Received: 14 September 2019 | Revised: 20 December 2019 | Accepted: 16 January 2020

DOI: 10.1002/spe.2800

RESEARCH ARTICLE

WILEY

A real-time attack defense framework for 5G network slicing

Michel Bonfim[1] | **Marcelo Santos**[2] | **Kelvin Dias**[1] | **Stênio Fernandes**[1]

[1]Centro de Informática (CIn), Universidade Federal de Pernambuco (UFPE), Recife-PE, Brazil

[2]Instituto Federal de Educação, Ciência e Tecnologia do Sertão Pernambucano (IF Sertão-PE), Salgueiro-PE, Brazil

Correspondence
Michel Bonfim, Centro de Informática (CIn), Universidade Federal de Pernambuco (UFPE), Av. Jornalista Aníbal Fernandes, s/n, Cidade Universitária (Campus Recife), 50.740-560, Recife-PE, Brazil.
Email: msb6@cin.ufpe.br

Summary

Network Slicing (NS) is a key enabler to support 5G network services on-demand. However, since NS is a result of the recent advancement in Software-Defined Networking and Network Function Virtualization, it introduces new security issues which include attacks against an NS instance within an operator network and interslice security threats. In this scenario, identifying and mitigating attacks in real-time is of paramount importance to improve security aspects. However, it is far from being straightforward. Therefore, this work proposes the FrameRTP4, a P4-based framework that aims to deliver real-time attack detection and mitigation mechanisms in 5G NS scenarios. For this, it provides a P4-based switch that implements an Service Function Chaining protocol layer, an efficient and scalable Access Control List for the detection and mitigation of known attacks, and a monitoring system aiming to reduce the overhead induced on the control channel. Furthermore, it delivers an orchestrator that aims to control all switches in order to enable lifecycle management of NS instances and P4 table rules. Besides, it also performs some autonomous tasks such as the wildcard rules generation and the detection of new threats by using machine learning algorithms. Preliminary results point to the potential benefits of FrameRTP4 to be part of a 5G NS infrastructure.

KEYWORDS

5G, bloom filter, cybersecurity, network function virtualization, network slice, P4

1 | INTRODUCTION

The Fifth Generation (5G) of wireless networks will radically change the everyday life of people and businesses through improved data rate, decreased latency, and increased capacity for consistent Quality of Service (QoS). In this context, both industrial and academic researchers have put research efforts on the architectural components of 5G networks.[1] A 5G infrastructure must provide features that support different types of vertical business such as Automotive (eg, car manufacturers), eHealth (eg, health industry), Energy (eg, power companies), Factories (eg, Internet of Things [IoT] technology providers), Media & Entertainment (eg, content providers).[2] All of those markets encompass different types of use cases (eg, automated driving, robotics for remote surgery, on-site live event experience, etc.) that have their characteristics (eg, data traffic patterns, mobility support, etc.), and requirements (eg, throughput, latency, etc.). An exhaustive list of case studies for 5G can be found everywhere.[3]

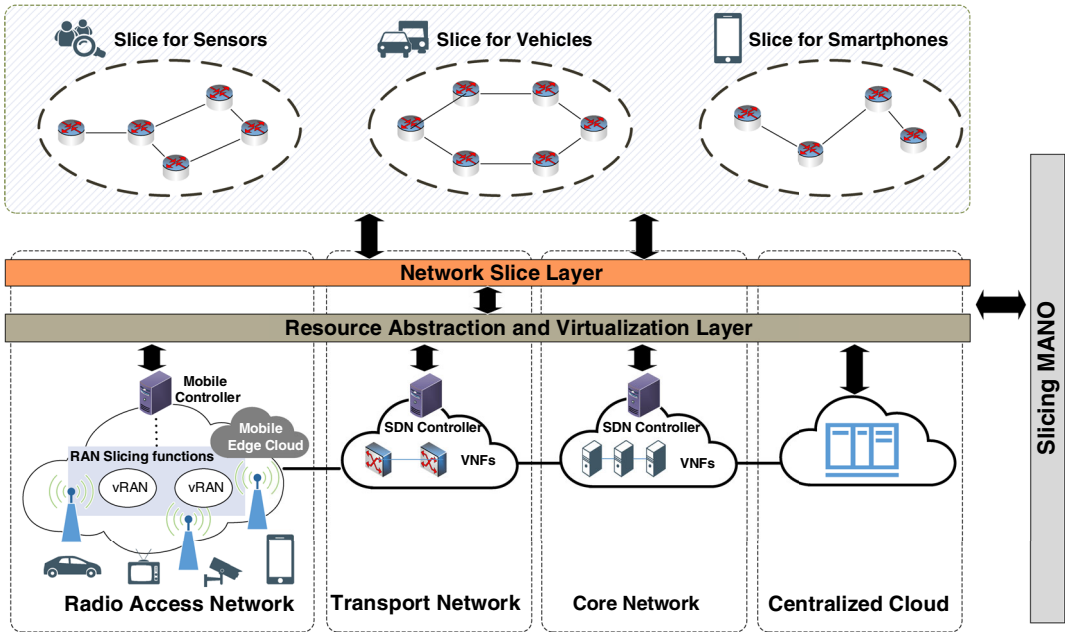


FIGURE 1 Conceptual illustration of network slicing[6] [Colour figure can be viewed at wileyonlinelibrary.com]

Network Slicing (NS) is a key enabler to support the above 5G network services on-demand.[4] It refers to the partitioning of certain physical infrastructure, composed of both network and computational resources, into multiple logical networks called network slices.[5] Each slice is a self-contained network with its virtual resources created on top of the underlying infrastructure. Figure 1 shows examples of slices with different requirements such as autonomous vehicles (very low end-to-end latency) and mobile broadband (high bandwidth).

NS aims at providing efficient resource sharing, traffic differentiation per slice, and management and protection tools.[2] When compared to traditional physical networks, NS has the following advantages:[5] (i) customization of logical networks according to service requirements; (ii) on-demand provisioning to scale resources up or down as conditions change, and (iii) network resource isolation for improved security and reliability. However, since NS is a result of the recent advancement in Software-Defined Networking (SDN) and Network Function Virtualization (NFV), it introduces new security issues, which include attacks against an NS instance within an operator network and interslice security threats.[4] For example, a Distributed Denial of Service attack could degrade performance or even make unavailable a set of slices that share resources on a multitenant virtualized networking infrastructure.[7]

The number and variety of cyberattacks have increased, and their impact can be catastrophic, threatening both network and IT infrastructures around the world. In this context, Distributed Denial of Service (DDoS) attacks, zero-day attacks, and malicious software gained momentum in recent years. Once a violation is detected, its impact on the reputation and market value of a company is almost immeasurable. For instance, the financial consequence of a cyberattack could be close to US$ 11.7 million for companies.[8] Most of the attacks these days are DDoS attacks, coming from billions of vulnerable IoT devices deployed worldwide. In this context, most powerful botnets today are IoT-based botnets.[9]

Furthermore, current security systems such as Security Information and Event Management have not proved to be efficient since 85% of attacks are detected weeks after their occurrences.[10] Therefore, identifying and mitigating attacks in real-time is then of paramount importance to improve security aspects. However, it is far from being straightforward.[11]

In this scenario, some authors argue that SDN provides ideal support for network security appliances (eg, setup of countermeasures against DDoS)[12-15] due to its flexibility and network device programmability, resulting in an exciting means to enforce security policies. OpenFlow is the most popular and the most widely deployed southbound standard for SDN.[6] However, solutions based on OpenFlow are limited to the protocols supported by the technology as well as the lack of flexibility in the implementation of the actions for packet processing such as complex monitoring functions (eg, sketch-based algorithms). In this context, the programmable data plane concept emerges as a promising alternative. It goes beyond the capability of an OpenFlow-enabled switch by providing flexibility to deploy new network protocols and complex in-switch packet processing algorithms. For this, programmable switches allow an operator to define which

packet headers should be parsed and manipulated in the forwarding tables and to introduce new processing actions in the operation.[16]

According to Bosshart et al,[17] to provide a mechanism for the detection and mitigation of attacks in real-time, the solution must operate at line rate on high-speed links. Besides, this processing is constrained to a small-time budget (dozens of nanoseconds) and a limited memory space (eg, ≈50 MB Static Random Access Memory [SRAM] and ≈5 MB Ternary Content-Addressable Memories [TCAM]). In this context, we advocate that such kind of solution must be performed directly inside the switches to support line-rate processing. Moreover, it must cope with the broad masses of data collected in the network, at a fast speed and taking into account the significant variability of Internet traffic.

Taking into account the use of a programmable data plane to address the issues mentioned above, we argue an envisioned solution should support the following mechanisms. First, it should provide an efficient and scalable Access Control List (ACL) for the detection and mitigation of known attacks. It is worth noting that managing flows individually increases the flow processing delay and the likelihood of buffer size overload. To solve this issue, we identify two approaches that provide quick reaction time through the efficient and scalable use of the limited size of flow tables: Bloom Filters (BFs) and Wildcard Rules. Applying BFs for attack detection seems to be a promising solution since it has a good space and time complexity.[18] BF is a probabilistic data structure used for membership query, that is, to check the presence of an element in a set by returning true or false. In this scenario, an operator can store information of all malicious flows in a highly scalable BF for further attack detection performed by filtering mechanisms. However, BF introduces some limitations. For example, it does not support deleting operations, an essential process for flow management. Besides, the use of BFs makes the controller dependent on the target since the latter must have the knowledge and implement the same hash functions of the switches. On the other hand, the use of Wildcard Rules as a solution for attack detection does not suffer from the problems above. Wildcard Rules aim to reduce the number of rules required to realize policies on a single switch. Such an approach enables us to handle flows as groups (ie, rules matching a broad set of flows) by creating patterns with wildcards instead of working with exact match rules.[19]

Second, a mechanism for the detection and mitigation of attacks in real-time should provide an efficient and scalable monitoring system aiming to reduce the overhead induced on the control channel by the communications between a switch and its controller. Such a system aims to track network flows to aid in the detection of new attacks. As a starting point, we argue that, in NS monitoring tasks, exact results are usually not necessary, and a high-quality approximation is enough. This assumption suggests the use of probabilistic data structures (eg, BFs, Sketches) that use smaller amounts of memory and require less computation to achieve the desired goals.[20] Such techniques have been widely accepted in network measurements, thanks to their higher accuracy compared to sampling techniques and their speed.[21] Finally, the detection of new attacks must be performed offline by Machine Learning (ML) Algorithms to take into account the high variability of Internet traffic.

Therefore, this work proposes the FrameRTP4, a framework that aims to deliver real-time attack detection and mitigation mechanisms in 5G NS scenarios. To this end, it follows the SDN architecture by separating the solutions to the problem into the data and control planes. Concerning the data plane, FrameRTP4 makes use of a promising data plane programming technology, namely P4,[16] for line rate processing (directly inside the switches). For this, it provides a customizable P4 program that includes a Service Function Chaining (SFC) implementation to enable the lifecycle management of NS instances. Furthermore, this program consists of an efficient and scalable P4 table-based ACL that applies wildcard rules for the detection and mitigation of known attacks. Besides, the P4 program deploys a monitoring system, namely SFCMon, one of our previous work[22] that uses BFs and sketches to support an efficient and scalable mechanism to track network flows.

On the other hand, in the control plane, the FrameRTP4 provides a Python-based controller that implements customizable modules and a Northbound Application Programming Interface (API). It enables the lifecycle management of both SFCs (ie, NS instances) and wildcard rules (adding and deletion) within the P4 table-based ACLs. Besides, it provides a customizable decision-making module that periodically collects statistical data from SFCMon on all SFC-enabled Switches. It extracts and delivers a feature set to an ML algorithm to automatize the detection of new threats. In this context, once a new attack is detected, the module triggers a request for the creation of a new access control rules within the switches.

In summary, our main contributions are as follows:

1. An *P4-based SFC reference implementation* that aims to enable low-level lifecycle management of NS instances. To this end, we follow the RFC 7665[23] to implement the two SFC components as P4 tables: Classifier and Service Function Forwarder (SFF). Besides, we follow the RFC 8300[24] to define a P4 header for the Network Service Header (NSH).

2. An efficient and scalable monitoring solution, namely *SFCMon*, a P4 program to keep track network flows in SFC environments. For this, we adapt a reference implementation that was proposed in one of our previous works.[22]
3. An *efficient and scalable P4-based ACL implementation* that applies wildcard rules for the real-time detection and mitigation of known attacks.
4. The *FrameRTP4 Switch*, a P4 program reference implementation that provides a processing pipeline that integrates the three components above.
5. The *FrameRTP4 Controller*, a P4Runtime-based SDN controller reference implementation that enables flow control and SFCMon management on one or more FrameRTP4 Switches.
6. A *Wildcard Rule Generator solution* that implements an algorithm for 5-tuple rules compression that aims to reduce the number of rules required to realize policies on a single FrameRTP4 Switch.
7. The *FrameRTP4 Orchestrator*, a Python-based program that controls one or more FrameRTP4 Controllers to provide security for end-to-end NS. To this end, it enables an operator or a third-party management system to perform the life-cycle management of SFCs (ie, NS instances) and P4 table rules (adding and deletion). Besides, it also performs some autonomous tasks such as the wildcard rules generation and the detection of new threats by using ML algorithms.
8. The validation and performance evaluation processes that point to the potential benefits of FrameRTP4. In this context, we have shown that SFCMon could be used as a FrameRTP4 switch component to keep track of malicious flows. Besides, we have demonstrated that a 5-tuple rules compression algorithm, despite time constraints, can be used as a tool for reducing ACL rules on switches.

The remainder of the article is organized as follows. Section 2 describes the background concepts. Section 3 presents the literature review. Section 4 describes the FrameRTP4 architecture, its components, and the details of its implementation. Section 5 describes the experiments performed to evaluate the FrameRTP4. Section 6 describes some threats to the validity of this study to evaluate the quality of this research. Finally, Section 7 concludes the article.

2 | BACKGROUND

2.1 | SDNs and security issues

SDN is a network paradigm that was designed to overcome the difficulty in developing and testing new solutions and protocols in production environments. In this scenario, the underlying code running in business switches and routers are proprietary and closed.[25]

According to Kreutz,[26] currently, both control and data planes are integrated into most commercial networking devices, which makes IP networks challenging to manage. In this context, operators need to configure network policies into each device individually, often using low-level commands that are specific to the manufacturer. Further, automatic reconfiguration mechanisms, necessary for network adaptation during failures and load changes, are nonexistent in today's networks. Such issues reduce the flexibility for deploying new network services and management strategies, as well as hindering development and innovation.

The main feature of the SDN paradigm is the separation of the control and data planes. It has clear advantages where network programmability is achieved through the centralization of the control plane in conjunction with the availability of open APIs, thus making easier the process of creating and deploying new network applications. SDN provides simplification and flexibility in network policy enforcement, facilitating network configuration, and management.[26]

The control plane, represented by a software called the SDN Controller, is responsible for decisions on how to handle network traffic, assuming the role of the "brain" of the network. The SDN Controller can run on Common-Off-The-Shelf platforms, separated from the network equipment. The data plane, represented by the network devices, is responsible for forwarding traffic according to a set of rules.[26] Such rules are created and managed by the SDN Controller. The SDN controller has a global view of the network topology and has direct control over the data plane elements through a southbound protocol, such as OpenFlow.[27]

Figure 2 shows the given three layers of an SDN architecture and the APIs responsible for the interaction between them. The SDN Northbound API is responsible for providing support for communication between the application layer and the control plane layer. It also includes support for SDN Applications, such as traffic engineering (TE), routing, firewall, QoS, etc. The Southbound API is responsible for the communication between the SDN Controllers and switches.

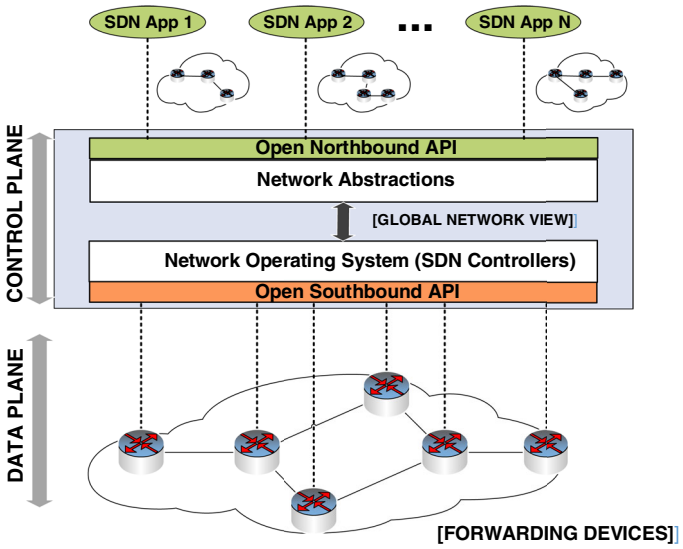


FIGURE 2 SDN reference architecture[25-27] [Colour figure can be viewed at wileyonlinelibrary.com]

Regarding security services, there are clear advantages to be gained from the SDN architecture concerning all security functions (eg, detection, prevention, mitigation).[28] For example:

- *Detection*: the security services (eg, anomaly detection and intrusion detection) in the network can trigger alerts so that the underlying traffic flow could be transferred to the centralized controller, and further be analyzed to provide feedback to the administrator.
- *Prevention*: by using the Controller's Northbound API, the administrator can create or update security policies across the network to prevent attackers or malicious code.
- *Mitigation*: based on the programmability and flexibility of SDN along with the complete network view, it is possible to provide fast control and containment of security threats. In this context, the following countermeasures can be efficiently deployed: traffic isolation, port blocking, packet dropping, rate-limiting (eg, for DoS attacks on the control plane), flow aggregation (multiple flows under a single flow rule), and shorter timeouts to reduce the impact of an attack.

In this sense, the resulting decoupling of the control and data plane can potentially simplify network monitoring, fault tolerance, and security policy enforcement. However, SDN introduces new security issues in the network. Some of the issues are listed below:

- *Unauthorized Access*: unauthorized controller access/controller hijacking, unauthorized/unauthenticated application.
- *Data Leakage*: flow rule discovery (side channel attack on input buffer), credentials capture (keys, certificates for each logical network), and forwarding policy discovery (by using packet processing time analysis).
- *Data Modification*: flow rule changes to modify packets (Man-in-the-Middle attack).
- *Malicious/Compromised Applications*: a malicious or poorly designed/buggy applications could unintentionally introduce vulnerabilities to the system.
- *Denial of Service*: controller-switch communication flood, switch flow table flooding, and triggerable CPU intensive operations.
- *Configuration Issues*: policy conflicts.

Among these security problems, the most relevant has been the controller-switch communication flood, a kind of Denial of Service (DoS) attack that exploits poor scalability of switch-to-controller communication.[28,29] Since traditional SDN solutions (eg, OpenFlow) implement static flow tables, any stateful control, and processing intelligence is necessarily delegated to the network controller, increasing latency and signaling overhead. In this case, the whole scenario may become a show stopper for applications that rely only on local flow/port states and which need to react at the packet-level

time scale (wire-speed). This problem is particularly real for network applications running in real-time, such as traffic management and security services. The latter is our focus in this article.

Therefore, to improve the security, scalability, and performance of real-time network applications, a solution must be performed directly inside the switches to support line-rate processing. Besides, it must cope with the broad masses of data collected in the network, at a fast speed and taking into account the high variability of Internet traffic. Besides, we need to reduce the interaction between the switches and the SDN controller. It is worth mentioning that the real-time network applications are constrained to a small-time budget (dozens of nanoseconds) and a limited memory space (eg, ≈50 MB SRAM and ≈5 MB TCAM).[17] In this scenario, traditional SDN solutions (eg, OpenFlow) are not recommended as they require the intervention of the SDN controller for all forwarding procedures based on network state or profile variation, leading to scalability and latency issues in the data plane.[30] Facing this problem, we argue that the new concept of Programmable Data Plane can help us achieve this goal.

2.2 | Programmable data plane

Data plane programmability has attracted attention from both academia and industry. For example, there are several discussions within Open Networking Foundation geared toward the definition of programmable switches already in the OpenFlow version 1.6. This technology is an evolution of the SDN paradigm. It aims to solve some of the traditional OpenFlow limitations. For instance, solutions based on OpenFlow are limited to the protocols supported by the technology as well as the lack of flexibility in the implementation of the actions for packet processing such as complex monitoring functions (eg, sketch-based algorithms). The programmable data plane goes beyond the capability of an OpenFlow-enabled switch by providing flexibility to deploy new network protocols and complex in-switch packet processing algorithms. For this, programmable switches allow an operator to define which packet headers should be parsed and manipulated in the forwarding tables and to introduce new processing actions in the operation.[16] Pioneering research in this field has focused on the following key technical aspects:

- Identification of new programmable switch architectures. Examples: SDPA,[31] FAST,[32] and OpenState.[33]
- High-level programming languages and compilers which could exploit programmable data planes. Examples: P4,[16] Domino,[34] Stateful NetKAT,[35] and SNAP.[36]

2.3 | Wildcard rules

An SDN environment consists of SDN switches and at least one SDN controller. There is a clear separation between the control plane and the data plane. In this context, SDN enables the creation of programmable networks at the same time that it is possible to deploy applications on the controller according to a global view of the network. SDN brings more efficiency and flexibility in the management of flows.

Despite all the benefits, there is a problem due to the size limit of TCAM. Classical Content-addressable memory (CAM) only perform binary operations.[37] In other words, due to the complexity of SDN rules, it is necessary to use TCAM to store routing tables in SDN switches, but there are some drawbacks compared to CAM: (1) higher cost, (2) higher power consumption, and (3) bigger physical size.[37] Consequently, the amount of TCAM available in each SDN forwarding device is limited. In general, an OpenFlow switch can only handle between 10K and 40K flow entries.[38]

One of the strategies to handle the TCAM memory space problem is to apply a flow rule compression algorithm. The objective is to merge multiples flow entries into one without loosing forward rule precision. The use of wildcards and binary numbers allows greater efficiency in the rule compression process. An example of dynamic aggregation traffic is shown in Figure 3. In this case, there is a reduction of about 77% in the flow table.

2.4 | Monitoring with probabilistic data structures

Probabilistic data structures provide a deterministic number of operations but probabilistic output. These structures use smaller amounts of memory and require less computation to provide a high-quality approximation of the exact results.[20]

FIGURE 3 Compression approach by wildcard

#	Decimal Number	Binary number	Aggregated
1	3	00011	00011
2	18	10010	1**1*
3	19	10011	
4	22	10110	
5	23	10111	
6	26	11010	
7	27	11011	
8	30	11110	
9	31	11111	

Probabilistic data structures such as BFs[39] and Sketches[40] have been widely accepted in network measurements, thanks to their higher accuracy compared to sampling techniques and their speed.[21] *BF* is a probabilistic data structure that provides a memory-efficient approach for insertions and membership queries.[39,41] A BF consists of a 1-bit vector of M cells. For insertion, an element is used as input in K hash functions, which map to different cells in the array. Then, each position is marked with bit 1. For membership queries, an element is used as input in the same K hash functions. An element is considered in the set if all the hash values map to a cell with bit 1.

In BFs, the False Negative Rate is always 0; that is, if at least one hash value map to a cell with bit 0, the probability of the element not being part of the set is 100%. However, false-positive matches are possible, that is, if all the hash values map to a cell with bit 1, the element is possibly part of the set. Fortunately, the False Positive Rate (FPR) is controllable and can be defined at design time by Equation (1), considering a design with N elements, M cells, and K hash functions. In practice, BF designers focus on the tradeoff among memory, number of operations, and FPR.

$$\text{FPR} = \left(1 - \left(1 - \frac{1}{M}\right)^{K \times N}\right)^{K}. \tag{1}$$

According to Reference 42, a good approximation to obtain lower FPR is to assume $M > K \times N$. In network measurement, one BF can be used within a switch to detect, rapidly and memory-efficiently, whether or not a flow is part of a set of monitored flows.[42] However, it cannot store data from these flows.

For instance, *Invertible Bloom Lookup Table (IBLT)* is an extension of BF that allows the storage of key-value pairs.[43] In addition to the insertions and membership queries, IBLT also allows for updates, deletions, lookups, and a complete listing. To this end, each IBLT cell contains three fields:

- *count*: it stores the number of entries mapped to this cell;
- *keySum*: it stores the sum of all the keys mapped to this cell;
- *valueSum*: it stores the sum of all the values mapped to this cell.

The listing method allows listing all the key-value pairs being stored in one IBLT (please refer to Reference 43 for more details). Therefore, using this method, some managers can retrieve a set of IBLTs and list their respective entries, with a certain probability of failure (only part of the entries are extracted successfully). In this context, one IBLT can be used to keep track flows as well as per-flow counters (metrics). However, some issues must be solved.

First, for each insertion, deletion, or update operation on an IBLT, an algorithm must verify whether or not the input belongs to the set (lookup) beforehand to avoid inconsistencies. Besides, the design of the size of the IBLT will generally be determined by the desired probability for a successful lookup operation that follows Equation (1).[43] Since an IBLT takes up more memory than a BF, such a restriction may render the use of this structure impracticable on low-memory devices. To reduce memory consumption, we can use an approach that applies a combination of a BF and an IBLT. For instance, the lookup operation will now be performed using the BF and the remaining operations in the IBLT. By using this approach, we can now target the probability of succeeding in listing entries that follow the theorem below.

Theorem 1. *As long as the number of cells M is chosen so that $M > (C_K + \epsilon) \times T$ for some $\epsilon > 0$, the listing method fails with probability $\mathcal{O}(T^{2-K})$ whenever $N \leq T$, being N the number of elements and K the number of hash functions.*

It worth noting that C_K is a constant whose value is determined by K and is always less than K. For example, if $K = 7$ then $C_K = 1.721$. Thus, the listing method is the key feature that enables us to design lower-memory IBLTs.

Second, even working with lower-memory IBLT, a large number of flows to manage can also make it unfeasible. For example, consider a dataset trace from a 10 GB/second ISP backbone link, recorded in 2016, and available from CAIDA*. This trace is 17 minutes long, and each 20-second chunk contains, on average, about 400 000 5-tuple flows. According to the theorem, an IBLT must be designed with approximately 518 402 cells to meet this load of flows, where $K = 4$. If we consider that *count* and *valueSum* have four bytes each and *keySum* has 13 bytes (5-tuple size), we will have a total of 21 bytes per cell and an IBLT that will occupy about 11 MB. We argue that this value is impractical, considering that the switch also needs to support other functionalities related to packet forwarding and access control. Besides, according to Reference 44, switches should work with a limited amount of memory available (about 1.4 MB) per stage to maintain line-rate processing (at 10-100 GB/second).

One solution to this problem is to keep track of only the large/heavy flows. A large flow is a flow that includes more than a certain percentage of the packets during a given time interval.[45] Studies have shown that the majority of flows tend to be short (mice flows) on high-speed links. In Reference 46, the authors showed that a 5% to 10% of flows by number account 60% to 80% of traffic by volume. According to Reference 47, both traffic monitoring and attack detection can benefit from accurately measuring only the large flows. Thus, we argue that accurately monitoring all flows in high-speed networks is infeasible. However, it is sufficient to keep track of only large flows to meet the advanced management capabilities required in SFC environments.

In this context, we can use *Count-Min Sketches (CMS)* to detect large flows. CMS is probabilistic data structure whose goal is to consume a stream of events (eg, packets) and count the frequency of different types of events. Theses frequencies can be queried at any time.[40] CMS is a two-dimensional vector of w columns and d rows. Given (ϵ, δ), w and d are defined at design time by setting $w = \lceil \frac{e}{\epsilon} \rceil$ and $d = \left\lceil \ln\left(\frac{1}{\delta}\right) \right\rceil$. Each row has a hash function associated $H = (h_1, \ldots, h_d)$, chosen from a pairwise-independent family. In turn, each hash function maps to one of the w columns.

A basic CMS has two methods: *Update* and *Estimate*. Once a new event i arrives, for each row j, the *Update* method applies the associated hash function to obtain the column index $h_j(i)$. Then, it increments the index by one. On the other hand, the *Estimate* method returns the estimated frequency of an specific event, that is, $\hat{f}_i = \min_{1 \le j \le d}(\text{CMS}[j, h_j(i)])$. This estimate ensures that $\hat{f}_i \le f_i + (\epsilon N)$ with probability $1 - \delta$, where f_i is the true frequency and $N = \sum_{i=0}^{n} f_i$ is the stream size. Note, CMS occasionally overestimates frequencies, but it never underestimates frequencies.

Therefore, we can use CMS to detect large flows. We can formalize this by taking the CMS to solve the *ϵ-approximate heavy hitters (HH) problem*.[48] In this problem, the input is a stream of length T and the user-defined parameters k (where $T \gg k$) and ϵ (error tolerance). As an output of a single pass over the stream, we want a list L of elements such that:

- x is in L if x occurs at least $\frac{T}{k}$ times in the stream;
- Every value in L occurs at least $\frac{T}{k} - \epsilon \times T$ times in the stream.

According to Reference 48, a good approximation to detect HH with high probability is to assume $\epsilon = \frac{1}{2 \times k}$.

3 | RELATED WORK

3.1 | The use of programmable data planes for network security

The number and variety of cyberattacks have increased, and their impact can be catastrophic, threatening both network and IT infrastructures around the world. In this context, DDoS attacks, zero-day attacks, and malicious software gained momentum in recent years. Once a violation is detected, its impact on the reputation and market value of a company is almost immeasurable. Most of the attacks these days are DDoS attacks, coming from billions of vulnerable IoT devices deployed worldwide. In this context, most powerful botnets today are IoT-based botnets.[9]

Many proposals for improving security aspects in NS have been studied. Emergent technologies like Blockchain plays a significant role in NS security.[49] Cloud Computing has also been used for detecting attacks.[50] In this scenario, we address the issue of cloud scalability to mitigate the growing load presented by attacks, as the popular DDoS. Specifically, the

*http://www.caida.org/data/passive/passive_2016_dataset.xml

TABLE 1 Programmable data plane security solutions

Reference	Use case	Platforms
Bonola et al[53]	Dynamic Traffic Analysis	StreaMon
Guan and Shen[54]	Dynamic Traffic Analysis	P4
Ding et al[55]	Dynamic Traffic Analysis	P4
Bianchi et al[33]	Traffic Classification (DPI)	OpenState
Sanvito et al[56]	Traffic Classification (DPI)	OPP
Bianco et al[57]	Traffic Classification (DPI)	OpenState
Capone et al[58]	Failure Detection and Recovery	OpenState
Carmelo et al[59]	Failure Detection and Recovery	OpenState, P4
Cascone et al[60]	Failure Detection and Recovery	Not Available
Arashloo et al[36]	DoS Detection (UDP Flooding Stateful Mitigation)	SNAP
Boite et al[61]	DoS Detection (Anomaly Detection and Mitigation)	OpenState
Afek et al[62]	DoS Detection (Spoofed SYN Flood Attack)	P4
Febro et al[63]	DoS Detection (Flood of SIP INVITE packets)	P4
Paolucci et al[30]	Traffic Classification (DPI) and DoS Detection (SYN Flood Attack)	P4
Lapolli et al[64]	DoS Detection (Anomaly Detection)	P4

Abbreviations: DoS, denial of service; DPI, deep packet inspection.

cloud resources should be expanded or scaled up in the presence of DDoS attacks. Some studies[51,52] present guidelines on how this approach can be performed.

In this work, we argue that programmable data planes provide ideal support for network security appliances (eg, setup of countermeasures against DDoS). In this context, we reviewed some relevant studies that implemented security solutions using programmable switches. Table 1 lists the most important studies we have found.

It is worth noting that the above research studies are very recent (from 2014), and most have been published in relevant journals such as Wiley International Journal of Network Management and conferences such as SIGCOMM, INFOCOM, and IEEE International Conference on Communications. They focused on proving the viability and practicality of the new research towards data plane programmability. Several types of proof-of-concept use cases and application scenarios have been considered. Below, we present a brief description of the techniques used in each of these use cases, as well as their limitations and drawbacks.

3.1.1 | Traffic analysis

Traffic Analysis is a process that performs traffic monitoring in the network to reveal communication patterns, for example, through the understanding of the volume and duration of different services. Solutions for this use case focus on solving the following problem:

- Complexity and dynamic nature of modern cyberthreats: How to promptly react quickly to traffic patterns changes without impacting the overall system performance?

Bonola et al[53] proposed a data-plane programming abstraction that allows the creation of monitoring tasks performed within the switches. The authors suggested an implementation that enables the use of state tables and flows directly running over network devices, allowing them to associate different metrics and features to be monitored for each flow/state. By using an XML-based language, a programmer can define his/her monitoring operations by defining states, configuring metrics and determining when and how to extract features, and describing the conditions to trigger actions.

Guan and Shen[54] implemented an efficient network monitoring framework, named FlowSpy. The main goal is to assign monitoring tasks to switches for balancing the load of traffic monitoring, thus reducing bandwidth consumption

between data plane and control plane. FlowSpy uses a P4 program to provide flexibility in creating monitoring tasks. For example, one task can determine the monitoring of packets from source IP address 192.168.1.1 to destination IP address 192.168.2.2, and the triggering of an alarm when the number of packets exceeds a predefined threshold. Once collected, the statistical data are forwarded to the FlowSpy application for further analysis.

Finally, for management purposes, network operators need to keep track of changes in network flow distribution to ensure that it behaves as intended. In this context, the most widely-used metric is entropy.[65] The estimation of network traffic entropy helps detect and diagnose performance and security issues by supporting management tasks such as congestion control[66] and DDoS attack detection.[64] In this context, Ding et al[55] proposed the P4Entropy, a P4 program that aims to reduce bandwidth consumption between data plane and control plane by estimating traffic entropy entirely in the switch. To this end, the authors overcome the limitations of P4 language by enabling the calculation of logarithm and division operations to provide a network traffic entropy estimation strategy that relies on Shannon entropy.[67] The P4Entropy can be used to allow for the execution of monitoring tasks requiring the computation of such functions.

3.1.2 | Traffic classification

Traffic classifiers comprise the set of techniques used to characterize traffic flows to differentiate the service types. Thus, each flow can be treated differently, enabling key application uses such as TE and profiling, QoS enforcing, and security. A common approach adopted for traffic classification is based on Deep Packet Inspection (DPI). Solutions for this use case focus on solving the following problem:

- How to reduce the required computing power of the DPI, as well as the network bandwidth between the switches and the DPI (as an SDN Application)?

The four solutions found[30,33,56,57] have used similar approaches to deal with the above problem. They exploit the data plane programmability to offload down to the network the process of filtering and counting traffic, previously performed by the DPI application on the SDN Controller. Thus, only a small sequence of packets of the same flow is filtered, counted, and sent to the controller to perform the traffic classification.

In the most recent work, Paolucci et al,[30] proposed a new node architecture with a P4 switch for an edge multilayer packet over optical node. This P4-enabled node provides native support of a DPI solution and two P4-based TE solutions for latency-sensitive traffic: traffic offloading and dynamic optical bypass. Moreover, the authors also implemented a P4-based cybersecurity mitigation mechanism that focuses on the detection and reaction of Synchronisation (SYN) flood attacks. Finally, the authors conducted experiments to validate these P4 solutions on both behavioral model P4 switches (BMV2) and the NetFPGA board.

3.1.3 | Failure detection and recovery

Some studies[58-60] apply labels in the packets aiming to alert neighboring devices of possible link failures. It is worth emphasizing that these three articles come from the same research group.

In Capone et al,[58] the authors proposed a controller-independent stateful link/node failure detection and recovery scheme, based on OpenState. This solution tags packets with labels, upon detecting a node or link failure, to communicate with previous nodes to use a detour path for subsequent packets.

3.1.4 | DoS detection

Denial of Service (DoS) attacks has been highlighted as the primary security problem reported by companies around the world.[62] In popular SDN approaches for DoS detection, a controller has to deal entirely with threat protection processes (eg, monitoring, detection, and mitigation), thus risking being overloaded (scalability) and failing (single point of failure) consequently. By using data plane programmability, one can exploit its efficient monitoring capabilities to achieve accurate detection levels. Besides, since anomaly detection techniques also involve local information, they can also be done inside the switch.

Some research studies[36,61,62] proposed solutions for monitoring, detecting, and mitigating DoS attacks, running into network devices. Arashloo et al[36] and Afek et al[62] proposed solutions for detection of specific attacks, whereas Arashloo et al[36] performed a comparison of counters with thresholds to detect different DoS attacks, i.e, SYN flood, User Datagram Protocol (UDP) flood, and Domain Name System (DNS) amplification. The authors highlighted how to perform the implementation using SNAP, but did not perform a complete evaluation. In Afek et al,[62] the authors proposed four different SYN antispoofing methods (Transmission Control Protocol [TCP] Proxy, HTTP redirect, TCP safe reset, and TCP Reset) and one DNS antispoofing method are implemented over SDN using OpenFlow 1.5 and P4. Furthermore, they orchestrate multiple switches as a way to distribute the attack mitigation (coordinated mitigation system).

On the other hand, Boite et al[61] designed on top of OpenState a novel DDoS detection and mitigation strategy named StateSec. To this end, the authors implemented a monitoring process that feeds an entropy-based detection algorithm, both running into the switch and using only local information. Besides, they performed a mitigation process into the controller. Regarding the monitoring tasks, they use one state table for counting each traffic feature (source and destination IPs and ports). For detection, they adopted a simple statistical-based algorithm from the literature.[68] Mitigation tasks allow to drop, queue, prioritize, black hole, or even forward traffic towards an Intrusion Detection System (IDS) (further diagnosis).

Febro et al[63] proposed a P4-based solution, namely TDoSD@DP, to enable early detection and mitigation of Telephony Denial of Service (TDoS) incidents directly within the data plane. TDoSD@DP focuses on protecting the Session Initiation Protocol (SIP) servers against the INVITE flood attack. For this, it tracks the state machine for the SIP protocol, and any deviation from the expected state transition will raise the mitigation mechanisms. In this context, the authors provided a P4-based application that operates at a SIP protocol layer and implements a state-based SIP INVITE flood attack detection and mitigation algorithm that is applied at every port on a network switch.

Finally, Lapolli et al[64] designed a mechanism to perform low-latency, fine-grained traffic inspection for real-time DDoS attack detection. As in Boite et al,[61] the authors implemented a monitoring process that feeds an entropy-based detection algorithm for anomaly detection. All process is performed directly inside the switches. Such a mechanism consists of a P4-based application that provides an in-switch processing pipeline to estimate the entropies of both source and destination IP addresses using Shannon entropy, taking into consideration all incoming packets.

3.1.5 | Problems, gaps, and opportunities

Several security designs and approaches are using the new concept of Programmable Data Plane. However, none of them focused on real-time application constraints and requirements, in our case, attack detection. A mentioned in Section 1, the real-time network applications are constrained to a small-time budget (dozens of nanoseconds) and a limited memory space (eg, ≈50 MB SRAM and ≈5 MB TCAM).[17] Furthermore, according to Sivaraman et al,[44] switches should work with a limited amount of memory available (about 1.4 MB) per stage to maintain line-rate processing (at 10-100 GB/second).

We argue that the emergent concept of Programmable Data Plane can help us achieve this goal. In this scenario, P4 emerges as a new high-level, platform-agnostic (portable implementations) language for SDN. It provides flexibility to decide how to process packets (custom pipelines), what headers a switch should recognize (user-defined protocols) and which actions to perform, thus enabling complex workflows and finite state machines enforcement. Different from OpenFlow, P4 empowers the creation of stateful SDN applications. To this end, P4 language specification provides the programmer with stateful objects such as counters, meters, and registers, which can store states that persist across multiple packets of a single flow (ie, flow states).[29] For example,[59] proposed the OpenState.P4 that deploys the OpenState platform in P4 behavioral model switches. For this, the authors have used an array of registries to implement states and state tables.

Besides, P4 allows the developer to implement its actions using if-else clauses, algebraic and boolean expressions, and different types of counters and hash functions, allowing the calculation of in-switch high-level features and the creation of customizable traffic analysis mechanisms. In this context, P4 presents new opportunities to create real-time network applications, including strategies against attacks.

3.2 | Network flow tracking with probabilistic data structures

We found some studies that focus on the use of probabilistic data structures for performance measurement and volume counting. UnivMon[69] proposed an application to detect HH and use this information to do better traffic management or

monitoring. To this end, UnivMon uses a CMS to perform top-k HH detection. The HashPipe[44] provides an in-switch processing algorithm that uses a pipeline of hash tables to track heavy flows with high accuracy, aiming to evict lighter flows from switch memory over time. Recently, the Elastic Sketch[21] proposed a novel sketch to achieve accurate network measurements no matter how traffic characteristics vary. For this, Elastic Sketch uses a pipeline of hash tables and a technique named Ostracism to keep large flows separated from the mouse flows. However, different from SFCMon, UnivMon, HashPipe, and Elastic Sketch do not save network flow data for further collection by external applications. In contrast, SFCMon uses an IBLT to store flow data as key-value pairs (ie, the flow tuples and the packet and byte counters).

Closest to our efforts, FlowRadar[70] provides a monitoring tool that keeps track flows within the constraints of emerging programmable switches. Similar to SFCMon, FlowRadar uses a BF for flow filtering and an IBLT to store flow data and per-flow counters. However, FlowRadar applies for all transient flows, while SFCMon stores only large flows.

4 | THE FRAMERTP4 FRAMEWORK

FrameRTP4 is a framework that aims to deliver real-time attack detection and mitigation mechanisms in 5G NS scenarios. For this, it seeks to provide security directly within switches by offloading the tools of detection and mitigation of known attacks. Moreover, the process of detecting new attacks is done offline, in the control plane, through the use of ML algorithms. In this scenario, to continue maintaining real-time processing requirements, FrameRTP4 makes use of SFCMon, an efficient and scalable monitoring solution, to aid in the generation of data for the classifier.

We have developed a proof-of-concept prototype of FrameRTP4 in Python. In this section, we introduce the FrameRTP4 architecture and the details of its implementation.

4.1 | FrameRTP4 architecture

Figure 4 shows the FrameRTP4 Architecture. It follows the SDN principles by separating the solutions to the problem into the data and control planes. Concerning the data plane, FrameRTP4 implements a customizable P4 program[16] to

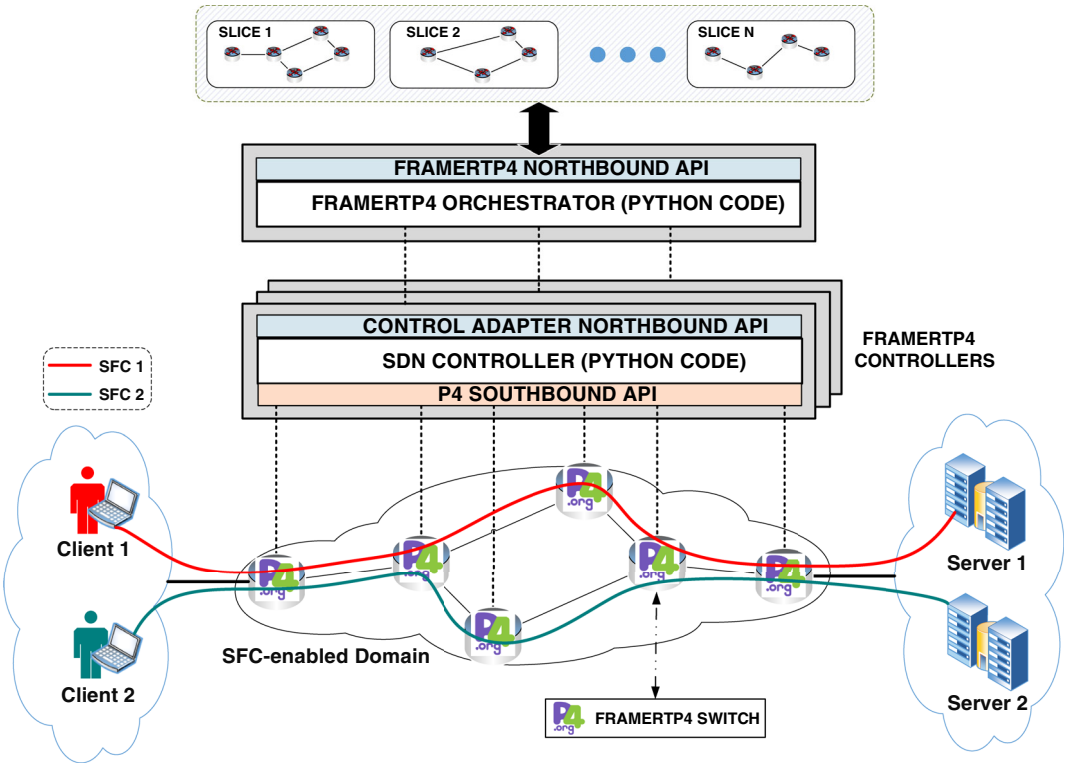


FIGURE 4 FrameRTP4 Architecture [Colour figure can be viewed at wileyonlinelibrary.com]

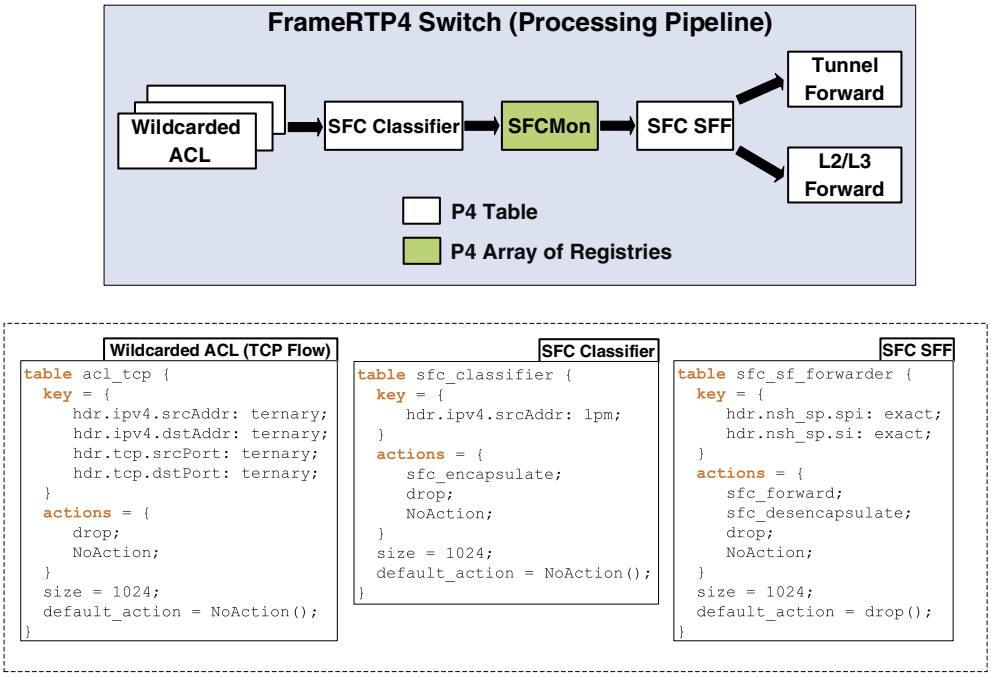


FIGURE 5 FrameRTP4 Switch overview [Colour figure can be viewed at wileyonlinelibrary.com]

provide line-rate processing directly within the switches. To this end, such program provides the following functionalities: (i) an SFC implementation[22] to enable low-level lifecycle management of NS instances; (ii) an efficient and scalable P4 table-based ACL that applies wildcard rules for the detection and mitigation of known attacks; and (iii) an efficient and scalable monitoring solution, namely SFCMon,[22] to keep track network flows in SFC environments.

On the other hand, in the control plane, the FrameRTP4 provides the following components:

- *FrameRTP4 Controller*: Python-based program that manages flow control to the P4 switches by using a Southbound API. FrameRTP4 is designed to support different types of SDN controllers, ie, target-independent. For this, the *Control Adapter Northbound API* consists of an interface defined to ensure the interoperability of different technologies to access the data plane (eg, P4Runtime, Thrift, OpenFlow). In this scenario, each controller type must implement such an interface to be supported by the framework. Our prototype currently supports FrameRTP4 controllers that implement a P4 Runtime API[†] to interact with the data plane components.
- *FrameRTP4 Orchestrator*: Python-based program that contains a collection of "pluggable" modules that enable an operator or a third-party management system to perform the lifecycle management of SFCs (ie, NS instances) and P4 table rules (adding and deletion). To this end, FrameRTP4 Orchestrator provides the *FrameRTP4 Northbound API*. Moreover, the modules mentioned above also performs some autonomous tasks such as the wildcard rules generation and the detection of new threats by using ML algorithms. Finally, it is worth noting that FrameRTP4 Orchestrator can control one or more FrameRTP4 Controllers to provide security for end-to-end NSs.

Below, we provide more details about these components.

4.1.1 | FrameRTP4 switch

In this subsection, we describe the P4 program processing pipeline that dictates the behavior of the FrameRTP4 switch. It is worth mentioning that this program is customizable so that a developer can define new protocols (header definitions), tables, and actions to modify its operations.

Figure 5 illustrates this pipeline that encompasses three components, as mentioned at the beginning of this section.

[†]https://p4.org/api/p4-runtime-putting-the-control-plane-in-charge-of-the-forwarding-plane.html

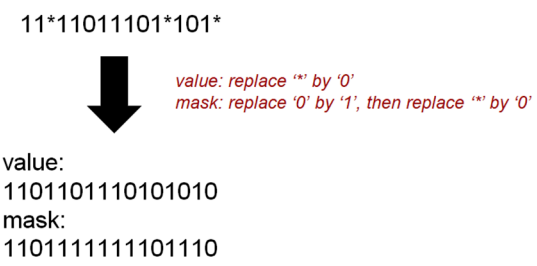


FIGURE 6 Conversion of wildcard values to P4 ternary match format [Colour figure can be viewed at wileyonlinelibrary.com]

The first component is an SFC implementation that aims to enable low-level lifecycle management of NS instances. For this, we adapt a reference implementation that was proposed in one of our previous works.[22] SFC is a key enabler for 5G NS.[71] It is a mechanism that allows an ordered set of network SF, which may or may not be virtual, to be connected to each to form an end-to-end service through multiple data centers, WANs, and application service providers.[23,72] SFCs deal with data traffic, control, and monitoring of a specific application/service, applying appropriate policies along the routes according to the service requirements and considering the availability status of the network. The SFC standardization is addressed by the Internet Engineering Task Force at the SFC working group (WG). Ongoing work at SFC WG focuses on aspects of the architecture (RFC 7665[23]) and protocol (RFC 8300[24]) needed to enable this network capability. SFC is a hot topic and encompasses various problems, including SFC path selection, optimization of SF placement in an SFC-enabled domain, policy enforcement, troubleshooting, and security.

In this scenario, the FrameRTP4 follows the RFC 7665 to implement the two SFC components: Classifier and SFF. It is worth mentioning that those components are sufficing to create SFC-enabled domains. In this context, we implement both Classifier and SFF as P4 tables (see Figure 5). Moreover, we follow the RFC 8300[24] to define a P4 header for the NSH, an SFC encapsulation protocol required to support the SFC architecture. Usually, the connection between SFC components is performed by NSH.

The second component is an efficient and scalable ACL implementation that applies wildcard rules for the real-time detection and mitigation of known attacks. As mentioned in Section 2.3, Wildcards Rules aim to reduce the number of rules required to realize policies on a single switch. Such an approach enables us to handle flows as groups (ie, flow rules that will match a broad set of flows) by creating patterns with wildcards instead of working with exact match rules.[19] Our ACL is implemented as a P4 table (see Figure 5) where, given the flexibility of the P4 language, we can use as filters fields of different types of headers as well as we can define different types of actions that represent various mitigation tasks. Currently, FrameRPT4 enables the creation of ACLs that use the 5-tuple as a filter (source and destination IP addresses, source and destination ports, and protocol type). According to Reference 73, the 5-tuple contains the primary information of network link, and existing monitoring mechanisms usually have excellent efficiency. Also, to avoid problems in field extraction, each protocol number must have its P4-based ACL table. For example, Figure 5 shows the implementation of an ACL table for the TCP protocol.

However, one challenge in this scenario is how to filter wildcard values within table P4. The language specification (P4 core library) provides the programmer with three-match kind types:[29] exact, long-prefix matching, and ternary. Among these, ternary is the only type that allows associating any mask with a field value. In this match type, both the field value and the mask are used as operands of the infix operator "$\triangle$". This infix operator takes two binary numbers of the size W and creates a set of binary numbers of size W. More formally, the set denoted by "$value \triangle mask$" is defined in (2), where:

$$\text{value} \triangle \text{mask} = \{x | \text{value} \wedge \text{mask} = x \wedge \text{mask}\}, \tag{2}$$

where x is of type bit$\langle W \rangle$ and $\wedge$ is the bitwise operator AND.

Therefore, we apply the ternary type in all ACL match fields. Figure 6 shows the algorithm used to convert the wildcard values in table entries. To generate field value, the algorithm takes the wildcard value and replaces all "*" with "0" to create the smallest value of the set. On the other hand, to generate the mask, the algorithm takes the wildcard value, replaces all "0" with "1" (indicates a fixed bit), and then replaces all "*" with "0". It is worth noting that each bit set to "0" in the mask indicates a "don't care" bit.

Finally, the third component consists of an efficient and scalable monitoring solution, namely SFCMon, to keep track network flows in SFC environments. To this end, we adapt a reference implementation that was proposed in one of our previous works.[22] To achieve the desired goals, SFCMon works with a pipeline of probabilistic data structures to detect

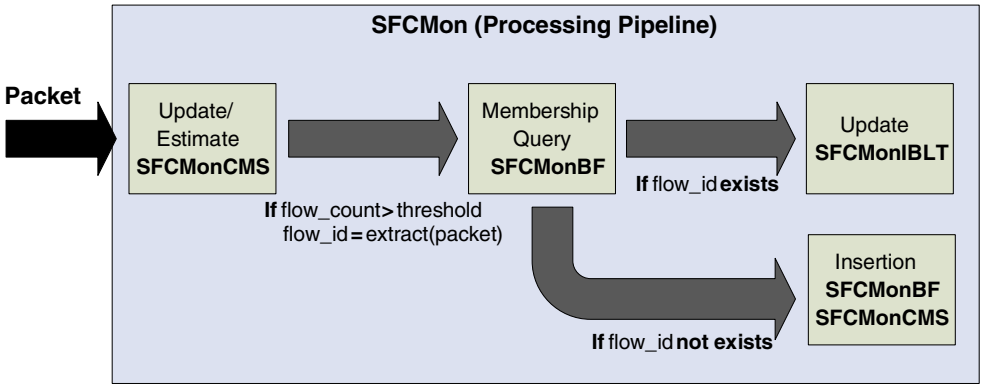


FIGURE 7 SFCMon processing pipeline [Colour figure can be viewed at wileyonlinelibrary.com]

large flows and to store some of their features, such as packet counting. As mentioned in Section 2.4, probabilistic data structures provide a deterministic number of operations but probabilistic output. These structures use smaller amounts of memory and require less computation to provide a high-quality approximation of the exact results.[20]

Figure 7 shows the SFCMon processing pipeline. For instance, the SFCMon pipeline includes the following structures.

- *SFCMonCMS*: an CMS which is responsible for detecting large flows, by solving the *ϵ-approximate HH problem*;
- *SFCMonBF*: an BF is used to reduce the SFCMonIBLT size and is responsible for answering lookup queries (membership) concerning the set of current large flows being tracked;
- *SFCMonIBLT*: an IBLT which is responsible for keeping track all large flows needed to assist in the monitoring tasks.

One flow in the SFCMonIBLT consists of the Service Path Identifier (SPI); which identifies a Service Function Path (SFP); the Service Index (SI), which provides location within the SFP; and the 5-tuple data (source and destination IP addresses and ports, and protocol type). Both SPI and SI allow us to differentiate between the different kinds of instantiated slices. On the other hand, the 5-tuple provides a more fine-grained flow monitoring system. Further details on the operation of SFCMon can be obtained in our previous work.[22]

4.1.2 | FrameRTP4 orchestrator

As mentioned at the beginning of this subsection, the FrameRTP4 Orchestrator is composed of a set of pluggable modules that enable the lifecycle management of both SFCs (ie, NS instances) and wildcard rules (creation and deletion) within the FrameRTP4 switches as well as the automatic detection of new threats.

Figure 8 illustrates those modules and how they are interconnected. Some of them can be easily modified by a developer to reflect new behaviors. We describe them below.

- *Database Management Module*: it serves as an intermediary of the communication between the database and the Core Engine. It provides Core Engine with a systematic way to create, retrieve, update, and manage data. Currently, FrameRTP4 only supports sqlite[‡]-based database, but can be easily extended to other managers (eg, MySQL, PostgreSQL).
- *Rules Management Module*: it comprises an abstraction layer over the Control Adapter Northbound API to facilitate the management of flow rules (eg, creation and deletion) in all defined P4 tables.
- *Flow Stats Collector Module*: it periodically collects statistical data from the *SFCMonIBLTs* on all FrameRTP4 Switches. The objective is to keep track of flow records (flow ids, packet, and byte counters). Besides, such records are made available for the Decision-Making Module. Furthermore, it periodically resets all SFCMon data structures on all switches. As seen in our previous work,[22] the desired goal is to remove flows that are no longer active and to keep the false positive probability low.

[‡]https://www.sqlite.org/index.html

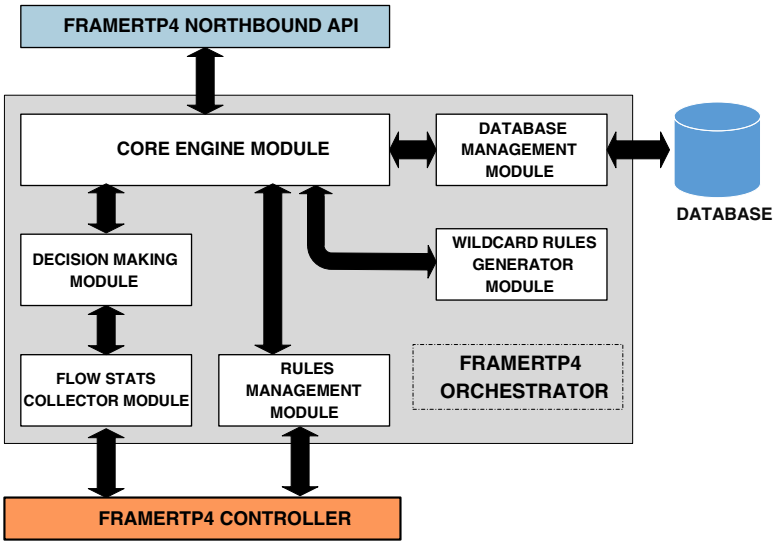


FIGURE 8 FrameRTP4 Orchestrator overview
[Colour figure can be viewed at wileyonlinelibrary.com]

- *Decision Making Module*: it is responsible for detecting new threats. For this, it periodically requests information from the active flows (flow records) to Flow Stats Collector Module, calculates some features, and uses them as input to a ML model, previously loaded from a pickle[§] file. This classifier determines whether a given flow consists of an attack or not. In this context, once a new threat is detected, this module requires the Core Engine Module to create one or more access control rules to mitigate these active attacks. This module is flexible because it allows a developer to easily modify both the calculated features and the ML-based model (replace the pickle file). Further details about this module will be present in Section 4.3.2.
- *Wildcard Rules Generator Module*: it is responsible for grouping rules by creating wildcard rules. To this end, it implements an algorithm that will be described in Section 4.3.1. This module aims to enable the mechanism for real-time detection and mitigation of known attacks, which will be performed within the switches, and that is efficient and scalable in the sense that it reduces the number of flow rules stored in P4 table-based ACLs.
- *Core Engine Module*: it is the main component responsible for enabling the Orchestrator's functionalities by dictating the behavior of the system as a whole through calls to other modules. The Core Engine implements a loop control mechanism. We provide more details about its operation in Section 4.2.

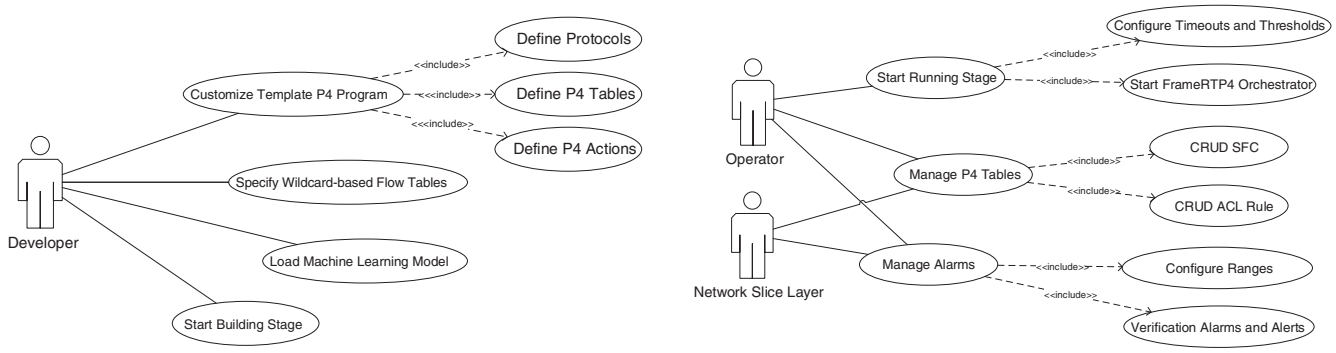
4.2 | FrameRTP4 roles and activities

The FrameRTP4 is multidimensional, requiring a combination of skills and activities for planning and execution. For this, we define every essential framework activity in terms of two roles: *developer* and *operator*.

Figure 9A shows the developer use cases. For instance, the developer prepares FrameRTP4 for *Building Stage*. In this context, it may customize the P4 program already available in the FrameRTP4 prototype by implementing new protocols (header definitions), P4 tables, and P4 actions. This flexibility enables the creation of different types of ACLs and mitigation actions. Besides, the developer should indicate to FrameRTP4 which of the P4 tables will be populated with wildcard rules (groups of access control rules), ie, the ACLs. Furthermore, the developer may add new ML models to detect new attacks by using pickle files. Finally, once the developer has finished his work, he can start the Building Stage of FrameRTP4.

On the other hand, Figure 9B shows the operator use cases. The operator role is responsible for initializing the *Running Stage*, which means putting the Orchestrator into operation. However, before this, the operator can set different timeouts and thresholds that determine the behavior of the system as a whole. Moreover, the operator may use FrameRTP4 in its Running Stage. In this scenario, by using the FrameRTP4 Northbound API, the operator can: query all information from the SQLite database, configure alarms and alerts for its use, and add and remove flow rules in any P4 table. It is worth noting that the operator role can be represented by individuals or a third-party management system (network slice layer).

[§]https://wiki.python.org/moin/UsingPickle



(A) Developer use cases. **(B)** Operator use cases.

FIGURE 9 FrameRTP4 roles

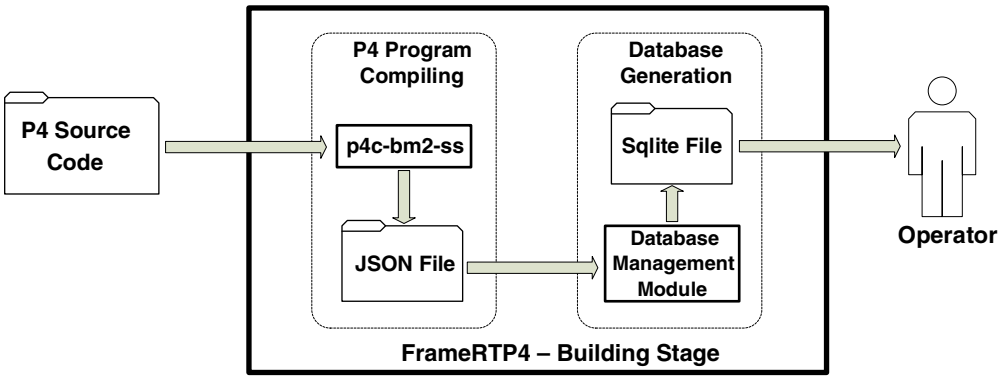


FIGURE 10 Building stage operations [Colour figure can be viewed at wileyonlinelibrary.com]

Furthermore, all the activities of the framework are organized in two stages: *Building* and *Running*. We describe these stages below.

Before starting up, FrameRTP4 must go through the Building Stage to generate the files and data necessary for its operation. Figure 10 shows the activities performed on this stage as well as the inputs and outputs. In this context, FrameRTP4 starts by compiling the customizable P4 program using the *p4c-bm2-ss*,[¶] an reference compiler for the P4 programming language. This compiler generates a JSON file describing all structures from the P4 program. Then, FrameRTP4 processes the JSON file to populate the SQLite database with data concerning tables, headers, actions, etc. Also, FrameRTP4 creates an SQLite table for each P4 table-based ACLs to store all the "clean" rules (not grouped). At the end of this stage, the FrameRTP4 is ready to run (Running Stage).

Once in operation, FrameRTP4 works on the Running Stage. At this stage, the FrameRTP4 initializes the Northbound API that can be used by an operator or a third-party management system to perform the lifecycle management of SFCs (ie, NS instances) and P4 table rules (eg, creating and deletion). The latter can be triggered for different reasons. Figure 11 illustrates a situation where an operator detects a new threat. In this scenario, once a new threat is detected, through the correlation of alarms or alerts, the operator uses the FrameRTP4 Northbound API to require Core Engine to create one or more access control rules to mitigate these active attacks. Once a request for rule creation arrives at the Core Engine Module, it identifies whether the target table is a P4 table-based ACL. If it is not, the Core Engine Module uses the Rules Management Module to create the rule on its P4 table within the switches. If so, the Core Engine Module first requests the Database Management Module to store the rule in the SQLite table related to this specific ACL. Then, it uses the Rules Management Module to create the rule on its P4 table within the switches.

[¶]https://github.com/p4lang/p4c

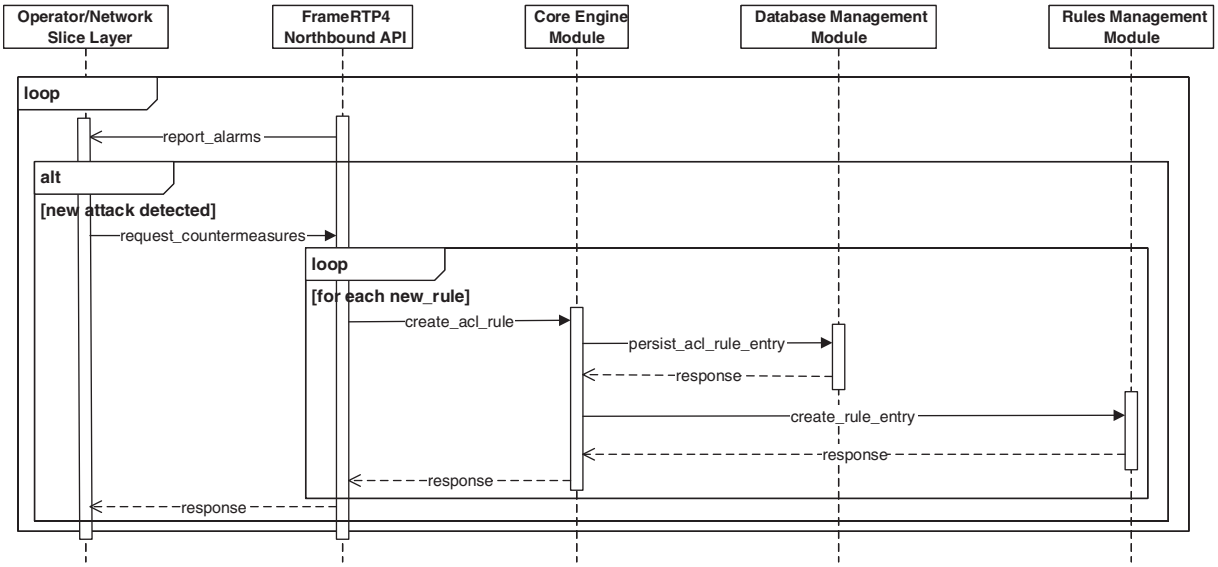


FIGURE 11 Running stage operations: manual detection of new threats

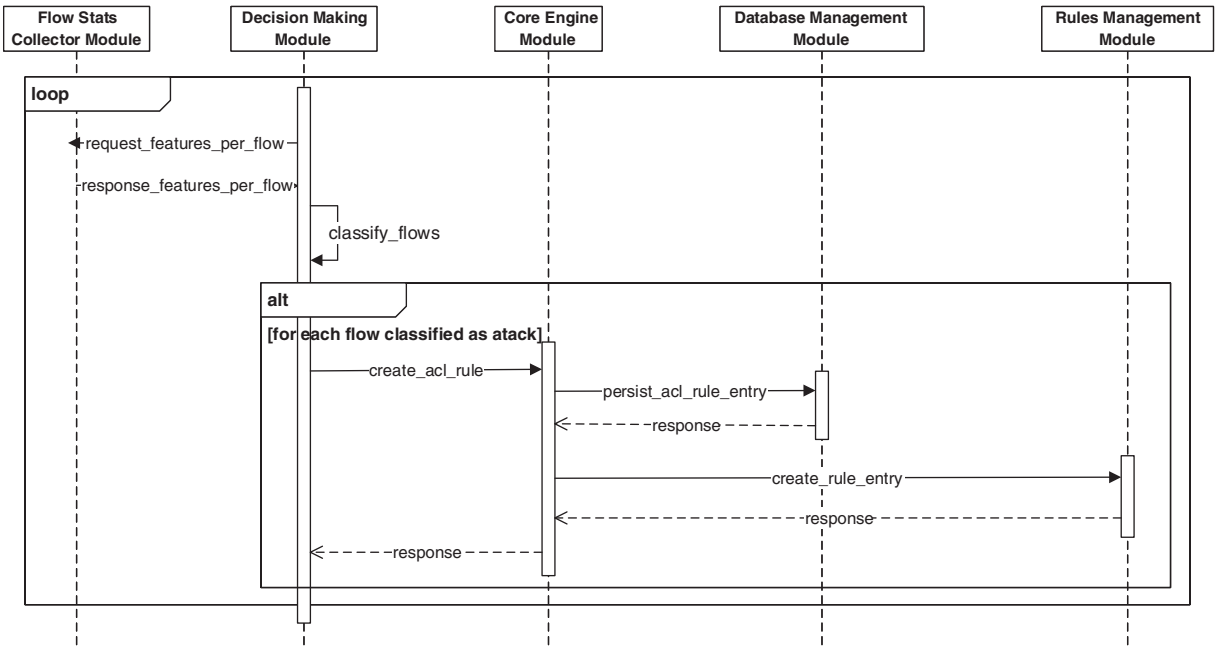


FIGURE 12 Running stage operations: automatic detection of new threats

Moreover, at the Running Stage, FrameRTP4 starts the Decision-Making Module, which will be responsible for the automatic detection of new threats (see Figure 12). To this end, it first reads a pickle file and loads the ML model. Then, it initiates a control loop where periodically requests information from the active flows to Flow Stats Collector Module, calculates some features, and uses them as input to the ML algorithm. In this scenario, if a new threat is detected, this module requires the Core Engine Module to create one or more access control rules to mitigate these active attacks.

Furthermore, at the Running Stage, FrameRTP4 performs automatic wildcard rules generation for the detection and mitigation of known threats (see Figure 13). Core Engine Module periodically queries the SQLite database to verify which P4 table-based ACLs have a number of new rules (not yet grouped) higher than a threshold value. For each table identified, the Core Engine Module uses the Database Management Module to request all "clean rules" and calls the Wildcarded

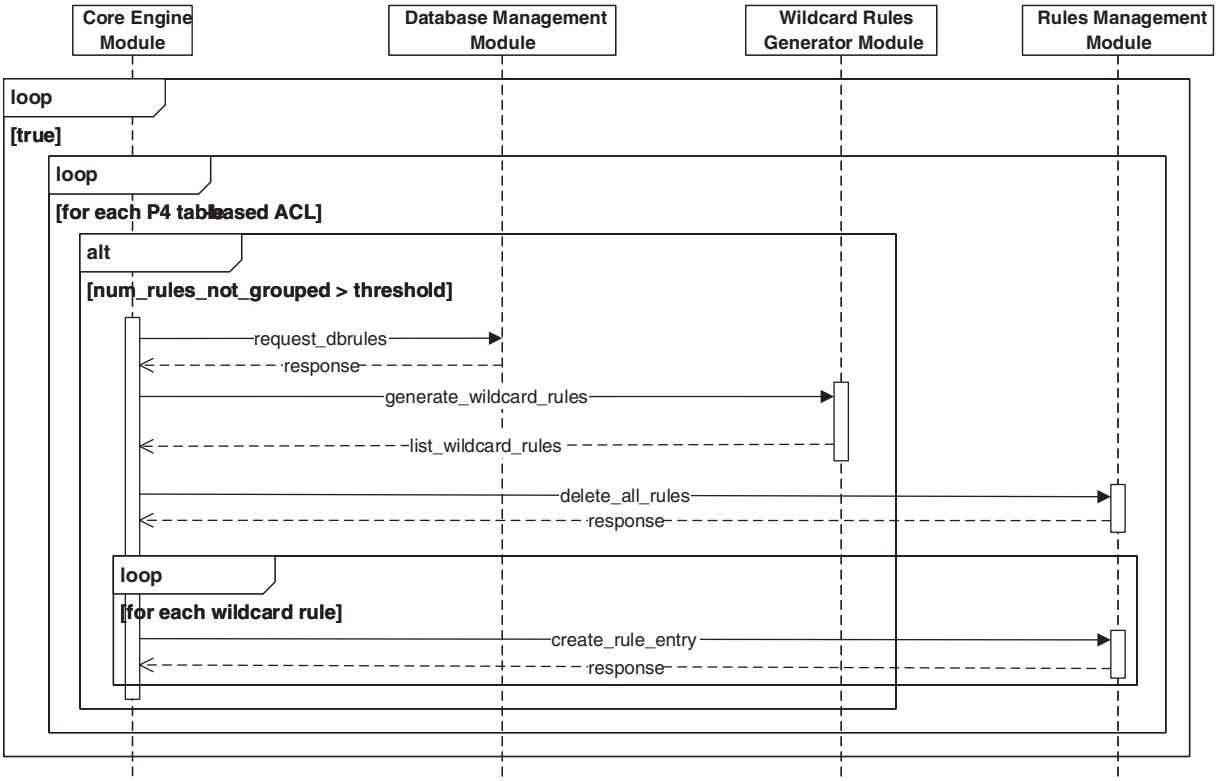


FIGURE 13 Running stage operations: automatic wildcard rules generation for the detection and mitigation of known threats

Rules Generator Module to generate the wildcarded rules. Then, the Core Engine removes all the rules from the respective tables within all the switches and adds the new generated wildcarded rules.

Finally, for scalability and autonomic purposes, it is worth mentioning that a real deployment scenario can include multiple FrameRTP4 Orchestrators. In this context, one FrameRTP4 Orchestrator can communicate with others via the Northbound API, that is, this interface also works as an Eastbound or Westbound API. In this case, by using this robust and highly adaptable communication system, orchestrators can work as distributed and secure autonomous agents to provide a multiagent system.[74]

4.3 | Additional implementation details

4.3.1 | Wildcard rules generator module

Generate and validate a wildcard position (*) is a hard task. It is necessary to group a set of different IPs. As an example, a decimal IPv4 address representation requires at least 10 IPs to put a single wildcard as 200.12*.30.40 (aggregation of 200.120.30.40, 200.121.30.40, 200.122.30.40 … 200.129.30.40). Because of that, aiming to achieve the smallest set of entries in the ACL table, we took into consideration bits to represent the IPv4 addresses and ports.

As mentioned in Section 4.1, the *Wildcard Rules Generator Module* aims to enable an efficient and scalable mechanism for real-time detection and mitigation of known attacks. To this end, it groups ACL rules into wildcard rules, thus reducing the number of flow rules stored in P4 table-based ACLs. The Algorithms 1 and 2 describe an solution for the rules compression problem, which we implement in our prototype to enforce security policies on a single FrameRTP4 Switch.

It is worth noting that FrameRPT4 enables the creation of ACLs that use the 5-tuple as a filter (see Section 4.1.1). However, as we implement a single P4 table for each protocol number (see Figure 5), our compression algorithm only considers four fields of a 5-tuple: source and destination IP addresses and ports.

Algorithm 1. Generate ACL Wildcard Rules

Require: 4-tuple ACL rules: *aclRawRules*
Ensure: Wildcarded ACL rules: *aclWildRules*
1: *filterColumns* ⇐ []
2: *wildValues* ⇐ GenerateValuesWithWildcards(*aclRawRules*[*SrcIP*])
3: ReplaceValues(*aclRawRules*[*SrcIP*], *wildValues*)
4: Add(*filterColumns*, *SrcIP*)
5: **foreach** *column* ∈ [*DstIP*, *SrcPort*, *DstPort*] **do**
6: *binIndexesGroups* ⇐ FilterByColumns(*aclRawRules*[*column*], *filterColumns*)
7: **foreach** *binIndexes* ∈ *binIndexesGroups* **do**
8: *binValues* ⇐ GetValuesByIndexes(*aclRawRules*[*column*], *binIndexes*)
9: *wildValues* ⇐ GenerateValuesWithWildcards(*binValues*)
10: ReplaceValuesByIndexes(*aclRawRules*[*column*], *wildValues*, *binIndexes*)
11: **end foreach**
12: Add(*filterColumns*, *column*)
13: **end foreach**
14: *aclWildRules* ⇐ RemoveDuplicates(*aclRawRules*)

The general process is described in Algorithm 1 while Figure 14 shows a simplified example of this process. Once invoked, the algorithm should receive an input that includes a table of 4-tuple rules (Source IP, Destination IP, Source Port, and Destination Port). Each line consists of an ACL rule (Figure 14). Then, it will iterate through each column of the 4-tuple table to generate wildcard values (*GenerateValuesWithWildcards*) for each column at a time. In this context, the algorithm begins by initializing the *filterColumns* variable as an empty list (line 1). This variable is used to identify columns that have already been processed, that is, those that have previously calculated wildcard values using the *GenerateValuesWithWildcards* method. Later we will see that this variable will be used to determine which groups of values from the same column should be used as input to that method. The first column to be processed by the algorithm consists of the one containing the source IP values (*SrcIP*), as illustrated in Step 1 of Figure 14. At this point, as no column

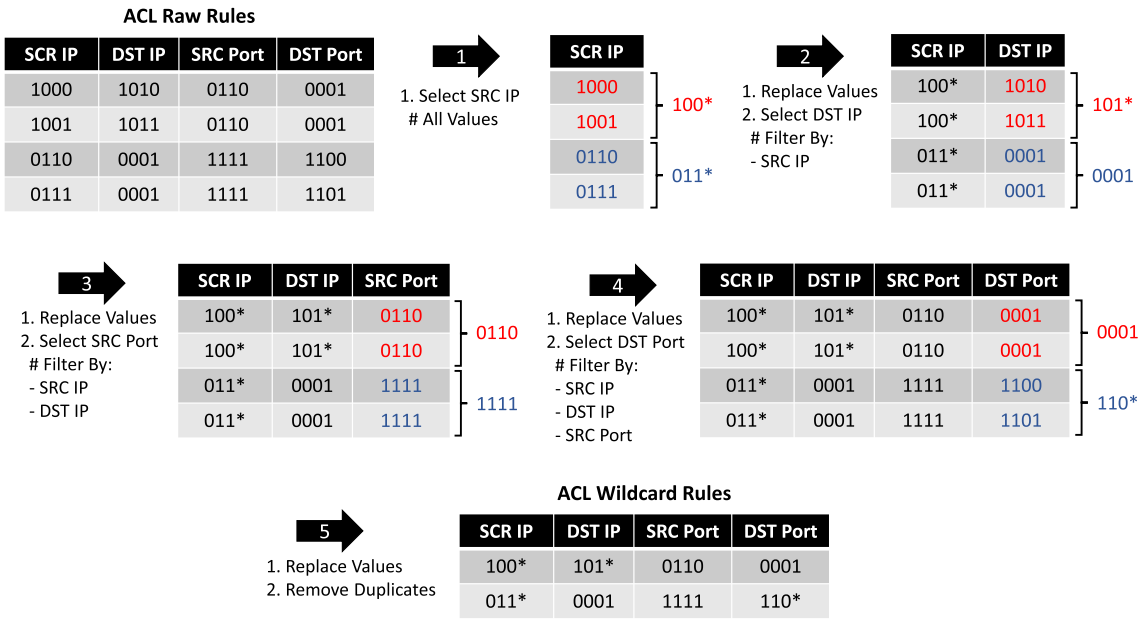


FIGURE 14 Algorithm 1 processing example [Colour figure can be viewed at wileyonlinelibrary.com]

has been previously processed, it uses all the values in this column as input to *GenerateValuesWithWildcards* method (line 2), which returns a list of wildcards values (*wildValues*). The algorithm proceeds by exchanging all binary values in this column for their respective wildcard values obtained previously, that is, each binary value will be replaced by a wildcard value to which it matches (line 3). Thus, once the *SrcIP* column is processed, it is added to *filterColumns* list (line 4) and the algorithm starts a loop to process the remaining columns (*DstIP*, *SrcPort*, and *DstPort*).

As Algorithm 1 goes through the loop (line 5), the variable *binIndexesGroups* is initialized by the result of calling *FilterByColumns* method (line 6). To explain how this method works, consider that the value set of a column, represented by *aclRawRules*[*column*], consists of an array where each element is associated with an index ranging from 1 to the array size. The *FilterByColumns* method returns index subsets of this array, where the indices of each subset point to elements that are part of a rule that contains the same values in the columns stored in *filterColumns*. For example, consider Step 2 in Figure 14. At this point, the loop will process the *DstIP* column. In this context, the *FilterByColumns* method will return two subsets, one with indexes 1 and 2 and the other with indexes 3 and 4, since their respective values are part of a rule that has the same value in the *SrcIP* column.

Once the *binIndexesGroups* list is initialized, the algorithm starts a loop to process each subset of that list (line 7). For each subset, it invokes the *GetValuesByIndexes* method to retrieve the list of binary values from array associated with its respective indexes (line 8). Such values are assigned to the *binValues* list. In turn, this list is used as input to the *GenerateValuesWithWildcards* method (line 9) and the returned wildcard values (*wildValues*) are used to override the binary values in the respective indexes of the *aclRawRules*[*column*] via the *ReplaceValuesByIndexes* method (line 10). At the end of a column loop, it is added to the *filterColumns* list to enable filtering (line 6) on the remaining columns. In Steps 2, 3, and 4 of Figure 14, you can see how this list evolves.

Once the loop in all columns is finished, the algorithm removes the duplicate rules in the resulting table (line 14), as illustrated in Step 5 of Figure 14. Finally, it assigns this result to a new table, represented by *aclWildRules* variable, which will be the algorithm's return.

Algorithm 2. Generate Values with Wildcards

Require: binary values list: *currentList*
Ensure: reduced list (wildcard values): *solutionList*

```
1:  amountWildCards ⇐ CalculateAmountWildcards(currentList)
2:  while #currentList > 0 do
3:      oldSize ⇐ #currentList
4:      pattern ⇐ GenerateTestPattern(currentList, amountWildCards)
5:      combList ⇐ GenerateCombinations(pattern, amountWildCards)
6:      foreach comb ∈ combList do
7:          possibleSolutionsList ⇐ SearchPossibleSolutions(currentList, pattern, comb)
8:          foreach solution ∈ possibleSolutionsList do
9:              validSolution ⇐ ValidateSolution(currentList, solution)
10:             if validSolution then
11:                 Add(solution, solutionList)
12:                 Remove(solution, currentList)
13:             end if
14:         end foreach
15:     end foreach
16:     newSize ⇐ #currentList
17:     if oldSize! = newSize then
18:         amountWildCards ⇐ CalculateAmountWildcards(currentList)
19:     else
20:         amountWildCards ⇐ amountWildCards − 1
21:     end if
22: end while
23: return solutionList
```

Nevertheless, the heart of the algorithm is Algorithm 2, which implements the solution for generating wildcard values from a binary value list (method *GenerateValuesWithWildcards* of Algorithm 1). It is worth mentioning that this problem consists of a combinatorial optimization problem, that is, we try to find an optimal solution from a finite set of combinations. As a starting point, we implement an exhaustive search-based approach. Any developer can replace this algorithm with one that improves performance through heuristics or metaheuristics. We describe the operation of Algorithm 2 below.

Once invoked, Algorithm 2 should receive a list of binary values (*currentList*) as input. Then, it will start a loop whose focus is to find possible wildcard value solutions that encompass *currentList* elements. When a solution is found, it is added to *solutionList*, and the elements that match it are removed from the *currentList*. The loop will continue until *currentList* has no more elements, that is, an optimal set of wildcard values has been found. In the end, the algorithm will return the *solutionList*. We will define the operations related to the solution search process below.

The solution discovery process consists of a set of methods (eg, *CalculateAmountWildcards*, *GenerateTestPattern*, *GenerateCombinations*, *SearchPossibleSolutions*, and *ValidateSolution*). For clarification purposes, we define only their main goals. For further implementation details, please refer to the following file on GitHub: https://github.com/michelsb/FrameRTP4/blob/master/prototype/framertp4/rtp4rules/rtp4rules_generator.py.

Before entering the loop, we first calculate (line 1) the number of wildcards that possible solutions should have (*amountWildCards*). This amount will be a maximum of $\log_2$(#*currentList*), that is, for each wildcard, it is necessary 2^n entries, where *n* is the amount o wildcards. Once within the loop, the first method to invoke is the *GenerateTestPattern*. This method is used to find out, given a list of binary values, which positions in possible solutions can receive wildcards for testing. The output of this method returns a binary value (*pattern*) that includes wildcards in a number of positions that must be greater than or equal to *amountWildCards*. Then, *pattern* is used as input to the *GenerateCombinations* method (line 5), which is responsible for extracting the wildcard position list from the *pattern* and returns a set of all combinations with *amountWildCards* positions. Thus, for each combination, we use the *SearchPossibleSolutions* method to search for possible solutions with wildcards at the selected positions. Besides, for each solution found, we call the *ValidateSolution* method to verify that all possible binary values that can be obtained from this solution are contained in *currentList*. If so, such a solution will be added to *solutionList*, and the elements that match it will be removed from the *currentList*.

Once all combinations have been tested, the algorithm evaluates if any solutions were found by examining whether the *currentList* size is reduced. If so, it will call the *CalculateAmountWildcards* method (line 18) again to calculate the new wildcard amount for possible solutions. Otherwise, it will just decrement the *amountWildCards* by one (line 20).

Finally, the list of wildcards found is returned (line 23).

4.3.2 | Decision making module

As mentioned in Section 4.1, the *Decision-Making Module* provides some autonomy for the framework by using ML to detect new threats. This module is flexible because it allows a developer to easily modify both the calculated features and the ML-based model (replace the pickle file).

In our prototype implementation, this module periodically requests information from the active flows (flow records) to *Flow Stats Collector Module*. Each flow record includes the following information: source and destination IPs and ports, protocol number, number of packets and bytes, and first time have seen. Then, the module uses such information to keep track of bidirectional flows and calculate the following metrics per flow: the number of packets/bytes sent/received, and packet/bit rate per second. Finally, these metrics are used as input to a Random Forest model. This classifier determines whether a given flow consists of an attack or not. In this context, if a new threat is detected, this module requires the Core Engine Module to create one or more access control rules to mitigate these active attacks.

It is noteworthy that it is not part of the scope of this article to evaluate the best algorithms of ML for the framework. Therefore, our goal is only to generate an example model that can be replaced or customized in future framework applications. Besides, we choose the Random Forest algorithm because it is one of the most popular ensemble classifiers and has been widely used as a technique for intrusion detection.[75]

The process of training our Random Forest model involves providing the learning algorithm with training data to learn from. For this, we generate a labeled dataset from a set of measurements performed by the GTA group of the Federal University of Rio de Janeiro (UFRJ).[76] These measurements consist of a set of pcap files from real/controlled traffic measurements held in 2016 (total of 95 GB). It includes both legitimate user behaviors and real network threats. The attacks

were performed in a controlled manner by the Kali[#] Linux penetration testing tool, with the most recent attack basis until its creation date, containing a total of seven types of denial of service attacks and nine types of port scans. Every attack has its pcap file, and normal traffic comprises four pcap files. We create our labeled dataset as follows. For each pcap, we generate a CSV file using the Tranalyser Tool[‖] that groups packets into bidirectional flows based on source IP/Port, destination IP/Port, and protocol number. Besides, for each flow, this tool generates 95 features (including key parameters and statistics). Finally, we concatenate all the CSVs and include two (2) Labels: traffic type (0 - Attack and 1 - Normal) and file name (which can be used for multiclass classification). Once we are done with training our model, we save it as a pickle[**] file and make it available for the FrameRTP4. For further details about the training process, please refers to the following GitHub file: https://github.com/michelsb/FrameRTP4/blob/master/prototype/framertp4/ml_model/generate_model.py

5 | EVALUATION AND RESULTS

In this section, we evaluate our approach. To this end, we guide our experiments to answer two research questions, as follows.

- *Research Question 1*: Can SFCMon be used as a monitoring tool to keep track malicious flows (see Section 5.1)?
- *Research Question 2*: How efficient is the wildcard rule generation algorithm (see Section 5.2)?

In the next subsections, we detail the experiments performed and present the results for each question mentioned above.

5.1 | Research question 1

Previous experiments[22] demonstrate that SFCMon introduces a negligible performance penalty while providing significant scalability gains. These results point to the potential benefits of SFCMon to be part of SFC architecture, aiming to assist with more advanced monitoring applications in an efficient and scalable way. However, the previous work[22] only evaluated the capacity of the SFCMon to keep track of large flows. Therefore, it is essential to assess how effective SFCMon is when applied for other monitoring tasks such as attack detection.

Therefore, this research question attempts to answer whether the use of SFCMon, as an efficient and scalable tool for tracking large flows, may apply to the task of detecting malicious flows in 5G NS scenarios.

5.1.1 | Experimental setup

We develop a simulator[††] to evaluate the SFCMon's ability to detect malicious flows. It is a Python program that simulates the execution of an adapted implementation of SFCMon (described in Section 4.1.1). In this scenario, we use as input the CTU-13 dataset[77] created in CTU University, Czech Republic. It encompasses 13 different scenarios where particular malware traffic was captured in each scenario. Each scenario dataset consists of botnet traffic mixed with normal and background communication traffic. Table 2 gives details of the traffic capture process (eg, botnet type, duration, number of infected machines) as well as the malware and traffic flow records in each scenario. For each scenario, this dataset provides two files that are used as input for our simulator. The first is a pcap file that contains the data for all packets. The second is a NetFlow file that includes all bidirectional flows with their respective labels (assigned manually).

Figure 15 shows how our simulator works. It is noteworthy that we run this simulator for each scenario. In this context, for each scenario, we use the tshark[‡‡] tool to convert the pcap file into a CSV file where each line consists of a packet described by the following data (columns): arrival time, source and destination IPs, source and destination ports,

[#] https://www.kali.org/

[‖] https://tranalyzer.com/

[**] https://wiki.python.org/moin/UsingPickle

[††] https://github.com/michelsb/FrameRTP4/tree/master/evaluation/sfcmon-simulator

[‡‡] https://www.wireshark.org/docs/man-pages/tshark.html

TABLE 2 Detail of flow records—CTU-13 intrusion dataset

Id.	Bot	Duration (h)	Size	#Bots	Total packets	Total flows (% Bot, % Norm, % Back)
1	Neris	6.15	52 GB	1	71 971 482	2 824 636 (1.44, 1.07, 97.47)
2	Neris	4.21	60 GB	1	71 851 300	1 808 122 (1.15, 0.5, 98.33)
3	RBot	66.85	121 GB	1	167 730 395	4 710 638 (0.56, 2.48, 96.94)
4	RBot	4.21	53 GB	1	62 089 135	1 121 076 (0.15, 2.25, 97.58)
5	Virut	11.63	37.6 GB	1	4 481 167	129 832 (1.68, 3.6, 95.7)
6	Mentri	2.18	30 GB	1	38 764 357	558 919 (0.82, 1.34, 97.83)
7	Sogou	0.38	5.8 GB	1	7 467 139	114 077 (0.05, 1.47, 98.47)
8	Merli	19.5	123 GB	1	155 207 799	2 954 230 (2.57, 2.46, 97.32)
9	Neris	5.18	94 GB	10	115 415 321	2 753 884 (6.68, 1.57, 91.7)
10	RBot	4.75	73 GB	10	90 389 782	1 309 791 (8.11, 1.2, 90.67)
11	RBot	0.26	5.2 GB	3	6 337 202	107 251 (7.6, 2.53, 89.85)
12	NSIS.ay	1.21	8.3 GB	3	13 212 268	325 471 (0.65, 2.34, 96.99)
13	Virut	16.36	34 GB	1	50 888 256	1 925 149 (2.07, 1.65, 96.26)

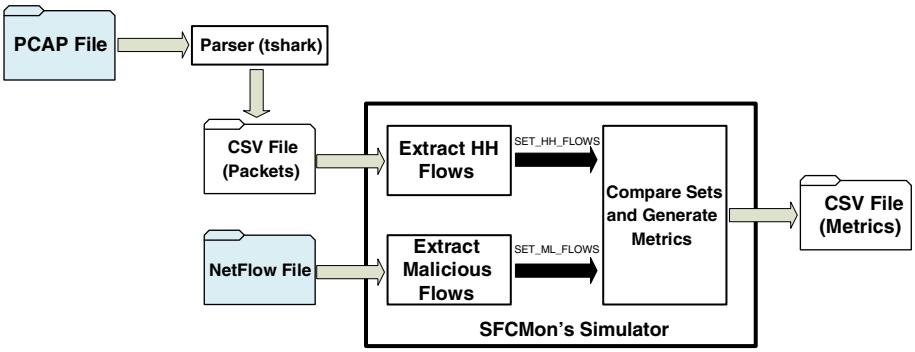


FIGURE 15 SFCMon's simulator overview [Colour figure can be viewed at wileyonlinelibrary.com]

protocol identifier, and packet size. This CSV file is then used as input to an attached implementation of SFCMonCMS that aims to extract all HH 5-tuple flows and store them in a set (*SET_HH_FLOWS*).

For this experiment, we consider that an HH is a flow that has more than 0.01% of the total packets that SFCMon has processed so far. Besides, SFCMon only begins performing HH tests after processing more than 1000 packets in a given time window. Moreover, we follow the recommendations of Reference 48 to design our CMS. In this context, considering a packet stream size of 10 million, we reach the following parameters: $w = 5436$ and $d = 5$. Besides, we set the cell size to 3 bytes. Therefore, our CMS consumes a total of approximately 82 KB, less than 1.4 MB. According to Sivaraman et al,[44] switches should work a limited amount of memory available (about 1.4 MB) per stage to maintain line-rate processing (at 10-100 GB/second).

In the end, we compare these HH flows with ground truth. To generate the ground truth, we implement a parser that processes the NetFlow file, extracts all malicious 5-tuple flows, and stores them in a set (*SET_ML_FLOWS*). Finally, after comparing the two sets, the simulator generates the following performance metric:

- *Recall Rate (True Positive Rate [TPR])*: refers to the proportion of flows that are malicious and that have been reported as HHs by SFCMonCMS;

The aforementioned metric aims to evaluate SFCMon's ability to track malicious flows by simply monitoring large streams. Moreover, we assess the above metric concerning a single factor: the time window to delimit a stream of packets. It is worth mentioning that SFCMon considers as stream any set of packets that it can process before having its structures reset by the FrameRTP4 Controller. In this context, we assumed the following levels: 30, 60, 90, and 120 seconds. Finally, we run the experiments using a Server with Intel Core i7-8550U 1.80 GHz processors and 16 GB RAM.

TABLE 3 Simulation results—CTU-13 intrusion dataset

	Time window size (s)			
Id.	**30**	**60**	**90**	**120**
1	99.88	99.98	99.97	99.99
2	99.86	99.91	99.92	99.92
3	99.55	99.68	99.66	99.65
5	99.76	99.88	99.88	99.88
6	94.54	96.79	97.87	98.67
7	90.90	90.90	86.36	90.90
8	10.42	10.72	10.62	10.32
9	99.62	99.74	99.76	99.79
11	91.66	91.66	91.66	91.66
13	97.08	98.33	98.98	99.14

5.1.2 | Results

In this subsection, we present the results of the experiments. Note that due to inconsistencies in the NetFlow file (incomplete or corrupted data), we were unable to run the simulator for scenarios 4, 10, and 12. For the rest, the results of the TPR metric by Time Window Size can be seen in Table 3.

For most scenarios, the results presented values above 90% and showed an increasing TPR as we increased the time window size. As an exception, scenario 8 reached bad values, in the range of 10%. However, the problem is probably in the static settings defined for the thresholds. Therefore, we can conclude that the results are promising.

Finally, it is worth mentioning that these experiments are only in their initial phase, enabling several opportunities for improvement. For example, the method for setting the thresholds is static. For the growth of the TPR metric, this method can be improved in future work by using adaptive thresholds based on Reinforcement Learning and Deep Reinforcement Learning, for instance.

5.2 | Research question 2

As mentioned in Section 2.3, the use of wildcards and binary numbers allows greater efficiency in the rule compression process. In this sense, the *Wildcard Rules Generator Module* provides an algorithm (see Section 4.3.1) for 5-tuple rules compression using wildcards to enforce security policies directly within FrameRTP4 switches, thus reducing the number of flow rules stored in P4 table-based ACLs. Since this problem is combinatorial, our initial solution only seeks to find the optimal solution without worrying about performance. However, any developer can replace this algorithm with one that improves performance through heuristics or metaheuristics.

Therefore, this research question attempts to answer whether our algorithm implementation is efficient only in its compression capacity since it deals with a problem that cannot be solved by a polynomial-time algorithm. Furthermore, due to the difficulty in finding real 5-tuple datasets for testing, we decide to evaluate the algorithm's capability only for a specific binary value list and focus our experiments on Algorithm 2 only (the core of our implementation).

5.2.1 | Experimental setup

As in the previous experiment (Section 5.1), we also developed a simulator,[§§] in this case, to evaluate the binary value compression capability of Algorithm 2. It consists of a Python-based program that reads a file with a set of subnets of different masks, transforms these ranges into a set of IPs, converts these IPs to 32-bit binary values and uses them as input to Algorithm 2. In this context, we conduct experiments using the FireHOL IP Lists[78] as input, a real blacklist that

[§§] https://github.com/michelsb/FrameRTP4/tree/master/evaluation/wildcard-simulator

TABLE 4 IP address compression results—FireHOL IP lists

Selected subnets	Input size	Output size	Duration (s)	Compression ratio (%)
/23 (1119)	572 928	938	4395	99,83 (Algorithm 2) 99,80 (simple mask)
/24 (1996)	510 976	1649	6412	99,68 (Algorithm 2) 99,60 (simple mask)
/22 (289), /23 (1119), /24 (1996), /25 (20), /26 (14), /27 (11), /28 (3), /29 (6), /30 (6), /31 (14), /32 (780)	1 384 604	3744	29 587	99,73 (Algorithm 2) 99,69 (simple mask)

contains a total of 4742 subnets (at the time of writing), and masks ranging from /8 to /32, totaling 624 101 504 unique IPs. These subnets are a composition from different sources (eg, unroutable IPs lists, malware lists), and they are updated automatically every time any of their sources are updated.

In the end, we compare the effects of a wildcard based representation concerning a simple mask representation in reducing the size of our dataset. For this, we define the following metric for performance evaluation:

$$\text{Compression ratio} = 1 - \frac{C}{T}$$

where:

- C is the number of compressed values generated;
- T is the number of unique IP addresses.

Furthermore, due to scalability issues, we have limited the size of the algorithm input. In this context, we decide to use only a couple of subnets from the dataset. In this scenario, we focus only on the subnets with the most entries. Therefore, we assumed three types of listings for metric evaluation: the first one includes only subnets with mask /23, the second one comprises only entries with mask /24, and the third one consists of all entries ranging from mask /22 to /32. Finally, the experiments are carried on a desktop computer with Intel Core i7-8550U 1.80 GHz processors, 16 GB RAM, and running Windows 10 operating system.

5.2.2 | Results

The result from the three experiments is given in Table 4. Note that, for all three input types, the output from Algorithm 2 achieved a higher Compression Ratio than using a simple mask. This difference happens because our algorithm evaluates the use of wildcards at any binary value positions, ie, it is not restricted to a contiguous sequence of positions, as in Classless Interdomain Routing. Therefore, our implementation can group IPs from different subnets.

However, our solution is in the early stage, and although the results look promising, the execution time is still a limiting factor (the shortest time was approximately 1 hour and 14 minutes). Nevertheless, if this work were done manually (by an operator), the time cost would probably be higher. In addition, we can use workarounds in FrameRTP4 to overcome the time problem, such as using caches to store wildcard rules previously calculated. Finally, as future work, we intend to implement a metaheuristic that aims to improve the performance of Algorithm 2.

6 | THREATS TO VALIDITY

This section describes several threats to the validity of this study to evaluate the quality of this research. The potential validity threats to our study and the strategies for overcoming them are listed below:

InternalValidity: In Sections 5.1 and 5.2 the experiments were performed using two simulators implemented by us in Python. For this reason, we cannot guarantee that there are no errors in the simulator, although several tests have been performed checking the capacity and integrity of the generated results. As an example, some of the validations were performed in small scale scenarios, where it was possible to predict the expected results and thus validate the behavior of the developed simulator.

Construct Validity: The evaluated metrics were based on works published in the area of computer networks that were validated through peer reviews at relevant conferences in the area, ie, they were not chosen randomly. Moreover, the developed solutions were based on heuristics and techniques known in the literature such as P4, CMS, IBLT, BF, exhaustive search-based algorithms.

External Validity: Concerning the experiments in Sections 5.1 and 5.2, the results looked promising. However, it is not possible to state that the same results will be obtained in real scenarios. Since the implementations were based on real-world simplifications, abstractions were made trying to approximate a model to a real scenario. Besides, for each experiment, we consider only one type of dataset. It is worth mentioning that these experiments are only in their initial phase, enabling several opportunities for improvement. For example, in Section 5.1, the method for setting the thresholds is static. For the growth of the TPR metric, this method can be improved in future work by using adaptive thresholds based on Reinforcement Learning and Deep Reinforcement Learning, for instance. On the other hand, in Section 5.2, we can use workarounds in FrameRTP4 to overcome the time problem, such as using caches to store wildcard rules previously calculated. Moreover, any developer can replace the Algorithm 2 (exhaustive search-based approach) with one that improves performance through heuristics or metaheuristics. Therefore, the next step of the work would be to validate the proposed solutions in a real environment and not just simulated ones.

Statistical Conclusion Validity: The focus of the experiments conducted in Section 5 aimed to answer two research questions (RQs). Both RQs sought to validate FrameRTP4 solutions according to their outputs, that is, a validation task. Therefore, we did not perform statistical tests on the experiments. As future work, we intend to apply these tests to evaluate the use of computational resources as well as compare our solutions with others presented in related works.

7 | CONCLUSION AND FUTURE DIRECTIONS

This work proposed the FrameRTP4, a framework that aims to deliver real-time attack detection and mitigation mechanisms in 5G NS scenarios. For this, it follows the SDN architecture by separating the solutions to the problem into the data and control planes. Concerning the data plane, FrameRTP4 Switch provides a customizable P4 program that implements an SFC protocol layer to enable the lifecycle management of NS instances. Moreover, this program offers a set o capabilities for attack detection and mitigation at the line rate. In this context, it supports an efficient and scalable P4 table-based ACL that applies wildcard rules for the detection and mitigation of known attacks. Besides, it also makes use of one of our previous work,[22] namely SFCMon, to provide an efficient and scalable mechanism to track network flows that aim to assist in the detection of new attacks.

On the control plane, the FrameRTP4 provides two components. The FrameRTP4 Controller enables target-independent flow control management by using different technologies such as P4Runtime or Thrift. On the other hand, the FrameRTP4 Orchestrator controls one or more FrameRTP4 Controllers to provide security for end-to-end NSs. To this end, it enables an operator or a third-party management system to perform the lifecycle management of SFCs (ie, NS instances) and P4 table rules (adding and deletion). Besides, it also delivers some autonomous tasks such as the wildcard rules generation and the detection of new threats by using ML algorithms.

Presented initial experiments point to the potential benefits of FrameRTP4 to be part of a 5G NS infrastructure. In this context, we have shown that SFCMon could be used as a FrameRTP4 switch component to keep track of malicious flows. Besides, we have demonstrated that a 5-tuple rules compression algorithm, despite time constraints, can be used as a tool for reducing ACL rules on switches.

FrameRTP4 is customizable, which means its key features can be extended or replaced by developers or operators. Our prototype implementation can be found in the following GitHub repository: https://github.com/michelsb/FrameRTP4/tree/master/prototype. Moreover, for reproducible research, all data and scripts used to analyze the results and the source code of the simulators can be found in the following repositories: https://github.com/michelsb/FrameRTP4/

tree/master/evaluation/sfcmon-simulator (SFCMon experiments) and https://github.com/michelsb/FrameRTP4/tree/master/evaluation/wildcard-simulator (Wildcard Generation Rules experiments).

As future research, we delimit the following points:

- Currently, in SFCMon, the method for setting the thresholds is static. This fact may cause some precision issues, as demonstrated in a previous work.[22] In future work, this method can be improved by using adaptive thresholds based on Reinforcement Learning and Deep Reinforcement Learning;
- Our current 5-tuple rules compression implementation consists of an exhaustive search algorithm. In this context, the execution time is still a limiting factor. In future work, heuristics or metaheuristics may be proposed to improve algorithm performance;
- Finally, it was not part of the scope of this article to evaluate the best algorithms of ML for our framework. Currently, our implementation performs new attack detection in offline mode within FrameRTP4 Orchestrator. Future work could investigate the possibility of offloading ML algorithms directly inside the switches.

ACKNOWLEDGMENTS

This work was partially funded by the National Science Foundation (NSF-USA) and the Rede Nacional de Ensino e Pesquisa (RNP-Brazil) under the "EAGER: USBRCCR: Securing Networks in the Programmable Data Plane" project. This work was developed while Stenio Fernandes was with UFPE, Brazil. He is now with Element AI, Canada.

ORCID

Michel Bonfim https://orcid.org/0000-0001-8665-3675

REFERENCES

1. Gupta A, Jha RK. A survey of 5G network: architecture and emerging technologies. *IEEE Access*. 2015;3:1206-1232. https://doi.org/10.1109/ACCESS.2015.2461602.
2. Group GPAW. View on 5G Architecture. tech rep; 2016.
3. Group GPAW. 5G PPP use cases and performance evaluation models. tech rep; 2016.
4. Zhang S. An overview of network slicing for 5G. *IEEE Wirel Commun*. 2019;26(3):111-117. https://doi.org/10.1109/MWC.2019.1800234.
5. Ordonez-Lucena J, Ameigeiras P, Lopez D, Ramos-Munoz JJ, Lorca J, Folgueira J. Network slicing for 5G with SDN/NFV: concepts, architectures, and challenges. *IEEE Commun Mag*. 2017;55(5):80-87. https://doi.org/10.1109/MCOM.2017.1600935.
6. Bonfim MS, Dias KL, Fernandes SFL. Integrated NFV/SDN architectures: a systematic literature review. *ACM Comput Surv*. 2019;51(6):114:1-114:39. https://doi.org/10.1145/3172866.
7. Sattar D, Matrawy A. Towards secure slicing: using slice isolation to mitigate ddos attacks on 5G core network slices. Paper presented at: Proceedings of the 7th Annual IEEE Conference on Communications and Network Security (CNS 2019); 2019; Washington, DC.
8. Accenture and Ponemon Institute. Cost of Cyber Crime Study - Insights on the Security Investments that Make a Difference; 2017:1-56.
9. Khan MA, Salah K. IoT security: Review, blockchain solutions, and open challenges. *Futur Gener Comput Syst*. 2018;82:395-411. https://doi.org/10.1016/j.future.2017.11.022.
10. Clay P. A modern threat response framework. *Netw Secur*. 2015;2015(4):5-10. https://doi.org/10.1016/S1353-4858(15)30026-X.
11. Rebecchi F, Boite J, Nardin P, Bouet M, Conan V. Traffic monitoring and DDoS detection using stateful SDN. Paper presented at: Proceedings of the 2017 IEEE Conference on Network Softwarization (NetSoft); 2017:1-2.
12. Bawany NZ, Shamsi JA, Salah K. DDoS attack detection and mitigation using SDN: methods, practices, and solutions. *Arab J Sci Eng*. 2017;42(2):425-441.
13. Sahay R, Blanc G, Zhang Z, Debar H. Towards autonomic DDoS mitigation using software defined networking. Paper presented at: SENT 2015: NDSS Workshop on Security of Emerging Networking Technologies. Internet Society; 2015.
14. Vizváry M, Vykopal J. Future of ddos attacks mitigation in software defined networks. Paper presented at: Proceedings of the IFIP International Conference on Autonomous Infrastructure, Management and Security; 2014:123-127; Springer.
15. Giotis K, Argyropoulos C, Androulidakis G, Kalogeras D, Maglaris V. Combining OpenFlow and sFlow for an effective and scalable anomaly detection and mitigation mechanism on SDN environments. *Comput Netw*. 2014;62:122-136. https://doi.org/10.1016/j.bjp.2013.10.014.
16. Bosshart P, Daly D, Gibb G, et al. P4: programming protocol-independent packet processors. *SIGCOMM Comput Commun Rev*. 2014;44(3):87-95. https://doi.org/10.1145/2656877.2656890.
17. Bosshart P, Gibb G, Kim HS, et al. Forwarding metamorphosis: fast programmable match-action processing in hardware for SDN. *ACM SIGCOMM '13 Computer Communication Review*. Vol 43. New York, NY: ACM; 2013:99-110.
18. Patgiri R, Nayak S, Borgohain SK. Preventing DDoS using bloom filter: a survey. *EAI Endorsed Transactions on Scalable Information Systems*. 2018;5(19):1-9 abs/1810.06689.
19. Shirali-Shahreza S, Ganjali Y. ReWiFlow: Restricted Wildcard OpenFlow Rules. *SIGCOMM Comput Commun Rev*. 2015;45(5):29-35. https://doi.org/10.1145/2831347.2831352.
20. Pereira F, Neves N, Ramos FMV. Secure network monitoring using programmable data planes. Paper presented at: Proceedings of the 2017 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN); 2017:286-291.

21. Yang T, Jiang J, Liu P, et al. Elastic sketch: adaptive and fast network-wide measurements. *SIGCOMM '18*. New York, NY: ACM; 2018:561-575.
22. Bonfim M, Dias K, Fernandes S. SFCMon: an efficient and scalable monitoring system for network flows in SFC-enabled Domains. Paper presented at: Proceedings of the Anais do 6th Workshop Pré-IETF. SBC SBC; 2019; Porto Alegre, RS, Brasil.
23. Halpern JM, Pignataro C. Service Function Chaining (SFC) architecture. *RFC*. 2015;7665. https://doi.org/10.17487/RFC7665.
24. Quinn P, Elzur U, Pignataro C. Network Service Header (NSH). *RFC*. 2018;8300.
25. McKeown N, Anderson T, Balakrishnan H, et al. OpenFlow: enabling innovation in campus networks. *SIGCOMM Comput Commun Rev*. 2008;38(2):69-74. https://doi.org/10.1145/1355734.1355746.
26. Kreutz D, Ramos FMV, Veríssimo PE, Rothenberg CE, Azodolmolky S, Uhlig S. Software-defined networking: a comprehensive survey. *Proc IEEE*. 2015;103(1):14-76. https://doi.org/10.1109/JPROC.2014.2371999.
27. ONF. OpenFlow switch specification - Version 1.3.5; 2015. https://3vf60mmveq1g8vzn48q2o71a-wpengine.netdna-ssl.com/wp-content/uploads/2014/10/openflow-switch-v1.3.5.pdf.
28. Scott-Hayward S, Natarajan S, Sezer S. A survey of security in software defined networks. *IEEE Commun Surv Tutor*. 2016;18(1):623-654. https://doi.org/10.1109/COMST.2015.2453114.
29. Dargahi T, Caponi A, Ambrosin M, Bianchi G, Conti M. A survey on the security of stateful SDN data planes. *IEEE Commun Surv Tutor*. 2017;19(3):1701-1725. https://doi.org/10.1109/COMST.2017.2689819.
30. Paolucci F, Civerchia F, Sgambelluri A, Giorgetti A, Cugini F, Castoldi P. P4 Edge node enabling stateful traffic engineering and cyber security. *IEEE/OSA J Optic Commun Netw*. 2019;11(1):A84-A95.
31. Zhu S, Bi J, Sun C, Wu C, Hu H. SDPA: enhancing stateful forwarding for software-defined networking. Paper presented at: Proceedings of the 2015 IEEE 23rd International Conference on Network Protocols (ICNP); 2015:323-333.
32. Moshref M, Bhargava A, Gupta A, Yu M, Govindan R. Flow-level state transition as a new switch primitive for SDN. *SIGCOMM Comput Commun Rev*. 2014;44(4):377-378. https://doi.org/10.1145/2740070.2631439.
33. Bianchi G, Bonola M, Capone A, Cascone C. OpenState: programming platform-independent stateful openflow applications inside the switch. *SIGCOMM Comput Commun Rev*. 2014;44(2):44-51. https://doi.org/10.1145/2602204.2602211.
34. Sivaraman A, Cheung A, Budiu M, et al. Packet transactions: high-level programming for line-rate switches. *SIGCOMM '16*. New York, NY: ACM; 2016:15-28.
35. McClurg J, Hojjat H, Foster N, Černý P. Event-driven network programming. *SIGPLAN Not*. 2016;51(6):369-385. https://doi.org/10.1145/2980983.2908097.
36. Arashloo MT, Koral Y, Greenberg M, Rexford J, Walker DSNAP. Stateful network-wide abstractions for packet processing. *SIGCOMM '16*. New York, NY: ACM; 2016:29-43.
37. Rifai M, Huin N, Caillouet C, et al. Minnie: an SDN world with few compressed forwarding rules. *Comput Netw*. 2017;121:185-207.
38. Braun W, Menth M. Wildcard compression of inter-domain routing tables for openflow-based software-defined networking. Paper presented at: Proceedings of the 2014 3rd European Workshop on Software Defined Networks; 2014:25-30.
39. Bloom BH. Space/time trade-offs in hash coding with allowable errors. *Commun ACM*. 1970;13(7):422-426. https://doi.org/10.1145/362686.362692.
40. Cormode G, Muthukrishnan S. An improved data stream summary: the count-min sketch and its applications. *J Alg*. 2005;55(1):58-75. https://doi.org/10.1016/j.jalgor.2003.12.001.
41. Grandi F. On the analysis of Bloom filters. *Inf Process Lett*. 2018;129:35-39. https://doi.org/10.1016/j.ipl.2017.09.004.
42. Broder A, Mitzenmacher M. Network applications of bloom filters: a survey. *Journal Internet Mathematics*; 2011;1/2004(4):485-509.
43. Goodrich MT, Mitzenmacher M. Invertible bloom lookup tables. Paper presented at: Proceedings of the 2011 49th Annual Allerton Conference on Communication, Control, and Computing (Allerton); 2011:792-799.
44. Sivaraman V, Narayana S, Rottenstreich O, Muthukrishnan S, Rexford J. Heavy-Hitter detection entirely in the data plane. *SOSR '17*. New York, NY: ACM; 2017:164-176.
45. Afek Y, Bremler-Barr A, Feibish SL, Schiff L. Detecting heavy flows in the SDN match and action model. *Comput Netw*. 2018;136:1-12. https://doi.org/10.1016/j.comnet.2018.02.018.
46. Mori T, Uchida M, Kawahara R, Pan J, Goto S. Identifying elephant flows through periodically sampled packets. *IMC '04*. New York, NY: ACM; 2004:115-120.
47. Estan C, Varghese G. New directions in traffic measurement and accounting: focusing on the elephants, ignoring the mice. *ACM Trans Comput Syst*. 2003;21(3):270-313. https://doi.org/10.1145/859716.859719.
48. Roughgarden T, Valiant G. Approximate heavy hitters and the count-min sketch; 2015. http://theory.stanford.edu/~tim/s15/l/l2.pdf. Online; accessed April 01, 2019.
49. Chaer A, Salah K, Lima C, Ray P, Sheltami T. Blockchain for 5G: opportunities and challenges. Paper presented at: Proceedings of the IEEE Global Communications Conference (GLOBECOM); 2019.
50. Salah K, Alcaraz CJM, Zeadally S, Al-Mulla S, Alzaabi M. Using cloud computing to implement a security overlay network. *IEEE Sec Priv*. 2013;11(1):44-53. https://doi.org/10.1109/MSP.2012.88.
51. Al-Haidari F, Sqalli M, Salah K. Impact of CPU utilization thresholds and scaling size on autoscaling cloud resources. Paper presented at: Proceedings of the 2013 IEEE 5th International Conference on Cloud Computing Technology and Science; vol 2, 2013:256-261.
52. Calyam P, Rajagopalan S, Seetharam S, Selvadhurai A, Salah K, Ramnath R. VDC-Analyst: Design and verification of virtual desktop cloud resource allocations. *Comput Netw* 2014; 68: 110 - 122. Communications and Networking in the Cloud https://doi.org/10.1016/j.comnet.2014.02.022

53. Bonola M, Bianchi G, Picierro G, Pontarelli S, Monaci M. StreaMon: a data-plane programming abstraction for software-defined stream monitoring. *IEEE Trans Depend Sec Comput*. 2017;14(6):664-678. https://doi.org/10.1109/TDSC.2015.2499747.
54. Guan B, Shen S. FlowSpy: an efficient network monitoring framework using p4 in software-defined networks. Paper presented at: Proceedings of the 2019 IEEE 90th Vehicular Technology Conference (VTC2019-Fall); 2019:1-5.
55. Ding D, Savi M, Siracusa D. Estimating logarithmic and exponential functions to track network traffic entropy in P4. Paper presented at: Proceedings of the IEEE/IFIP Network Operations and Management Symposium (NOMS); 2020.
56. Sanvito D, Moro D, Capone A. Towards traffic classification offloading to stateful SDN data planes. Paper presented at: Proceedings of the 2017 IEEE Conference on Network Softwarization (NetSoft); 2017:1-4.
57. Bianco A, Giaccone P, Kelki S, Campos NM, Traverso S, Zhang T. On-the-fly traffic classification and control with a stateful SDN approach. Paper presented at: Proceedings of the 2017 IEEE International Conference on Communications (ICC); 2017:1-6.
58. Capone A, Cascone C, Nguyen AQT, Sansò B. Detour planning for fast and reliable failure recovery in sdn with openstate. Paper presented at: Proceedings of the 11th International Conference on the Design of Reliable Communication Networks (DRCN); 2014: https://doi.org/10.1109/DRCN.2015.7148981.
59. Carmelo C, Davide S, Luca P, Antonio C, Brunilde S. Fast failure detection and recovery in SDN with stateful data plane. *Int J Netw Manag*. 1957-2017;27(2):e1957-e1957. https://doi.org/10.1002/nem.1957.
60. Cascone C, Pollini L, Sanvito D, Capone A. Traffic management applications for stateful SDN data plane. Paper presented at: Proceedings of the 2015 4th European Workshop on Software Defined Networks; 2015:85-90.
61. Boite J, Nardin P, Rebecchi F, Bouet M, Conan V. Statesec: stateful monitoring for ddos protection in software defined networks. Paper presented at: Proceedings of the 2017 IEEE Conference on Network Softwarization (NetSoft); 2017:1-9.
62. Afek Y, Bremler-Barr A, Shafir L. Network anti-spoofing with SDN data plane. Paper presented at: Proceedings of the IEEE INFOCOM 2017 - IEEE Conference on Computer Communications; 2017:1-9.
63. Febro A, Xiao H, Spring J. Telephony Denial of Service defense at data plane (TDoSD@DP). Paper presented at: Proceedings of the NOMS 2018-2018 IEEE/IFIP Network Operations and Management Symposium; 2018:1-6.
64. Lapolli AC, Adilson Marques J, Gaspary LP. Offloading real-time DDoS attack detection to programmable data planes. Paper presented at: Proceedings of the 2019 IFIP/IEEE Symposium on Integrated Network and Service Management; 2019:19-27.
65. Lall A, Sekar V, Ogihara M, Xu J, Zhang H. Data streaming algorithms for estimating entropy of network traffic. *SIGMETRICS Perform Eval Rev*. 2006;34(1):145-156. https://doi.org/10.1145/1140103.1140295.
66. Jiang L, Shah D, Shin J, Walrand J. Distributed random access algorithm: scheduling and congestion control. *IEEE Trans Inf Theory*. 2010;56(12):6182-6207. https://doi.org/10.1109/TIT.2010.2081490.
67. Shannon CE. A mathematical theory of communication. *Bell Syst Tech J*. 1948;27(3):379-423. https://doi.org/10.1002/j.1538-7305.1948.tb01338.x.
68. Lakhina A, Crovella M, Diot C. Mining anomalies using traffic feature distributions. *SIGCOMM Comput Commun Rev*. 2005;35(4):217-228. https://doi.org/10.1145/1090191.1080118.
69. Liu Z, Manousis A, Vorsanger G, Sekar V, Braverman V. One sketch to rule them all: rethinking network flow monitoring with univmon. *SIGCOMM '16*. New York, NY: ACM; 2016:101-114.
70. Li Y, Miao R, Kim C, Yu M. FlowRadar: a better netflow for data centers. Paper presented at: Proceedings of the 13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16). USENIX Association; 2016: 311-324; Santa Clara, CA.
71. Bhamare D, Jain R, Samaka M, Erbad A. A survey on service function chaining. *J Netw Comput Appl*. 2016;75:138-155. https://doi.org/10.1016/j.jnca.2016.09.001.
72. Medhat AM, Taleb T, Elmangoush A, Carella GA, Covaci S, Magedanz T. Service function chaining in next generation networks: state of the art and research challenges. *IEEE Commun Mag*. 2017;55(2):216-223.
73. Zhou D, Yan Z, Fu Y, Yao Z. A survey on network data collection. *J Netw Comput Appl*. 2018;116:9-23. https://doi.org/10.1016/j.jnca.2018.05.004.
74. Rashvand HF, Salah K, Calero JMA, Harn L. Distributed security for multi-agent systems - review and applications. *IET Inf Secur*. 2010;4(4):188-201. https://doi.org/10.1049/iet-ifs.2010.0041.
75. Mishra P, Varadharajan V, Tupakula U, Pilli ES. A detailed investigation and analysis of using machine learning techniques for intrusion detection. *IEEE Commun Surv Tutor*. 2019;21(1):686-728. https://doi.org/10.1109/COMST.2018.2847722.
76. Gonzales Pastana Lobato A, Andreoni M, OCM. B. Duarte Um sistema acurado de detecção de ameaças em tempo real por processamento de Fluxos. Paper presented at: Proceedings of the 36th Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos-SBRC; 2016.
77. García S, Grill M, Stiborek J, Zunino A. An empirical comparison of botnet detection methods. *Comput Secur*. 2014;45:100-123. https://doi.org/10.1016/j.cose.2014.05.011.
78. IP Blacklists: IP reputation feeds. Online; accessed August 14, 2019.

How to cite this article: Bonfim M, Santos M, Dias K, Fernandes S. A real-time attack defense framework for 5G network slicing. *Softw Pract Exper*. 2020;1–30. https://doi.org/10.1002/spe.2800