates that our system can scale linearly. Similarly, consider one INT packet is generated per flow. With 3.3M flows per second at a server [35, 47], our primitives can easily process them.

We varied the input load in another experiment and observed the corresponding message processing throughput to understand system scalability. We observed that the baseline:NIC program scaled linearly, and we observed the saturation throughput of ~19M messages/sec, while the AUTH_set and ENC primitives scaled linearly at low loads (up to 20% load) with observed throughput of ~4M messages per second, and the saturation throughput of ~6.8M messages per sec. The AUTH_set+ENC primitive does not scale linearly and saturates at ~4M messages per second.

The memory classes supported by Netronome smartNIC include: (1) the local memory (LM) register is used for data that is used in every packet; (2) The cluster local scratch (CLS) is used for data, which is needed for most packets and small shared tables; (3) The cluster target memory (CTM) is used for packet headers and coordination between other sub-systems; (4) The internal memory (IMEM) is used for packet bodies and medium-sized shared tables; and (5) The external memory (EMEM) is used for large shared tables.

Figure 5 shows that even after implementing crypto primitives on the NIC, we have free memory resources of up to 44% LM, up to

![Figure 4](image-url)
With the rise of Programmable Data Planes (PDPs), a multitude of researchers have extensively looked into authentication and confidentiality implementations on programmable hardware such as Barefoot Tofino or smartNICs. The authors of P4Knocking [61] present the implementation of a port knocking-based authentication mechanism as a network function that can be offloaded to PDPs. Anonymization implementations in the PDP [31] include implementations on Barefoot Tofino such as ONTAS [33] that enables anonymization of packet fields to hide Personally Identifiable Information addresses, PANEL[41] manipulates certain header fields to anonymize user information and PINOT [38] obfuscates packet headers of DNS traffic to disassociate client IP addresses from DNS requests. Unlike these anonymization approaches, we rely on encryption using the ChaCha algorithm to ensure that control packet data are not identifiable by adversaries.

PDPs have limited resources and computational capabilities, including a limited set of operations supported, which means that sophisticated primitives for cryptography cannot be realized [29]. This challenge has motivated researchers to propose workarounds that allow cryptographic functions to be implemented in PDP devices. The authors of [60] present an implementation of a secure keyed hash function, HalfSiPHash, on Barefoot Tofino while [53] implements SIP hash for three different P4 targets. In [19], the Advanced Encryption Standard (AES) algorithm is extended to PDPs using the scrambled lookup table technique. Given the relatively simpler operations required for the ChaCha algorithm than for AES, we implement the algorithm on a Netronome Agilio SmartNIC and perform experiments to show encryption and decryption using ChaCha on messages up to 256 bytes long.

In terms of offloading cryptographic functions to smartNICs, the work explored in [49] leverages the presence of hardware accelerators on SmartNICs to offload TLS handshake and data path encryption/decryption. Similarly, in [32], the TLS handshake and TCP connection setup process are offloaded to the smartNIC while the rest of the TCP stack runs on the host. The work presented here is different from the existing research. We propose three cryptographic primitives – encryption, decryption, and authentication, using the ChaCha10 algorithm that is offloaded to the smartNIC without leveraging hardware accelerators.

6 DISCUSSION

Crypto code placement. Due to constrained smartNIC resources, the crypto code placement depends on whether the smartNIC has enough CPU and memory resources to run crypto functions alongside the offloaded applications. (1) If there are enough resources, we should co-locate the crypto code and the offloaded applications on the same smartNIC; (2) Otherwise, we should implement a bump-in-the-wire design where the offloaded applications and the crypto code run on two separate smartNICs, connected port-to-port via physical cables.

Handling MTU size messages. As discussed earlier, our current implementation cannot handle messages longer than 256 bytes due to parser memory constraints. In our future work, we plan to handle MTU size messages by using design options such as, (1) reducing the number of threads per ME; (2) fragmenting packet messages longer than 256 bytes; (3) accessing the message payload without parsing.

Implement standard authentication algorithm. Our current implementation does not use ChaCha’s standard authentication algorithm, Poly-1305, as this algorithm uses modulus operation, which is not supported by the smartNIC hardware. Therefore, our system can be used to secure control and management messages shared within the data center network without using standard TLS connections. We plan to use approximation data structures to implement Poly-1305 for message authentication as part of future work.

API design. As part of future work, we plan to provide crypto primitive APIs for confidentiality and authentication. The smartNIC developers can invoke these APIs within the P4/micro C programs that offload host applications. In our future work, we plan to design a toolchain that preprocesses the crypto API calls and translates the given source program to the target program.

Portable cryptosystem design. This work is tightly bound to the target machine and is not simple to port. In our future work, we plan to provide abstractions for target-dependent components and design a portable cryptosystem.

7 CONCLUSION

We design and develop crypto primitives based on the ChaCha algorithm for applications offloaded to the Netronome Agilio smartNIC. We address challenges while implementing the cryptographic primitives - authentication, encryption, and decryption. From the evaluations, we observe that our implementation meets the processing rates required for control messages of offloaded applications.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their insightful feedback. We thank Lasani Hussain for his contribution towards evaluation; we also thank Dr. Sambuddho Chakravarty, Dr. Pravein Kannan, and Ranjitha K for their valuable comments on the earlier drafts. This research is supported by NMICPS TiHAN IIT Hyderabad faculty fellowship.
We present additional details about the Netronome architecture, on and off path SmartNIC designs and the packet format of an incoming application message here.

### A.1 Netronome architecture

In this work, we use the Netronome Agilio SoC smartNIC platform that belongs to the NFP-4000 device family. NFP-based Agilio SmartNICs supports a User Datapath Programming Model that allows users to program and customize the datapath on the SmartNIC. The NFP-4000 processor includes 48 packet processing cores (PPCs) and 60 flow processing cores (FPCs). The FPCs are programmable blocks that can run programs written in P4 and microC, while the PPCs ensure basic functionality. Each FPC is an independent 32-bit core at 800 MHz with 8 hardware threads, 32 KB instruction memory, 4 KB data memory, and CRC acceleration. The crypto accelerators shown in the figure are not yet supported by the network cards we have but there are different types of hash functions available on the hash accelerator.

![Figure 6: NFP-4000 Flow Processor Block Diagram](image-url)

The NFP data path can be customized and programmed for custom packet/flow processing using P4 and C languages. In this paper, we use the Netronome’s SDK that offers an environment supportive of both P4 and C software development. Along with code and data store for the FPCs, the NFP-4000 includes four other kinds of memory available to FPCs [5]: (i) 64 KB of Cluster Local Scratch (CLS); (ii) 256 KB of Cluster Target Memory (CTM); (iii) Internal Memory Unit (IMEM) that provides 4 MB of SRAM; and (iv) External Memory (EEMEM) that has 2 GB of DRAM with a 3 MB SRAM cache.

### A.2 On and Off-path smartNICs

Multicore smartNICs can be characterized into On-path and Off-path smartNICs based on the packet flow [19]. In on-path smartNICs, all traffic is handled by the NIC cores and tasks are executed on the smartNIC by adding logic to the processing pipeline [38]. The NIC cores are able to invoke special hardware accelerators for tasks such as crypto and compression [23]. In off-path smartNICs, the NIC’s cores are not directly on the data path from the host to the network, but instead, there exists a NIC-switch connecting the network ports, the host cores and the NIC cores[54]. The NIC-switch is a specialized hardware unit with match-action...
engines and runtime-configurable rules for routing packets [23]. Marvell LiquidIO [7] and Netronome NICs[3] are on-path smart-NICs while Mellanox BlueField[8] and Broadcom Stingray [6] are off-path smartNICs. On-path smartNICs have the advantage that the NIC cores have direct access to packet memory thus resulting in low latency packet processing while the NIC switch in off-path smartNICs allows packets to skip NIC cores and directly access host cores. Figure 7 shows the two smartNIC modes.

A.3 Packet format

Fig. 8 shows the format of an incoming message that wants to leverage crypto primitives for message protection. The sending application appends the primitive header after the transport (layer 4) header. The primitive header contains information about the crypto operations requested by the application and the corresponding parameter values.

**Type.** The first byte of the primitive header contains information about the requested crypto operations. The first four bits are the flags and the rest four bits are reserved for future use. The first bit, AUTH_set, indicates if secure hash should be computed and appended for packet data authentication. The second bit, AUTH_test, indicates if an authentication check is required. If authentication is unsuccessful, the packet is dropped. The third bit, ENC, indicates if the payload needs to be encrypted. The fourth bit, DEC, indicates if the payload needs to be decrypted.

**Secure hash.** If AUTH_test is set in the Type byte, the incoming packet already has the 32-bit secure hash that was appended by another smartNIC (end-point).

**Proto.** To indicate the presence of the primitive header, we set the protocol field of the IP header to the special value "145" (unassigned port number [11]). The protocol field value of the IP header is copied to the 8-bit proto field of the primitive header so that the original protocol value can be copied back to the IP header when the primitive header is decapsulated.

**Padding.** Our ChaCha implementation generates the keystream of 64 bytes. To align the IP payload to 64 bytes, some padding bytes are used. The count of padded bytes is specified in the 8-bit padding field of the primitive header.

**Nonce.** The ChaCha algorithm uses a 96-bit nonce. The nonce is one of the inputs to generate the keystream for encryption and decryption. The nonce value used during encryption is appended in the nonce field since the same nonce value should be used for decryption.