Efficient Service Chain Verification Using Sketches and Small Samples

Reuven Cohen Liran Katzir Aviv Yehezkel
Department of Computer Science
Technion, Haifa, Israel

***Abstract*—A service function chain defines an ordered or partially ordered set of abstract service functions and ordering constraints that must be applied to packet flows as a result of classification. Service chain verification is an important logic, which should verify that each flow indeed traverses the intended set of services, and in the desired order. In this paper we address this service chain verification problem in a new way. The main idea is to convert each service chain verification instance to its "equivalent set-expression cardinality equation," and to use statistical algorithms to verify that each equation is satisfied.**

I. Introduction

The delivery of end-to-end services often requires various service functions, such as firewalls and load balancers. Each service function can act at various layers of a protocol stack. It can be a virtual element or be embedded in a physical network element [18].

Service chaining [16] is the idea of routing a data flow through a sequence of network functions. When it is combined with SDN (Software Defined Networking), flow-based service chaining can be implemented without changing the physical topology. A service function chain defines an ordered or partially ordered set of abstract service functions and ordering constraints that must be applied to packet flows as a result of classification [18]. For example (Figure 1), the datacenter operator may decide to route all HTTP traffic through a service chain that consists of a firewall, an IDS (Intrusion Detection Server), and a proxy server, and to route all non-HTTP traffic through a different service chain that contains a firewall and an IDS.

With service chaining, incomplete or inconsistent software configuration could cause significant network and security problems. For example, due to misconfiguration, some flows that should traverse the firewall might not traverse it at all, or they might traverse it before, rather than after, traversing the IDS. Service chain verification is an important logic, which should verify that each flow indeed traverses the intended set of services, and in the desired order.

In traditional networks, there were very few possible data paths and very few possible service chains. Thus, service chain errors were rare. For example, all traffic is usually forwarded from the egress router to a nearby firewall. The traffic is then forwarded to an IDS, co-located with the firewall. However, with the virtualization of network functions, and in particular when it is combined with SDN, the problem of service chain verification is exacerbated by the ability to route every flow through any sequence of services. It is exacerbated even more by the requirement to have many copies of the same service in many servers that are located in different places of the network, due to load balancing considerations.

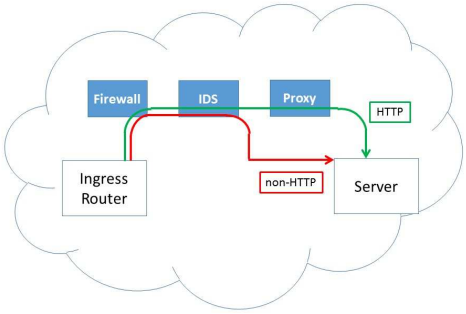


Fig. 1. An example of two service chains

One of the most challenging service chain verification problems is whether each flows is indeed routed along its intended sequence of services and in the intended order. Existing approaches for addressing this problem [3] are based on a small meta-data, referred to as "proof of transit," which is added to every packet. This meta-data should be updated by every service node along the path. It is finally received by a verifying controller, typically in the egress of the network, which verifies that each packet has indeed traversed its intended service chain. Cryptographic mechanisms are used to protect the meta-data from configuration mistakes and malicious attacks.

The above mentioned approach requires complex and expensive computations in every service node; it likewise requires that all the packets be forwarded to the verifying controller. Moreover, as indicated before, the datacenter operator does not always care whether a flow passes through a specific server, but whether it passes through one of many servers that perform the same function. Thus, we define the service chain verification problem as the problem of ensuring that each flow that enters the network through one of a given set of ingress nodes, traverses afterwards one of a given set of egress nodes. For example, verifying that every flow that enters the datacenter through router R_1 or R_2 or R_3 traverses afterwards one of the firewalls F_1 or F_2. By reapplying a procedure that solves this problem, we can verify that a subset of the flows

that traverse one of the firewalls continues to one of the load balancers, and that another subset continues to one of the IDSs, and so on.

In this paper we address this service chain verification problem in a new way. The main idea is to convert each service chain verification instance to its "equivalent set-expression cardinality equation," and to use statistical algorithms to verify that each equation is satisfied. The following example illustrates this idea. Suppose again that flows enter the datacenter via three routers, R_1, R_2 and R_3. Suppose also that the datacenter has two firewalls, F_1 and F_2, and that each flow must traverse one of them. Translating this requirement into a set-expression cardinality equation means that $|R \setminus F|$ should be equal to 0, where $R = R_1 \cup R_2 \cup R_3$ is the set of all incoming flows, $F = F_1 \cup F_2$ is the set of all flows traversing the firewalls, $R \setminus F$ is the set of flows that are in R but not in F (i.e., enter the datacenter but do not traverse any firewall), and $|R \setminus F|$ is the number of flows (cardinality) in this set.

Our verification scheme is briefly described in Figure 2. The datacenter operator determines a set of requirements to be verified. Each requirement is then translated into its "equivalent set-expression cardinality equation" (e.g., $|\{R_1 \cup R_2 \cup R_3\} \setminus \{F_1 \cup F_2\}| = 0$). Each participating network device (router, switch, server, etc.) samples a small number of the packets it receives. Its CPU periodically creates from the samples a sketch and sends it to the network controller. A sketch is a small fixed-size data structure, which summarizes the relevant information on the traversing data flows. For example, a max sketch maps each packet to a flow ID, and maintains the max flow ID of all the packets. From this information, the number of flows can be accurately approximated [4]. The network controller receives sketches from all relevant network devices, and verifies that the various set-expression cardinality equations indeed hold.

The main problem with sketch-based algorithms is their requirement for hardware support, because each received packet must be compared against the sketch [15]. However, since our scheme requires that each participating device will run the sketch only against sampled packets (e.g., 0.1% of the packets), this can be done by a CPU or a DPU[1], with no hardware support.

The rest of this paper is organized as follows. Section II discusses previous work. Section III presents the main framework of the proposed service chain verification schemes. Section IV presents the proposed scheme. Section V presents simulation results. Section VI discusses implementation issues and Section VII concludes the paper.

II. Related Work

The problem of service chain verification is known for several years [18]. In [11], a platform for service chain verification is proposed. This platform makes use of the virtual switches that are placed on physical cloud nodes. Service function chaining rules are stored within the flow tables of these virtual switches to gather the actual "Overlay and Traffic Steering" (SOTS) snapshot of the cloud environment, and model it as property-based graphs. Subsequently, these graphs can be used to verify if the actual SOTS is conform to the targeted service function chain specification.

[1]A DPU (Data Processing Unit) is a new class of programmable processor that can be found today in smart network interface cards (NICs).

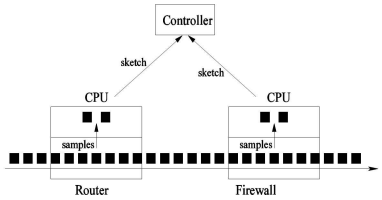


Fig. 2. The proposed verification scheme: the host's CPU or NIC's processes of every participating device processes only packet samples, and it sends to the controller only a sketch (summary) of these samples

In [19], the authors study the problem of verifying that certain performance guarantees, such as latency, packet loss rate and bandwidth, are satisfied when service function chaining is used.

Recently, [14] proposes the design and implementation of AuditBox, which (1) enables what you see is what you get auditing practices, and (2) enforces at runtime that the system operates correctly. To ensure that the correct network function software is running, AuditBox runs them atop hardware enclaves. By changing the trust model, verified routing is reformulated to audit actions between trusted network functions, with an untrusted network in between. To tackle dynamic packet modifications, immutable packetIDs are carried by an AuditBox packet trailer. This enables to logically bind modified packets to incoming packets when creating audit trails. Given trusted network functions, a simplified path attestation protocol is defined. The protocol focuses on the packets at individual network function hops. The use of trusted NFs inside enclaves enables a number of simple-yet-effective cryptographic optimizations such as the use of symmetric keys and updatable MAC computations to significantly improve data plane performance.

Network management requires accurate on-line estimation of various stream statistics, such as the number of flows ("cardinality"), flow size distribution, entropy, heavy hitters, etc. Finding the number n of distinct flows in a long stream of IP packets, known as "the cardinality estimation problem," is useful in numerous network monitoring and security applications. Several works address the cardinality estimation problem of a single stream [4], [7], [10] and propose min/max sketches for solving it. These algorithms are efficient because they make only one pass on the data stream, and because they use a fixed and small amount of storage.

Sketch-based algorithms are also useful for multiple streams [2], [6], [9]. An estimation of $|A \cup B|$, namely, the cardinality of $A \cup B$, can be easily found using any min/max sketch estimator for the cardinality estimation problem [12]. Such an

estimator considers $A \cup B$ as a single stream and estimates its cardinality. An estimation of $|A \cap B|$ can be found using the inclusion-exclusion principle [8]. In [2], [6], [9], it is proposed to estimate the Jaccard similarity and then use it to estimate the intersection cardinality. In [2] the estimators are generalized to set expressions between more than two streams.

III. Service Chain Verification Using Max Sketches

The challenge of processing large volumes of data that arrive at high speed has led the research community to develop new families of algorithms that work over continuous streams and produce accurate real-time estimations while guaranteeing: (a) low processing time per element, (b) fixed-size memory, which is sub-linear in the length of the stream, and (c) high estimation quality. The two main families are:

- Sub-linear space algorithms, also known as sketch-based streaming algorithms. These algorithms typically use a sketch, namely, a small fixed-size storage that stores a summary of the input data. Then, they employ a probabilistic algorithm on the sketch, which estimates the desired quantity.
- Sub-linear time algorithms. These algorithms are allowed to see only a small portion of the input. A common practice is to use sampling and process only the sampled stream elements.

The scheme presented in this paper satisfies both, i.e., it requires both sub-linear space and sub-linear time.

A. General Description

We start with a high-level description of the proposed scheme, as depicted in Figure 2. Let $V_1, V_2, \ldots, V_k$ denote the various service nodes (switches, routers, servers, etc.) in a service chain. The datacenter operator determines a set of requirements on the flows traversing these nodes and translates them into their "equivalent set-expression cardinality equations". Each participating node continuously samples packets traversing through it and copies these packets to a network processor, which periodically creates sketches and sends them to a centralized entity (a "service chain verification controller"). The controller uses the algorithm proposed in this paper to verify that the various cardinality equations indeed hold.

The basic building block in our scheme is to verify that all the flows that traverse V_1 also traverse V_2. To simplify the presentation, we start by assuming that V_1 is a single device and so is V_2. But we later show that each of them can be a set of devices as well (e.g., V_1 can be a set of ingress routers and V_2 a set of firewalls). Suppose that at the considered time interval, the packets seen by V_1 are: $a^1, b^1, b^2, a^2, c^1, d^1, d^2, c^2, a^3, b^3$, where "$a^i$" represents the ith packet of flow a. Thus, the flows traversing V_1 are $\{a, b, c, d\}$. Our goal is to verify that all these flows[2] also traverse V_2. This is equivalent to verifying that $|V_1 \setminus V_2| = 0$.

[2]To reduce the number of notations, we use the name of a device, e.g., V_i, to indicate also the set of flows traversing it.

An impractical approach to verify that $|V_1 \setminus V_2| = 0$ is to send all the packets of V_1 and V_2 to the verifying controller. It is even impractical to sample the packets traversing V_1 and V_2, and send only the sampled packets to the verifying controller. Therefore, in our scheme, sample packets are forwarded to the local processor. This processor periodically creates a sketch ("summary") of the sampled packets, and sends to the verifying controller only these sketches. Thus, as already indicated, our scheme **takes advantage of both sketching and sampling** and it is therefore very efficient.

The whole verification process is illustrated in Figure 3. As described in Section V, we found that sending a sketch of only $3,200$ bits by each participating device is sufficiently accurate for detecting service chain violations. Sketches are sent to the controller periodically, according to the required monitoring resolution.

B. A Building Block for the Proposed Service Chain Verification Scheme

We now show how to convert each service verification problem into its "equivalent set-expression cardinality equation". Consider again the general scenario from Section I: verifying that all the flows that enter through a given set $I = \{I_1, I_2, \ldots, I_p\}$ of ingress devices afterwards traverse one of multiple copies of egress devices from a set $E = \{E_1, E_2, \ldots, E_q\}$. Examples of this scenario include verifying that all the flows that enter the datacenter through routers in I traverse one of the firewalls in E, or verifying that after leaving the firewalls (now the set I), all the HTTP flows traverse one of the load balancers (E).

Proving that all the flows that traverse one of the devices in I also traverse one of the devices in E is equivalent to verifying that $I \setminus E = \emptyset$, and the latter is equivalent to verifying that $|\cup I_j \setminus \cup E_k| = 0$. The verifier of $|\cup I_j \setminus \cup E_k| = 0$ is used as a building-block of our scheme, and it is denoted $ESD(I \rightarrow E)$ (ESD stands for "empty set difference").

C. The Min/Max Sketch for Cardinality Estimation

A min/max sketch for cardinality estimation stores only the minimum/maximum hashed value of a packet header. When a packet of flow e_j is received, a hash function h is used in order to associate e_j with a uniform random variable, $h(e_j) \sim \mathrm{U}(0,1)$. The expected minimum value of $h(e_1), h(e_2), \ldots, h(e_n)$ is $1/(n+1)$. When only one hash function is used, the variance of the estimator is infinite. Thus, multiple different hash functions are usually used in parallel. In this case the estimator keeps the minimum/maximum value for each hash function, and then averages the results. The HyperLogLog algorithm [10] is the most practical approach today for solving the cardinality estimation problem. Its relative standard error is $1.04/\sqrt{m}$, where m is the number of hash functions (and sketches).

To minimize the number of packets that have to be processed, the proposed verification scheme processes only a small sample of the stream, compute their max sketches (or min sketch) and sends these sketches to the verifying

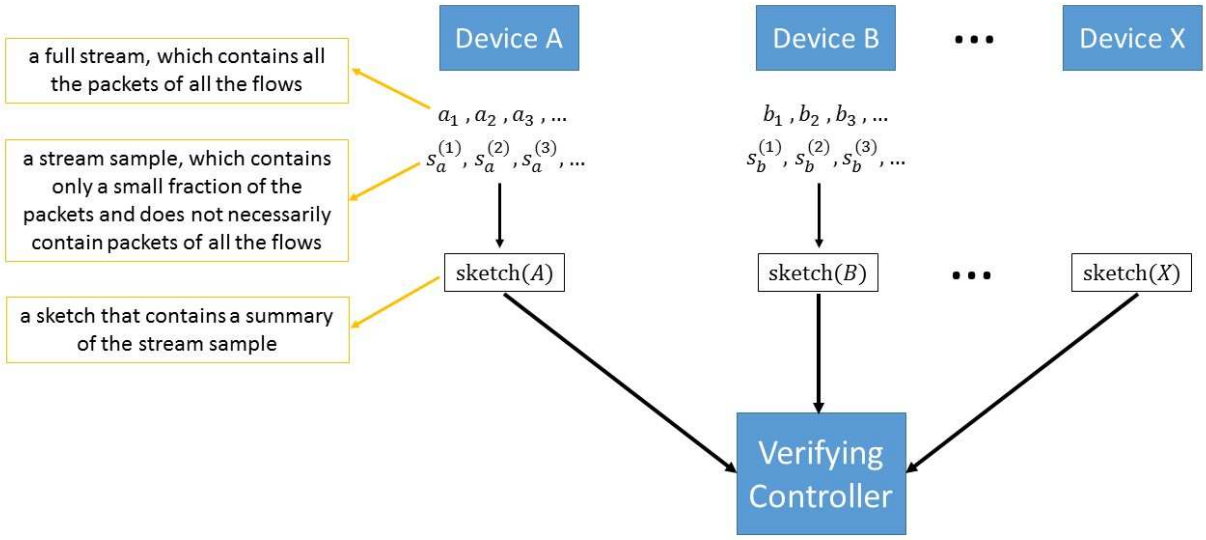


Fig. 3. The proposed verification scheme: each device samples the various packets, continuously creates sketches from the recently sampled packets, and sends these sketches to the controller. The controller receives sketches from all participating devices and verifies that the various cardinality equations indeed hold

controller. This is done in the following way. When a device receives a packet, it determines whether or not to sample it, according to the sampling rate. The header(s) of every sampled packet, which uniquely identify a flow ID, is copied to the host's or NIC's processor for local processing. The processor hashes the flow ID of each sampled packet to a uniformly distributed $\sim \mathrm{U}(0,1)$ random variable. It then computes the m max sketch of the received sampled packets, and continuously send these sketches, over a reliable (TCP) connection, to the verifying controller.

The following example illustrates the process of generating a max sketch. Suppose that the following packets are sampled and forwarded to the processor:

$$a^1, b^1, b^2, a^2, c^1, d^1, d^2, c^2, a^3, b^3,$$

where "a^i" represents the ith packet of flow a. Suppose also that $m = 2$. Suppose that for the first hash function h_1, we get that $h_1(a) = 0.11$, $h_1(b) = 0.82$, $h_1(c) = 0.64$, $h_1(d) = 0.55$. Thus, the max sketch for h_1, to be sent to the controller, is 0.82. Suppose that for the second hash function h_2, we get that $h_2(a) = 0.74$, $h_1(b) = 0.38$, $h_1(c) = 0.41$, $h_1(d) = 0.44$. Thus, the max sketch for h_2, to be sent to the controller, is 0.74.

IV. The Proposed Scheme

Suppose that we want to verify that all the flows that enter the datacenter through a set A of devices continue to one of the devices in set B, and that the flows that enter B continue to one of the devices in set C. This is translated into two instances of $ESD(I \rightarrow E)$ verification. In the first instance, $I \equiv A$ and $E \equiv B$, while in the second instance, $I \equiv B$ and $E \equiv C$. In this way, we handle a service chain of any length.

To verify $ESD(I \rightarrow E)$, namely that $|I \setminus E| = 0$, the following steps are taken:

(a) The nodes in I perform sampling, and compute the m sketches as explained above.

(b) The nodes in E compute the m sketches only on packets that traversed and were sampled by a node in I. These sketches must not consider packets that did not traverse a node in E or packets that traversed a node in E but were not sampled by that node.

(c) The sketches from the nodes in I and from the nodes in E are sent to the controller.

(d) For each of the m sketches, the controller views the maximum received from all the nodes in I as the sketch of the whole I set. Similarly, it views the maximum received from all the nodes in E as the sketch of the E set. For each of the m I sketches: (i) it estimates the number of flows that have traversed the I set and were sampled; (ii) it estimates the number of flows that have traversed the I set, were sampled and then traversed the E set; (iii) compare the two numbers. There is a violation if for any sketch, the two numbers are different.

There are several ways to enforce the requirements in (b) using Network Service Header (NSH)[17]. The simplest way is to use a bitmap in the NSH with a bit allocated to each

set of devices that implement the same function (A, B and C in our example). For the verification of $ESD(A \rightarrow B)$, when a packet traverse a node in A **and** it is sampled by that node, the node sets the A-bit in the packet's header bitmap. When the packet traverses a node in B, that node samples every packet whose A-bit is set. This concept is known as coordinated sampling [5].

Consider m hash functions[3], and let h_j^I denote the maximal hash values **of all the nodes in** I for the jth hash function. We use $h_j^{(E,I)}$ to denote the maximal hash value in the sketch maintained by all the E nodes for the traffic received from I and for the jth hash function.

Observation 1:
If there is no $ESD(I \rightarrow E)$ violation, $h_j^I = h_j^{(E,I)}$ holds for each hash function j.

This observation leads to the following detection algorithm:

Algorithm 1:

1) Let the m max sketches of I be: $h_1^I, h_2^I, \ldots, h_m^I$.
2) Let the m max sketches of E with respect to the packets received from I be: $h_1^{(E,I)}, h_2^{(E,I)}, \ldots, h_m^{(E,I)}$.
3) Return Violation if there exists j such that $h_j^I \neq h_j^{(E,I)}$.

By Observation 1, each violation of the j'th maximum indicates that the flow associated with h_j^I does not traverse E. A direct conclusion is that Algorithm 1 makes **no false detections**.

We now analyze the number of violating flows required for Algorithm 1 to detect the violation with a given probability, and also derive the required number m of sketches. Let n be the total number of flows traversing I, and let b be the number of violating flows, namely, flows that traverse I but not E. In addition, let n_s be the sample cardinality and b_s be the number of violating flows in the sample. For the analysis, we make the following assumption:

Assumption 1: The fact that f is a violating flow does not affect the probability that it will be sampled.

Note that this assumption does not require all the flows to have the same probability of being sampled, but that b_s/n_s is highly concentrated around b/n.

Lemma 2:
In order for Algorithm 1 to detect an $ESD(I \rightarrow E)$ violation with probability $1 - A$, the number b of violating flows needs to satisfy

$$b > n \cdot (1 - A^{1/m}).$$

Proof:
For the jth hash function, recall that h_j^I and $h_j^{(E,I)}$ are the maximal hash values in I and E respectively. Also, let e_j^I and $e_j^{(E,I)}$ be the flows associated with these maximal hash values, respectively. Assume that e_j^I is a violating flow, namely, a flow that traversed I but did not traverse E. From Observation 1 we obtain that $h_j^I \neq h_j^{(E,I)}$. Thus, the violation is detected by Algorithm 1. The probability that e_j^I will be a violating flow is b_s/n_s. Thus, the detection probability is

$$\begin{aligned}
&\Pr(\text{detection probability}) = \\
&= \Pr\left(\exists e_j^I \text{ such that } e_j^I \text{ is a violating flow}\right) \\
&= 1 - \Pr\left(\text{all } e_j^I \text{ are not violating flows}\right) \\
&= 1 - (1 - \frac{b_s}{n_s})^m. \qquad (1)
\end{aligned}$$

For the detection probability to be larger than $1-A$, we require that

$$1 - (1 - \frac{b_s}{n_s})^m > 1 - A,$$

or, equivalently, that

$$A > (1 - \frac{b_s}{n_s})^m.$$

This yields that

$$b_s/n_s > 1 - A^{1/m},$$

which, by Assumption 1, can be approximated by

$$b/n > 1 - A^{1/m}.$$

■

From Lemma 2 follows that in order to detect violation of b violating flows with probability $> 1 - A$, the number of maximal hash values should satisfy:

$$m = \frac{1}{\log_A(1 - \frac{b}{n})}. \qquad (2)$$

V. Simulation Study

In this section we validate the presented schemes. We use 2 real traffic traces taken from [1]. These traces were collected at the border router of the UCLA CS Department, and are summarized in Table I.

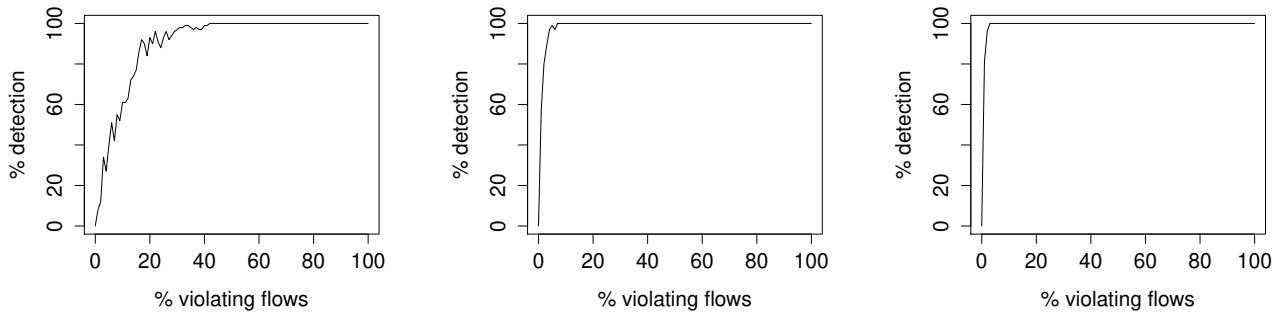
Trace Number	Number of Packets	Number of Flows (n)
Trace 1	283,063	8,396
Trace 2	553,361	173,014

TABLE I
REAL TRACES USED FOR OUR SIMULATIONS

We want to verify that all the flows that enter through a given set $I = \{I_1, I_2, \ldots, I_p\}$ traverse afterwards one of multiple copies of another set $E = \{E_1, E_2, \ldots, E_q\}$. We use the traces to represent the entire incoming traffic received via the p ingress devices. The total traffic represents n flows. These n flows are divided into p subflows, each representing the traffic entering through one ingress device. The division of n flows to p ingress devices is made randomly.

In each simulation test we choose $b \geq 0$ of the total n received flows to be violating flows; i.e., flows that do not traverse any egress device. When $b = 0$, there is no violation. The goal of the verifying controller is to detect with high probability and with minimal false detection probability all

[3]We show later how m sketches can be computed using only 2 hash functions, for every m.



(a) $m = 10$, $P = 0.001$ (b) $m = 100$, $P = 0.001$ (c) $m = 1000$, $P = 0.001$

Fig. 4. Detection percentages for Trace 1

the cases where $b > 0$. A false detection occurs when the verifying controller detects a violation although $b = 0$.

We start our study with $p = 5$ ingress and $q = 3$ egress devices. In each simulation test we first set the value of m. We then use 100 different values of b, equally spaced between 0 and n, i.e., $b \in \{0, n/100, 2n/100, \ldots, n\}$. For each value of b, we randomly choose b violating flows and apply the detection algorithm. We repeat this process 100 times, each with a different, randomly chosen, set of b violating flows. For each value of b we get a vector: $r_1^{(b)}, r_2^{(b)}, \ldots, r_{100}^{(b)}$, where $r_j^{(b)} = 1$ if the algorithm detects a violation in the j'th run, and $r_j^{(b)} = 0$ if not. We then average the results to obtain the detection probability for this value of b: $d^{(b)} = \sum_{j=1}^{100} r_j^{(b)}/100$. For the case where $b = 0$, the detection probability $d^{(0)}$ is actually the probability of false detection.

As expected, the false detection percentage is always 0, so it is not discussed any more. Figure 4(a)-(c) show the results of 1 for the first trace, and for three values of m: 10, 100 or 1000. For all these graphs, the sampling rate is $P = 0.001$. It is evident that $m = 10$ is not enough for a good detection probability. For example, for 20% violating flows and $m = 10$, the violation is detected with $\approx 90\%$. Increasing m to 100 brings the detection probability to $\approx 100\%$. We can also see that it is not needed to increase m beyond 100.

We repeated the same simulation with a sampling rate of $P = 0.01$, and obtain almost the same detection percentages as for $P = 0.001$. The results are not shown due to lack of space. This indicates that Algorithm 1 is only negligibly affected by the sampling rate.

Figure 5 shows the detection percentages for the second trace. As before, we use $P = 0.001$ and consider three value of m: 10, 100 and 1000. We see again that $m = 10$ is insufficient. Increasing m to 100 yields excellent results, and it is not needed to increase it further.

Table II compares the lower bound values of Algorithm 1 from the analysis (Lemma 2) to those obtained by the simulations. Recall that the lower bound is the minimal percentage of violating flows needed for Algorithm 1 to detect the violation with a given accuracy (probability of success). In the table we use $P = 0.001$. Two values of detection accuracy are used: 0.9 and 0.99, and three values of m: 10, 100 and 1000. We can see that the lower bounds found by our analysis are always very close to the actual values. We also obtain similar results for $P = 0.01$. Thus, we conclude that Lemma 2 can be used to determine the value of m required for obtaining a certain detection accuracy.

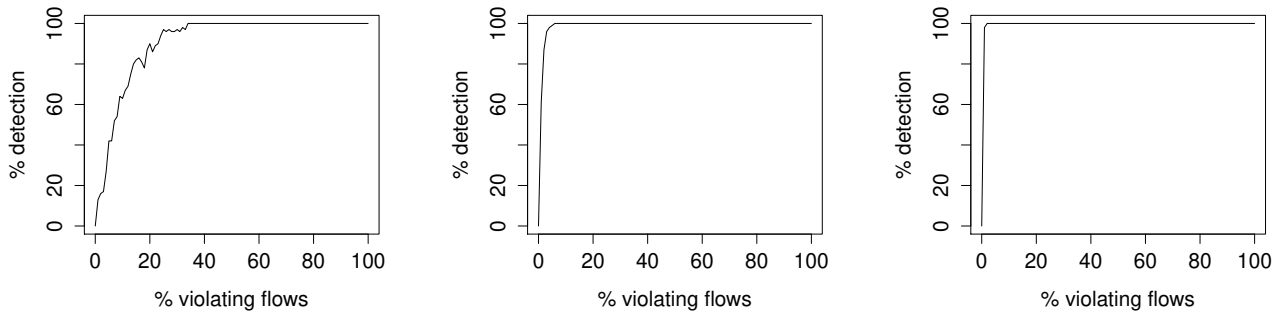
m	accuracy	analysis	Trace 1	Trace 2
10	0.9	20.6	18	20
	0.99	36.9	33	34
100	0.9	2.27	3	3
	0.99	4.5	5	5
1000	0.9	0.22	1	1
	0.99	0.46	1	2

TABLE II
COMPARING OUR LOWER BOUND'S ANALYSIS TO THE SIMULATIONS RESULTS OF ALGORITHM 1

VI. IMPLEMENTATION NOTES

It was shown that in order to get good accuracy, the algorithm needs to use m sketches, where $m \approx 100$. It is very expensive for the processor to implement each sketch using a different hash function and to hash every sampled packet 100 times. But in practice, the m sketches can be maintained using only two hash functions, h_1 and h_2. When a sampled packet is received by the processor, the first function is used for hashing the relevant header fields (e.g., source/destination IP addresses and source/destination port numbers) into a uniformly distributed variable that represents the flow ID. Then, the second hash function is used for splitting the flow ID into one of m buckets. These m buckets replace the need for m different hash functions. In addition, in order to increase computation speed, the estimator does not keep the value of the maximal sketch for each bucket, but rather only the highest position of the leftmost 1-bit of each sketch, i.e., the largest $i \geq 0$ such that the $i - 1$ leftmost bits are all 0.

Today, many virtualized servers are connected to the network by means of a new generation of intelligent network



(a) $m = 10$, $P = 0.001$ (b) $m = 100$, $P = 0.001$ (c) $m = 1000$, $P = 0.001$

Fig. 5. Detection percentages for Trace 2

interface cards (NICs), also known as smartNICs. These smartNICs can be implemented in many flexible ways to perform complex actions that save host's CPU cycles. A SmartNIC hosts a programmable engine that allows for the implementation and extension of the NICs functions, e.g., for offloading expensive operations from the host's CPU to the NIC. The core of the SmartNIC is a programmable element that can be a multicore system-on-chip (SoC), based on general-purpose CPU cores (mainly ARM cores)[13].

When a smartNIC is used by the considered network function, the proposed scheme can be implemented in the following way. The NIC samples the packets and forwards the headers of the sampled packets to the NIC's cores. The cores compute the 2 hash functions and the sketch for each sampled packet. The m sketches are maintained on the NIC's memory, and are periodically transferred to the controller, using RDMA. We believe that RDMA Unreliable Datagram (UD) is the best service type for these messages. Using this implementation, the proposed scheme does not impose any processing burden on the host's general purpose CPU.

VII. Conclusion

In this paper we studied the problem of service chain verification, namely, verifying that all the traffic that should traverse a specific service chain indeed follows the intended path. We presented a novel scheme for efficiently and accurately solving many aspects of service chain verification using a statistical algorithm that needs to observe only a small fraction of the traffic. The proposed scheme takes advantage of both sketching and sampling and is therefore very efficient in terms of its processing and bandwidth requirements. Simulation results, using real traffic traces, showed that the proposed scheme can detect chain violation with very good accuracy.

References

[1] UCLA traffic traces. https://lasr.cs.ucla.edu/ddos/traces/.

[2] K. S. Beyer, P. J. Haas, B. Reinwald, Y. Sismanis, and R. Gemulla. On synopses for distinct-value estimation under multiset operations. In *SIGMOD 2007*.

[3] F. Brockners et al. Proof of transit. IETF Internet-Draft, 2020.

[4] P. Clifford and I. A. Cosma. A statistical analysis of probabilistic counting algorithms. *Scandinavian Journal of Statistics*, 2011.

[5] E. Cohen and H. Kaplan. What you can do with coordinated samples. In *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques*, 2013.

[6] R. Cohen, L. Katzir, and A. Yehezkel. A minimal variance estimator for the cardinality of big data set intersection. In *KDD 2017*.

[7] R. Cohen, L. Katzir, and A. Yehezkel. A unified scheme for generalizing cardinality estimators to sum aggregation. *Inf. Process. Lett.*, 115(2):336–342, 2015.

[8] G. Cormode, M. N. Garofalakis, P. J. Haas, and C. Jermaine. Synopses for massive data: Samples, histograms, wavelets, sketches. *Foundations and Trends in Databases*, 4(1-3):1–294, 2012.

[9] T. Dasu, T. Johnson, S. Muthukrishnan, and V. Shkapenyuk. Mining database structure; or, how to build a data quality browser. In *SIGMOD Conference*, pages 240–251, 2002.

[10] P. Flajolet, É. Fusy, O. Gandouet, and F. Meunier. Hyperloglog: the analysis of a near-optimal cardinality estimation algorithm. In *Analysis of Algorithms (AofA) 2007*.

[11] M. Flittner, J. M. Scheuermann, and R. Bauer. Chainguard: Controller-independent verification of service function chaining in cloud computing. In *"IEEE NFV-SDN 2017"*, 2017.

[12] P. B. Gibbons. Distinct-values estimation over data streams. In *Data Stream Management - Processing High-Speed Data Streams*, pages 121–147. 2016.

[13] L. Linguaglossa et al. Survey of performance acceleration techniques for network function virtualization. *Proceedings of the IEEE*, 107(4), 2019.

[14] G. Liu et al. Don't yank my chain: Auditable NF service chaining. In *NSDI 2021*.

[15] M. Moshref, M. Yu, R. Govindan, and A. Vahdat. SCREAM: sketch resource allocation for software-defined measurement. In *CoNEXT 2015*.

[16] Z. A. Qazi, C. Tu, L. Chiang, R. Miao, V. Sekar, and M. Yu. Simplefying middlebox policy enforcement using SDN. In *SIGCOMM 2013*.

[17] P. Quinn, U. Elzur, and C. Pignataro. Network Service Header (NSH). RFC 8300, Jan. 2018.

[18] P. Quinn and T. Nadeau. Problem Statement for Service Function Chaining. RFC 7498, Apr. 2015.

[19] Y. Zhang, W. Wu, S. Banerjee, J.-M. Kang, and M. A. Sanchez. Slaverifier: Stateful and quantitative verification for service chaining. In *IEEE INFOCOM 2017*.