To Embed or Not to Embed SHA in Programmable Network Interface Cards

Diego Rossi Mafioletti*†‡, Magnos Martinello*, Moisés R. N. Ribeiro*, Marco Ruffini† and Frank Slyne†
*Federal University of Espírito Santo, Vitória, Brazil, †Trinity College Dublin, Dublin, Ireland
‡Federal Institute of Espírito Santo, Colatina, Brazil

***Abstract*—Cryptographic hash functions are widely used to provide from digital time stamping to authenticity and digital signatures, mapping an extensive collection of messages into a small set of message digests and help to secure network connection and data, consequently consuming CPU resources. P4 enables data plane customisation using a high-level programming language to facilitate in-network computing development across diverse hardware targets, including Network Interface Cards (NICs). Currently, most P4 targets do not implement secure hash functions due to a lack of hardware instructions or the absence of formal functions to expose their native hardware-based implementation. Moreover, many applications and protocols cannot be instantiated using in-network computing due to stringent requirements based on these hash functions. In order to empower the security and other hash-based applications, in this paper we propose and implement a P4 shared object library for a secure hash algorithm 2 (SHA-2). Our goal is to enable SHA-2 to be used as an embedded Network Function (eNF), overcoming the lack of support in a SmartNIC architecture, in order to address the latency and throughput requirements of Service Function Chain (SFC) forwarding performance within the Network Function Virtualization (NFV) paradigm. Thus, our prototype is evaluated against kernel-level Open vSwitch (OvS) and user-space Data Plane Development Kit (DPDK) implementations. The outcomes demonstrate different tradeoffs over each platform, from the randomness added by the OS to the high cost of executing the aforesaid function using a network programmable device, leading us to highlight the best choice for each specific application.**

***Index Terms*—In-Network Computing, P4 programming, Cryptographic hash functions**

I. Introduction

In the area of networking, one of the most CPU-demanding activities is the securitisation of communications using cryptography. Payload data processing and specialised cryptographic hash functions are commonly employed in secure and resilient communication, avoiding unauthorised access, data manipulation and modification. In-network processor-based programmable network interface cards (NIC) offer a solution for offloading network traffic, allowing hosts to process general computations onto the programmable NIC while keeping the support of high-level applications on them.

The advent of new programming paradigms like P4 [1] for high-speed packet processing platforms has enabled a wide variety of networking applications. For instance, in-network caching [2], heavy-hitter detection [3], [4], and network load balancing [5] enable these functions that were primarily designated for running into commodity servers to migrate to the data plane, using hash-based data structures like bloom filters, count–min and hash tables to track network flows directly into the data plane.

However, the P4 language currently only supports a few non-cryptographic hash algorithms based on cyclic redundancy check (CRC) or checksum computations typically used in network protocols like TCP, IPv4 and IPv6 checksums due to target hardware constraints. However, there is still a need for packet-based functionalities involving basic data security and verification. In those cases, "true" cryptographic hash functions are required, for example, hash-based message authentication codes (HMAC) [6], providing message authentication, or any other cryptographic hash functions used to increase resilience against hash collisions for hash-based applications cited early.

To address advanced secure applications implementation using in-network computing and also the disaggregation of services into different virtualised functions, we argue that complex cryptographic hash functions extensively used nowadays, like Secure Hash Algorithm 2 (SHA-2), should be ported onto P4 targets. For this end, however, one has to deal first with the limited computational resources, such as the ones available at SmartNICs [7], when enabling the offloading of secure applications to the data plane.

To the best of our knowledge, this is the first proof-of-concept implementation of an SHA-2 variant library on a commodity programmable in-networking processor using the P4 language. Our strategy is to implement a shared object library as an *"extern"* to overcome language-related restrictions. This way we manage to embed a complex algorithm that includes loop statements and other non-native features.

A testbed is also created for a benchmark study on finding out whether SHA-2 embedding as Network Function (NF) is worth not considering throughput and latency features. Those metrics are key for providing SHA along with Service Function Chaining (SFC) in a Network Function Virtualization (NFV) modern infrastructures. The SmartNIC embedded Network Function (eNF) prototype implementation is checked against other two software data plane implementations, namely, Open vSwitch software switch (OvS) and Intel Data Plane Development Kit (DPDK), both ported with the same SHA-2 library.

The remainder of this paper is organised as follows: First, we review related work in Section II. We argue for the inclusion of cryptographic hash function in programmable network cards in Section III. In Section IV we discuss our

978-3-903176-51-5

approaches for checking three different virtualisation techniques with the functionality to calculate cryptographic hashes. We then conduct an extensive evaluation of our prototype implementation focusing on performance metrics in Section IV. Section V closes the work and gives the future directions.

II. Background and Related Work

We start this section by giving an overview of cryptographic and non-cryptographic hash functions, showing the main aspects of both. We also describe the related works using the P4 data plane for implementing these algorithms in different approaches and purposes. We survey from simple implementations that made use of hash functions for classifying network flows, to more complex use-cases, such as layer 2 or layer 3 data encryption using intricate algorithms for assuring security into the programmable data plane.

A. Cryptographic hash functions

Cryptographic hash functions are one of the most important tools in the field of cryptography and are used to achieve a number of security goals like authenticity, digital signatures, pseudo number generation, digital time stamping and others.

Hash functions map a large collection of messages into a small set of message digests and can be used for error detection, by appending the digest to the message during the transmission. The error will be detected if the digest of the received message, on the receiving end, is not equal to the received message digest. With the advent of public key cryptography and digital signature schemes, cryptographic hash functions gained much more prominence. Using hash functions, it is possible to produce a fixed-length digital signature that depends on the whole message and ensures the authenticity of the message. To produce a digital signature for a message M, the digest of M, given by $H(M)$, is calculated and then encrypted with the secret key of the sender [8].

In order to provide security services, cryptographic hash algorithms need to guarantee some properties which are not necessarily guaranteed by general-purpose hash functions. The **one-way** or **preimage resistance** property of cryptographic hash functions implies that it is computationally infeasible to compute the message M given its hash $DM(M)$. The **second preimage resistance** property means that, given a hash value $DM(M)$, it is computationally infeasible to find a different message $M' \neq M$ that yields the same hash value. The **pseudo-randomness** property means that the hash value of a message must expose statistical randomness. Finally, the **collision resistance** property means that it is computationally infeasible to find a pair of messages $M1$ and $M2$ which produce the same hash value [9].

The Secure Hash Algorithm (SHA) is a family of cryptographic hash functions defined by the National Institute of Standards and Technology (NIST) and published as the Federal Information Processing Standard (FIPS) 180, Secure Hash Standard (SHS). The algorithm is an iterative, one-way hash function that can process a variable-size message to produce a fixed-size condensed representation called a message digest. This algorithm enables message integrity verification, i.e., any change to the message will, with a very high probability, result in a different message digest. This property is useful in the generation and verification of digital signatures and message authentication codes, and in the generation of random numbers or bits [10]. The SHA-2 family of cryptographic hash functions was first announced in 2001 and includes SHA-224, SHA-256, SHA-384 and SHA-512, named according to the length of the message digest created by each one, suppressing the previous SHA-1 implementation due to security improvements.

As cryptographic hash functions are compute-intensive applications, which begs the question of extending P4 and its hardware platforms into cryptographic algorithms. This would enable offloading secure applications/tasks to the data plane. The expected benefits are twofold: i) saving CPU resources for other applications running at the hypervisor and their tenants, and ii) reducing latency and increasing the number of processed packets per time unit. The latter benefit will come from avoiding the operating system stack in between the network and the SHA application.

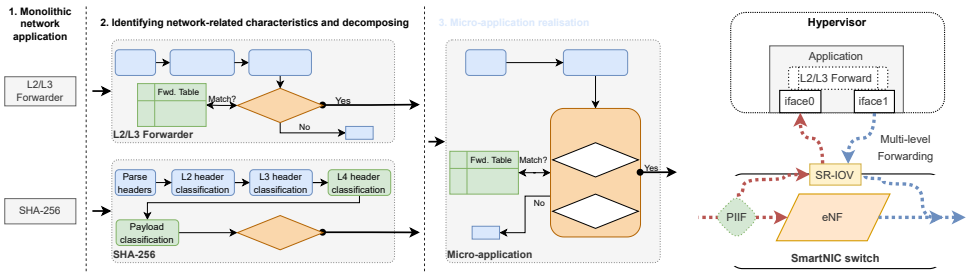
B. P4 data plane for hashing algorithms

Programmable data plane based on P4 programming language is a widespread technology that has been recently redesigning the networking programmability, allowing to redefine the behaviour of network devices (e.g., SmartNICs and programmable switches) and enabling the offloading of applications to the data plane, reducing latency and increasing the overall throughput. Regardless of the data plane programmability, certain classes of applications (mainly security-related network functions) may require specific hashing and securing functions to be offloaded to the data plane.

However, most current SmartNIC architectures [7] are not equipped with a specific encryption processor accelerator or simply do not expose an API or method for accessing it using the P4 language, supporting only simple arithmetic operations and limited lookup actions. Thus, it is not a trivial task to implement a cryptographic hash function or an encryption algorithm using P4 language on that target hardware.

Related works [4] and [3] present the use of hash functions available in P4 for optimised flow routing. However, these papers are limited to exploiting binary classification of flows (elephant/mice) employing preexisting algorithms on the target platform (e.g. checksum, CRC16, CRC32). The linear dependency between hash input and hash value makes such native algorithms vulnerable to bias and security attacks [11], [12].

In [13], the authors propose an extension of the P4 Portable Switch Architecture for cryptographic hashes over three different P4 target platforms for security, including a Netronome SmartNIC card. Nevertheless, it does not address the use of complex algorithms – such as SHA-2 – in that specific target, arguing that the application may be too big to fit in the generated firmware image, discouraging the implementation of such a secure hash algorithm version, focusing the evaluation of existing functions on the platform.



(a) (b)

Fig. 1: Decomposition for deployment and offloading: a) Illustrative example of decomposition using a set of network applications. b) Multi-level forwarding using Intercepting and Forwarding (PIIF) element.

Other related works go beyond and propose data plane secure data transmission using encryption algorithms [14], [15], aiming for layer 2 and layer 3 security, respectively, employing Advanced Encryption Standard (AES) ported as a P4 *extern* function on BMv2 software switch. The missing hardware version of the proposal is justified in both works. The authors found severe limitations imposed by the absence of variable-length payload parsing and limited packet data exchange between the P4 processing pipeline and *extern*.

Our work focuses on an implementation using a variant of the SHA-2 algorithm – the SHA-256 version based on 256-bit hash size – ported as a cryptographic library to a P4 data plane based on a Netronome SmartNIC via an *extern* written in a C-like language, enabling the generation of this kind of hashes using in-network computing and P4 language. To check the potential advantages of running such a hash function into a programmable network device, we have implemented the same hash function using the same library onto two software-based data planes: Open vSwitch switch [16] and Intel DPDK L2FWD application [17], in order to compare the overall performance and, consequently, the latency introduced by the hash function when forwarding network packets using an L2 packet forwarder.

III. An Architecture for In-Network SHA

In order to address the early-mentioned challenges, we design a simple architecture based on the following enablers: i) functional decomposition for deployment and ii) multi-level forwarding between hardware and software, which will be explained below.

A. Functional decomposition for deployment

Most network functions are deployed as a monolith application, whereby all components are bundled together into a single piece of code. This can lead to maintenance challenges, as well as slow down the trailing of new technologies. Decomposition is therefore a significant step towards evolving VNFs to be cloud-native and much more agile and scalable. And decomposing must be undertaken for the majority of the data plane, control and signalling VNFs. In this context, we define decomposition as breaking a monolithic network function into a set of small applications, or "micro-applications".

Decomposition also allows for common functions to be stripped away from the core logic of the applications. This enables the applications themselves to be lighter – which makes them easier and quicker to develop, manage, and deploy into programmable network hardware. In addition, it centralises core functions and operates in an "as-a-service" model. We also want to be able to reuse common functionalities, but not have to pay for them multiple times. The process of decomposition also gives micro-applications owners the opportunity to remove redundant functionality from the application logic [18].

The challenge in realising any micro-application-based software using decomposition is to identify the set of functionalities, and this is difficult because it requires domain-specific knowledge. One way to approach decomposition is to leverage domain expertise (e.g., by consulting with domain-specific developers) or to study existing open-source network applications, identifying smaller functional units. As a key for employing functional decomposition, PIaFFE Framework [19] defines a roadmap towards creating an embedded network function (eNF) using an approach based on the application network-related characteristics (e.g., packet parsing, classifying, processing and forwarding), re-architecting the network programming ecosystem.

Using for reference two monolithic network applications, the first one based on a **L2/L3 packet forwarder** and the second one based on **SHA-256 algorithm**, we can outline the steps for decomposing these two applications into a new micro-application, as shown in Figure 1a. The first step for decomposing multiple applications is to identify the network-related characteristics of each one, starting by the **L2/L3 Forwarder**.In the same way, we enumerate the characteristics of the next application, in this case, **SHA-256 algorithm**: here, we use the same colour (blue) to highlight the overlapped characteristics and another colour (green) for specific functions. We also have used a specific colour (orange) to emphasise the main logic behind both applications. Based on the colour map, we can merge the "duplicated" functions of the applications into the **micro-application realisation**, using a single structure representing the two main applications, enabling the portability to a programmable network device.

B. Multi-level forwarding between hardware and software

In order to switch traffic between hardware and software implementations, we employ the Intercepting and Forwarding element (PIIF) from PIaFFE, steering network traffic through the embedded Network Function or sending it up to the application at the virtualisation layer, using a bottom-up approach as shown in Figure 1b: as soon as a packet arrives at the SmartNIC, it is either intercepted and forwarded to the "OS level", or kept into the "hardware level", being processed using the SmartNIC network hardware.

In Listing 1, it is presented a fragment of the P4 source code illustrating the PIIF element utilised in this work. This code defines a basic $(key, value)$ data structure. Whenever an incoming packet hits a table's entry (defined by its source IPv4 and UDP port), then it is steered to the software, otherwise, it is sent to the eNF pipeline that applies a specific hash function.

```
@name(".piif_table")
table piif {
  key = {
    standard_metadata.ingress_port: exact;
    hdr.ipv4.srcAddr: exact;
    hdr.udp.srcPort: exact;
  }
  actions = {
    NoAction;
  }
  size = 512;
  default_action = NoAction;
}
...
apply {
  if (piif.apply().hit) {
    sw_forward.apply();
  } else {
    do_sha256();
    hw_forward.apply();
  }
}
```

Listing 1: Pseudo code of the PIIF

C. Hash algorithm implementation

Our implementation is focusing on porting SHA-256 to a P4 programmable device for evaluating the performance of the basic cryptographic hash operations, which could be applied for a range of use cases further, such as HMAC calculations, hash-based networking applications and others.

P4 language lacks a loop control flow statement, and this limits the possibilities for implementing complex applications, such as SHA-256 which requires that for stages like shuffling and compressing data. To overcome this limitation, we have created an extern based on micro-C language (a variant of C language), as shown in the fragment of code in 2, which can be called from the P4 program as a "function", interacting with the P4 pipeline and its data/metadata.

The target hardware used for the implementation of our architecture (i.e., Netronome SmartNIC), has limited resources for hosting large applications. Thus, the firmware and NIC "storage" memory capacity must be properly used, otherwise, the image can no longer be loaded onto the SmartNIC. To tackle this constraint, we have developed a slim P4 code for L2/L3 packet forwarder based on static table entries.

As mentioned before, the micro-C code fragment 2 starts by exposing the headers and match data from the P4 program, which can be accessed and modified inside the extern, including the packet payload, which is represented by a "payload header" split into blocks of 512-bit size, due to header size restrictions imposed by the used target hardware. The SHA-256 function is called in three steps, in order to accomplish all stages of the algorithm, namely, padding process, shuffling and compression. In the end, the hash is added to a specific header field previously defined in the P4 program. Here is defined by the $sha256\_field$ variable, being transmitted with the original network packet. In this phase of the processing, we are also using this approach in order to measure the time to generate and insert hashes using in-network computing, however, the functionality can be extended to accomplish with more elaborated scenarios in future implementations.

One may even think of a combination of an optimised integration with native dedicated hardware acceleration components. However, crypto security accelerators are not usually accessible nor documented when using P4 language shipped with the NIC's SDKs. Thus, this integration is beyond the scope of this paper.

```
#include "pif_plugin.h"
#include "plugin.h"
...
int pif_plugin_do_sha256(
            EXTRACTED_HEADERS_T *ext_hdrs,
            MATCH_DATA_T *match_data)
{
 PIF_PLUGIN_payload_T *payload;
 BYTE buf[SHA256_BLOCK_SIZE];
 SHA256_CTX ctx;
 /* Grab a pointer to the payload 'header' */
 payload=pif_plugin_hdr_get_payload(ext_hdrs);
 BYTE text[] = payload->block0;
 /* begin sha256 calculation */
 sha256_init(&ctx);
 sha256_update(&ctx, text, strlen(text));
 sha256_final(&ctx, buf);
 /* end sha256 calculation */
 /* storing hash into the specific field */
 payload->sha256_field = buf;
 return PIF_PLUGIN_RETURN_FORWARD;
}
```

Listing 2: A fragment of micro-C extern SHA-256 function.

IV. Evaluation

The performance of the secure hash function, in terms of latency and processing time (processing plus forwarding packets), is critical for high-performance applications. Thus, our evaluation aims to answer the following key question for NFV applications: i) Compared with host-side execution, what are the expected latency savings of forwarding and processing network traffic through an in-network device? ii) What are the latency statistics when running such micro-applications on

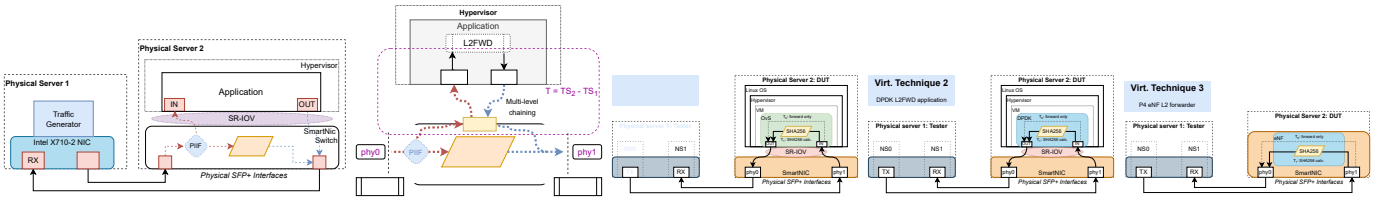


Fig. 2: Testbed specification and virtualisation techniques: a) Prototype implementation and testbed. b) Using hardware timestamp to measure the time to process/deliver a packet. c) Open vSwitch software switch d) Intel DPDK L2 forwarder application e) P4-enabled hardware

different platforms? iii) How close can we get to the maximum possible throughput in terms of packets per second (PPS)?

A. Benchmarking the virtualisation techniques

As far as test scenarios are concerned, we are comparing the in-networking computing capability with network forwarding technologies used on the Linux OS doing the same operation, as follows:

1) **Open vSwitch switch**: modified OvS software switch executed at kernel-mode as a layer 2 network forwarder with a selectable SHA-256 application (Figure 2c).
2) **Intel DPDK**: application acting as a layer 2 forwarder and SHA-256 algorithm which can be turned on/off according to the traffic, as same as previous OvS application (Figure 2d).
3) **P4-enabled hardware**: a Netronome SmartNIC with a layer 2 forwarder eNF, also running a ported SHA-256 application dynamically enabled (Figure 2e).

In all virtualisation techniques, the topology and network functionality are the same in each level (software and hardware levels): packets are generated in one host, being forwarded through the SmartNIC hardware in another host running Linux OS and network drivers in three different modes, respectively: 1) Linux netdev mode, 2) Intel DPDK mode or 3) P4 mode. All details about the testbed will be discussed below.

B. Experimentation Setup

For the benchmark tests, we used the environment described in III and illustrated in Figure 2a, using the technique explained in Figure 2b.

The test consists of generating traffic that goes through each virtualisation technique specified before, measuring latency performance to forward-only and forward-calculate using each specific case. As a proof-of-concept, the Secure Hash Algorithm 256 (SHA-256) library was used. This algorithm is public and open-source, portable for most platforms. We built a library using our previous work, allowing us to check the feasibility of the development and deployment of complex applications using the framework to deliver a micro-application onto a programmable network device.

Hence, UDP packets are generated using pktgen-dpdk through the TX interface and received using the same application in RX. These packets are diverted according to the purpose of each experiment: selecting either the SmartNIC hardware only or software level using a virtual machine. This virtual machine runs a forwarder application with the SHA-256 algorithm inside, that can be enabled or disabled dynamically during the experiments using a different UDP source port in TX, permitting the measurement for each virtualisation technique.

Testbed description: Our testbed consists of two machines connected back-to-back without any switching element in between, as shown in Fig. 2a. One of them hosts the traffic generator, while the other bears the prototype and the Netronome SmartNIC NFP-4000 2x10G. Each machine is equipped with a 1x6-core Intel Xeon E5-2620 v3 2.4Ghz CPU and 2 threads per core (hyper-threading enabled), 8GB memory, and a DPDK compatible Intel X710-2 2x10G Ethernet NIC on the traffic generator side.

To collect the latency results, we use the SmartNIC's internal hardware clock to calculate the timestamps of each packet before it goes out of the SmartNIC to an application and inserts another timestamp when it comes back to the SmartNIC. This gives us more precision to calculate the total cost to run an application inside and outside of the SmartNIC, using a nanosecond scale, as shown in Figure 2b. To collect the throughput (and consequently, the processing capacity) of the target, we employ a Lua script[1] to automatise the experiment and to collect packet data into a comma-separated value (CSV) file, to later be processed and generated the results.

Traffic generator and DUT: *Physical Server 1* runs Intel pktgen-dpdk[2] version 20.04 traffic generator, while *Physical Server 2* acts as a Device Under Test (DUT), using applications developed and deployed into a network device, processing packets using in-network processor or a traditional application running on a qemu-kvm hypervisor, like in Figure 2a. The application runs on their respective qemu-kvm virtual machine and has a dedicated embedded network function from SmartNIC, provided by SR-IOV pass-through on the hypervisor, with 8GB RAM and 2 CPU cores unpinned, in order to create a more realistic environment as possible. The physical servers run Ubuntu Linux Server 20.04, with kernel version 4.15.0-60, as same as the OS running on virtual machines.

[1] https://www.lua.org
[2] http://git.dpdk.org/apps/pktgen-dpdk

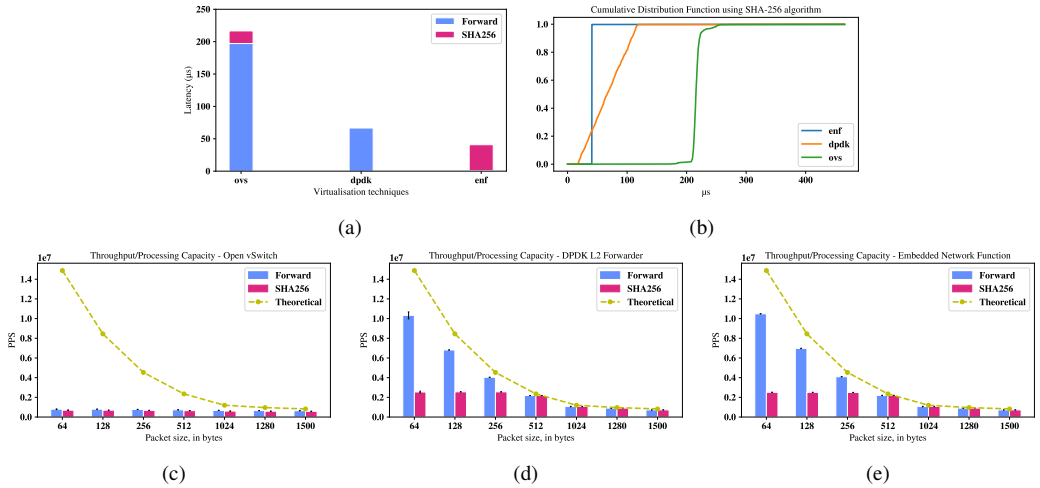


Fig. 3: Latency (processing and forwarding) for software and hardware micro-applications running SHA-256 algorithm and network forwarding: a) Total latency using different types of forwarding/calculation virtualisation techniques; and b) Cumulative Distribution Function (CDF) probability for SHA-256 calculation and forwarding. Packet processing capacity comparison between software and hardware micro-applications running SHA-256 algorithm, using different packet sizes and virtualisation techniques: c) Open vSwitch (OVS) software switch, d) DPDK application and e) eNF with P4 in NIC

C. Results

As we can see, in Figure 3a, the Open vSwitch software switch, despite running on kernel-mode, presents the worst latency level: $\approx$197 μsec to forward only and $\approx$216 μs to calculate the SHA-256 function ($+9.89$ %). DPDK implementation improves it significantly down to around 67 μs in both cases (with and without SHA-256 function), showing that, as might be expected, the Linux kernel bypass method used by DPDK is a key technology for such applications. And finally, using an eNF, we can achieve the best result in terms of latency, reaching 834 nanoseconds to forward a packet (not visible in the chart, due to its small value), but this value increases to ≈ 40.8 μs when we run the SHA-256 function on the hardware, rising at least 40x the time to process and forward a network packet. This highlights the cost of running such a complex application on a resource-limited network programmable device. However, the latency values stay below the best software use-case (DPDK), showing that the network device could be used for offloading the calculation of SHA-256, alleviating the host-side processing.

As far as Cumulative Distribution Function (CDF) is concerned, Figure 3b allows us to see that the eNF has a small variability (almost a deterministic behaviour) when compared with other implementations (Open vSwitch and DPDK) based in software. This is due to the randomness added by the stacks in the OS used to forward the packet from the NIC to the software and then back to the NIC. The DPDK-based application, that enables a shortcut to the user-level applications, the CDF resembles a well-behaved uniform distribute function but with enlarged variability when compared to the OvS implementation. This is important in NFV because latency higher-order moments are important for real-time applications. There may be a trade-off between end-to-end latency and jitter that may work against choosing the DPDK solution.

Figures 3c, 3d and 3e depict the capacity of each virtualisation technique for processing packets using SHA-256 algorithm. In addition, a theoretical line showing the maximum packets per second (PPS) throughput according to the packet size using 10 Gbit/s speed, following the Ethernet frame MTU (64-bytes and 1500-bytes packets) of the experiment in order to check on the maximum and sustained processing rates when forwarding packets through each virtualisation technique.

It is important to highlight that the SmartNIC is not able to achieve line-rate speeds with small packets ($<$512 bytes), as confirmed in a set of benchmarks available on the manufacturer's document library[3]. However, the validity of our comparative results was not affected by this limitation once all experiments were performed using the same setup.

Starting by Open vSwitch switch, we noted that the software switch cannot process and forward small to large packets (Fig. 3c) alike, keeping the packets per second rate below a figure of $800K$, possibly due to the kernel/user modes and the OS stack, being unable to forward nor process packets close to 10 Gbit/s. Meanwhile, DPDK has a more consistent result, showing that the polling mode and CPU resource reservation makes a difference for packet processing/forwarding actions. The eNF results are practically the same as those of the DPDK application, showing that despite the fact that packets are being processed in the network hardware, and consequently avoiding the OS stack and interrupts, the hardware is stressed when calculating and forwarding small packets at near line rate. A tradeoff between cost and performance may lean the choice of the platform toward DPDK solutions when PPS is concerned.

[3]https://www.netronome.com/document-library

Based on the results and low-latency application requirements in need of cryptographic hash functions, we can conclude that Open vSwitch, which is a well-known software switch intensively used by hypervisors, had an unsuitable performance to use as an enabler for low latency applications. It takes more than 210 μs to compute the SHA-256 algorithm and forward a packet. Thus, the time to send the network packets from the "hardware plane" to the "software plane" is quite high using the Open vSwitch switch. It can be reduced when using the DPDK application. Considering that the time to forward a packet on SmartNIC hardware is lower than 1 μs, all values exposed previously basically correspond to the time to switch the context from hardware to software levels. Moreover, comparing each approach, the eNF is the best option to forward a packet, in the same way, to compute a small work on its hardware. In fact, the eNF can complete the SHA-256 hash function, and in sequence forward a packet, using $\approx 60\%$ of the time needed by the best VNF case (DPDK) and $\approx$19 % of the time needed by the worst case (OvS).

V. Conclusion and Future Works

Cryptographic hash functions are essential for assuring the security and privacy of network communication obliterating data from malicious entities, avoiding attacks and establishing mutual or multi-party trust relationships. This paper investigated alternatives for offloading the processing of cryptographic hash algorithms from application-level hosts into in-network computing for SFC in NFV infrastructures.

We described an architecture and prototype implementations integrating secure hashing algorithm 2 (SHA-2) in different virtualisation techniques to benchmark the P4 hardware target platform. Our design of a P4 data plane is able to run a secure hash algorithm using a mechanism for steering traffic between software and hardware levels, enabling the use of complex applications for generating hashes.

Results obtained for hashing using in-network computing demonstrate that it is feasible, and in some cases better than the traditional virtualisation techniques, including DPDK. Applications running on general-purpose CPU can do offloading of its processing to the network device, freeing resources for tenants and improving latency and throughput.

As part of future work, we can cite the use of the SHA-256 algorithm applied with a securitisation solution, such as hash-based message authentication code (HMAC), running on SmartNIC hardware for offloading these applications using the P4 language. Combining the dedicated cryptographic hardware capabilities of the target to P4 applications using *externs* may overcome the limitations imposed by the programming language and improve the portability of other cryptographic hash functions at line-rate speed.

Acknowledgement

Financial support from Science Foundation Ireland grants 14/IA/2527 and 13/RC/2077 is gratefully acknowledged. This study is also supported in part by the following Brazilian agencies: CNPq, RNP, CAPES (Finance Code 001), FAPESP/MCTI/CGI.br (PORVIR-5G #20/05182-3, SAWI #20/05174-0, and SFI, #18/23097-3), FAPES (#94/2017, #281/2019, #515/2021, and #284/2021). CNPq fellows Dr. Martinello #306225/2020-4 and Dr. Ribeiro #315463/2020-1.

References

[1] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker, "P4: Programming protocol-independent packet processors," *SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 3, p. 87–95, Jul. 2014.

[2] X. Jin, X. Li, H. Zhang, R. Soulé, J. Lee, N. Foster, C. Kim, and I. Stoica, "Netcache: Balancing key-value stores with fast in-network caching," in *Proceedings of the 26th Symposium on Operating Systems Principles*, ser. SOSP '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 121–136. [Online]. Available: https://doi.org/10.1145/3132747.3132764

[3] V. Sivaraman, S. Narayana, O. Rottenstreich, S. Muthukrishnan, and J. Rexford, "Heavy-hitter detection entirely in the data plane," in *Proceedings of the Symposium on SDN Research*, ser. SOSR '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 164–176. [Online]. Available: https://doi.org/10.1145/3050220.3063772

[4] M. V. B. da Silva, A. S. Jacobs, R. J. Pfitscher, and L. Z. Granville, "Ideafix: Identifying elephant flows in p4-based ixp networks," in *2018 IEEE Global Communications Conference (GLOBECOM)*. IEEE, 2018, pp. 1–6.

[5] G. Grigoryan, Y. Liu, and M. Kwon, "Iload: In-network load balancing with programmable data plane," in *Proceedings of the 15th International Conference on Emerging Networking EXperiments and Technologies*, ser. CoNEXT '19 Companion. New York, NY, USA: Association for Computing Machinery, 2019, p. 17–19. [Online]. Available: https://doi.org/10.1145/3360468.3366774

[6] J. M. Turner, "The keyed-hash message authentication code (hmac)," *Federal Information Processing Standards Publication*, vol. 198, no. 1, 2008.

[7] NETRONOME, "Netronome Agilio SmartNIC," https://www.netronome.com/products/agilio-cx/, 2019.

[8] R. Sobti and G. Geetha, "Cryptographic hash functions: a review," *International Journal of Computer Science Issues (IJCSI)*, vol. 9, no. 2, p. 461, 2012.

[9] R. Martino and A. Cilardo, "SHA-2 Acceleration Meeting the Needs of Emerging Applications: A Comparative Survey," *IEEE Access*, vol. 8, pp. 28 415–28 436, 2020.

[10] Q. H. Dang *et al.*, "Secure hash standard (shs), standard fips 180-4," *National Institute of Standards and Technology*, 2015.

[11] M. Molina, S. Niccolini, and N. Duffield, "A comparative experimental study of hash functions applied to packet sampling," in *Proc. of International Teletraffic Congress (ITC)*, 2005.

[12] C. Henke, C. Schmoll, and T. Zseby, "Empirical evaluation of hash functions for multipoint measurements," *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 3, pp. 39–50, 2008.

[13] D. Scholz, A. Oeldemann, F. Geyer, S. Gallenmüller, H. Stubbe, T. Wild, A. Herkersdorf, and G. Carle, "Cryptographic hashing in p4 data planes," in *2019 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*, 2019, pp. 1–6.

[14] F. Hauser, M. Schmidt, M. Häberle, and M. Menth, "P4-macsec: Dynamic topology monitoring and data layer protection with macsec in p4-based sdn," *IEEE Access*, vol. 8, pp. 58 845–58 858, 2020.

[15] F. Hauser, M. Häberle, M. Schmidt, and M. Menth, "P4-ipsec: Site-to-site and host-to-site vpn with ipsec in p4-based sdn," *IEEE Access*, vol. 8, pp. 139 567–139 586, 2020.

[16] B. Pfaff, J. Pettit, and T. e. a. Koponen, "The design and implementation of open vswitch," in *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'15. USA: USENIX Association, 2015, p. 117–130.

[17] L. Foundation, "Data plane development kit (DPDK)," 2015. [Online]. Available: http://www.dpdk.org

[18] C. Richardson, *Microservices patterns: with examples in Java*. Simon and Schuster, 2018.

[19] D. R. Mafioletti *et al.*, "Piaffe: A place-as-you-go in-network framework for flexible embedding of vnfs," in *ICC 2020 - 2020 IEEE International Conference on Communications (ICC)*, 2020, pp. 1–6.