TRUSTSKETCH: Trustworthy Sketch-based Telemetry on Cloud Hosts

Zhuo Cheng
Carnegie Mellon University

Maria Apostolaki
Princeton University

Zaoxing Liu
University of Maryland

Vyas Sekar
Carnegie Mellon University

***Abstract*—Cloud providers deploy telemetry tools in software to perform end-host network analytics. Recent efforts show that sketches, a kind of approximate data structure, are a promising basis for software-based telemetry, as they provide high fidelity for many statistics with a low resource footprint. However, an attacker can compromise sketch-based telemetry results via software vulnerabilities. Consequently, they can nullify the use of telemetry; e.g., avoiding attack detection or inducing accounting discrepancies. In this paper, we formally define the requirements for trustworthy sketch-based telemetry and show that prior work cannot meet those due to the sketch's probabilistic nature and performance requirements. We present the design and implementation TRUSTSKETCH, a general framework for trustworthy sketch telemetry that can support a wide spectrum of sketching algorithms. We show that TRUSTSKETCH is able to detect a wide range of attacks on sketch-based telemetry in a timely fashion while incurring only minimal overhead.**

I. INTRODUCTION

Cloud providers (*e.g.,* AWS [2], Microsoft Azure [14] and GCP [8]) rely on network telemetry for management tasks such as anomaly detection [79], billing [17], and traffic engineering [33], [46]. Traditionally, network monitoring relied largely on telemetry capabilities installed on network hardware (e.g., routers and programmable switches). Increasingly, however, we find that monitoring traffic in software on end hosts is becoming a more attractive alternative thanks to its immediate deployability and flexibility. Moreover, with the recent announcements on reduced support for programmable router hardware (e.g., the discontinued Tofino programmable switch product line at Intel [11]), end-host telemetry capability will likely play an even more vital role.

In this context, sketch-based telemetry [55], [67], [68], [91] has been shown to be particularly promising for monitoring traffic in end-host software. In contrast to capturing full packets or flows, sketches are a class of probabilistic data structures that estimate traffic statistics online (*e.g.,* tracking heavy hitters [36], [39], [74], [85], measuring entropy [62], counting distinct flows [30]) with low resource footprints. Recently, sketches have been integrated into popular network packet libraries; for example, NitroSketch [19] has been used in Intel DPDK [41].

Existing work on sketch-based telemetry in end hosts largely considers a benign setting and overlooks the likelihood of the telemetry results being compromised by an attacker. For example, consider a cloud telemetry deployment as shown in Fig. 1, where cloud providers offer virtual machines to customers and run sketches on the hypervisor to monitor network traffic on the end host. Malicious cloud users may obtain privilege escalation, access to hypervisor, and virtual switches by exploiting existing software vulnerabilities in the data center (*e.g.,* SaltStack [9], VMware ESXi [20], Xen [22]).

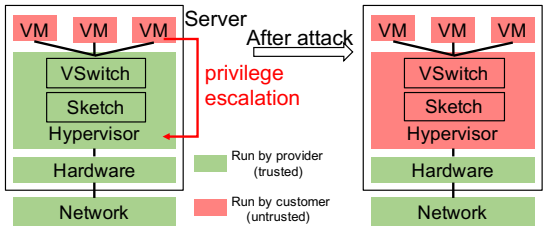


Fig. 1: *System Model: a cloud provider deploys sketch-based telemetry in the hypervisor to monitor network traffic of tenants' VMs. A malicious customer could get privilege escalation and get access to the hypervisor to compromise the telemetry results.*

Using such exploits, a malicious cloud attacker can compromise the telemetry results in one of three possible ways: *(i)* modifying the sketch execution logic; *(ii)* modifying the relevant memory regions that store the sketch counters; or *(iii)* injecting or dropping packets, or modifying packet headers in the memory. This is especially worrisome as disabling or corrupting sketch-based telemetry gives adversaries extraordinary power to launch novel attacks that can go unnoticed for extended periods of time; i.e., because the telemetry results are corrupted downstream anomaly detection or alerting systems are rendered ineffective. For example, attackers may be able to launch new attacks on other VMs or hosts without being detected.

An ideal trustworthy software sketch framework must satisfy three requirements:

- *Correctness*: The framework must meet three integrity requirements (formally defined in §III-A). Concretely, the sketch memory is updated exclusively (memory integrity) by the packets that traverse the network (input integrity) according to the predefined sketch logic (compute integrity).
- *Generality*: Operators might deploy different sketches for different telemetry tasks, so the framework needs to support a wide range of sketches.
- *Performance*: The framework must incur minimum throughput degradation and additional latency with minimal impact on resource usage (computation, memory, and network) and minimal hardware modifications.

Unfortunately, existing solutions and seemingly natural extensions cannot satisfy these requirements. For example, code

Network and Distributed System Security (NDSS) Symposium 2024
26 February - 1 March 2024, San Diego, CA, USA
ISBN 1-891562-93-2
https://dx.doi.org/10.14722/ndss.2024.23065
www.ndss-symposium.org

attestation [49] cannot prevent an attacker from changing the computation and memory of the sketch at run time, failing the correctness requirement. Techniques such as mirroring traffic to the cloud controller [80] to cross-validate the data-plane telemetry would double the traffic and would be impractical, failing the performance requirement.

In this paper, we present the design and implementation of TRUSTSKETCH. Our design uses secure enclaves (*e.g.,* Intel SGX [71]), which provide a private region of memory for secure computation and are already deployed in today's data center [4], [5]. Given the limited size of enclave memory, we carefully partition the telemetry system (consisting of a sketch, a virtual switch, and an I/O module which is a connection between network applications and the NIC) and only place the sketch in the secure enclaves to achieve runtime computation and memory integrity.

However, only using the enclave cannot provide the input integrity: packets go through untrusted host memory between the NIC and the enclave, so an attacker could corrupt the packets monitored by the sketch. To solve the problem, we consider the NIC as the second point of trust. This is a reasonable design choice because the NIC is standalone hardware and has a smaller and more stable code base, which means a smaller attack surface. Our solution uses the NIC to verify that the packet sequence between the enclave and the NIC is unchanged: the enclave and the NIC would periodically compute a hash of the packets they saw based on a shared secret key. In the evaluation, we show that this solution only uses a small resource footprint of the NIC.

We solve two practical challenges to make TRUSTSKETCH feasible. First, we need to decide how often to compute the hash. Simply computing one hash for each packet would incur high computation overhead and greatly degrade throughput. Computing hash for a predefined number of packets would fail to detect an attack in time because the packet rate is changing dynamically. We run timers at the enclave and the NIC, which times off periodically to divide packets into epoch and calculate a new hash for each epoch. Second, modern NICs use multithread to process packets in parallel to achieve high throughput. Each NIC thread would only process a substream of the packets, so we need a way to match the packet stream between the NIC and the enclave. To solve the problem, we append the packet sequence number as a tag to the packet, and reconstruct the packet substreams at the enclave and the NIC based on the tag. Then we compute one hash for each packet substream.

We validate the correctness of the design TRUSTSKETCH using a qualitative case-by-case analysis and a formal model based on a system modeling language Alloy [58]. We build an end-to-end prototype of TRUSTSKETCH using Intel SGX and Netronome SmartNIC [15]. We run our prototype against multiple realistic attack scenarios and demonstrate that it successfully and timely detects the attacks without any prior knowledge of the attacker's strategy. We run TRUSTSKETCH with eight sketches to protect five distinct telemetry tasks and show that TRUSTSKETCH adds 7% overhead in terms of throughput and 6μs to overall latency. We also show that TRUSTSKETCH only uses 1.9% CPU and 0.53% memory resources of the NIC, which means that it is promising to be implemented on commodity NICs.

II. BACKGROUND & MOTIVATION

In this section, we first discuss the cloud telemetry model that we consider. Next, we give some background on sketches. Finally, we present our threat model and show how an attacker could compromise sketch-based telemetry.

A. Setting: Cloud Network Telemetry

We consider a public cloud environment in which providers (*e.g.,* AWS [2], Microsoft Azure [14], and GCP [8]) operate physical servers interconnected by network fabrics in data centers and offer infrastructure-as-a-service (IaaS) to their customers. Through virtual resource orchestration platforms (*e.g.,* OpenStack [16] and Kubernetes [13]), cloud tenants have access to custom VM instances[1] on physical machines. This setting is practical today. Although offloading the network stack and the hypervisor to SmartNICs is a promising direction, deploying hypervisors with telemetry capability on the CPU is still mainstream due to the benefits of easy programming and fewer hardware changes. For example, AWS runs Firecracker [23] as a software hypervisor for the Lambda service. Azure NIC deployment uses the RDMA capability [29] between servers while the virtual network is still on the CPU. Therefore, our work focuses on software-based telemetry that is deployed on host servers (*i.e.,* on software switches).

Such a telemetry infrastructure typically obtains packet header information (e.g., source and destination IPs, ports, and protocols) and aggregates them as flows to compute statistics (e.g., entropy [75], distinct flow [28], [30], heavy hitters [36], [40]). Operators then use the statistics to perform various management tasks, such as DDoS detection [30], [68], traffic engineering [33], [46], anomaly detection [79], and accounting [17].

B. Background on Sketches

Research efforts in network telemetry [57], [67]–[69], [88], [89] have proposed to run sketches on software switches. At a high level, sketches are compact probabilistic data structures that can provide accurate flow-level statistics and use low computational and memory resources [53], [67]. Thus, the cloud provider can integrate sketches into the data plane in virtual switches [53], [67] (e.g., Open vSwitch [77], VPP [45]) and obtain traffic statistics online, as shown in Fig. 1.

To show how the basic structure of sketches works, we use the popular Count-Min Sketch [40] as an example as shown in Fig. 2. The Count-Min Sketch model captures a general model for a wide range of sketches [31], [36], [68]. While different sketching algorithms differ in specifics of how they maintain counters and report estimates, the general structure follows this similar pattern.

At a high level, Count-Min Sketch maintains a 2D array of counters (e.g., r rows and d counters per row). When a packet arrives, the sketch extracts the flow key from the packet header (*e.g.,* source IP address or 5-tuple) and uses the flow key to compute r independent hash values. These r hash values will then be used as indices to decide which counters to increment ($+1$ in this case) in each row. Thus, the minimum of the r counters is an estimate of the total number of packets in this flow. The Count-Min Sketch also maintains a heap to track the Top-K largest flows. Using this sketch, one can accurately

[1]Our work also applies to containerized deployments. For clarity, we focus on VMs in the rest of the paper.

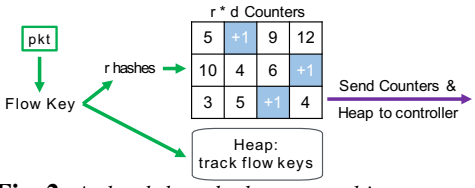


Fig. 2: *A sketch-based telemetry tool is composed of a 2D counter array and a heap.*

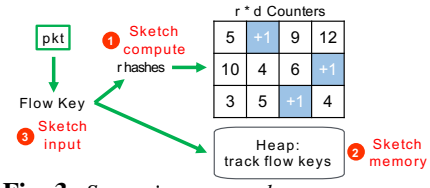


Fig. 3: *Strategies an attacker can use to compromise the sketch telemetry results.*

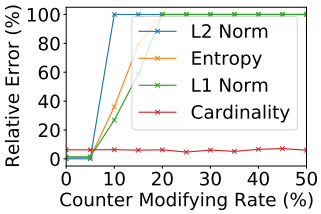


Fig. 4: *An attacker could increase the telemetry error to 100% by modifying 20% of the counters.*

estimate the heavy hitters in the traffic for each monitoring window. At the end of each monitoring interval, the sketch can report the *Counters* and the *Heap* to the controller, which can be used for other tasks (*e.g.,* DDoS detection and accounting).

C. Threat Model and Attacks

Goals We consider an attacker whose *goal* is to compromise the integrity of the telemetry results for their benefit. For instance, an attacker using multiple cloud VMs to launch a DDoS attack against another tenant could try to remain undetected by preventing telemetry from catching her VMs' traffic to the victim. Similarly, an attacker using the cloud could try to reduce her network costs by altering the telemetry to impact the cloud's accounting.

Our goal is to provide accurate and trustworthy telemetry results even in the presence of potential attackers. Although our solution does not directly mitigate such attacks, it provides a way to detect anomalous behaviors promptly and reliably, which facilitates subsequent forensic analysis and mitigation efforts.

Capabilities We consider an attacker who gets privilege escalation and controls the software stack of a server, including a hypervisor, virtual switch, and virtual machines. While strong, the threat model is practical: a VMware hypervisor vulnerability [20] (*i.e.,* uninitialized stack memory usage in the virtual network adapter) has been used to escape the VM and get a root shell on the host. In addition, there are other software vulnerabilities in the data center that allow the attacker to get privilege escalation, such as SaltStack [9] and Xen [22]. We assume that the attacker cannot control the enclave (a secure memory region) and the SmartNIC due to the hardware security features provided by the vendors (more details in §IV-A). We do not consider side-channel attacks since our work focuses on protecting the integrity of the telemetry results rather than the confidentiality. Our threat model is equivalent to prior work in the cloud computing space (e.g., [42], [66], [76], [78]).

Illustrative attacks. To make our discussion concrete, we next describe three exemplar attacks.

Runtime Compute Attack (① in Fig. 3): An attacker could modify the runtime execution logic of the sketch (*e.g.,* by modifying the runtime libraries). As a concrete illustration, Fig. 4 shows the impact on the accuracy of the UnivMon sketch when an attacker modifies counter update logic by updating a fraction of counters to random numbers. The accuracy degrades significantly; e.g., the error can be as high as 100% even when only 20% of the counters are modified.

Memory Attack (② in Fig. 3): An attacker could also modify the sketch data in memory. For example, he could read the memory mapping table maintained by the operating system to know the exact memory address of the sketch heap and remove the relevant flow key of the attack flows. We find that the error can be as high as 40% even when only 20% of the entries are dropped (figure not shown).

Input Attack (③ in Fig. 3): An attacker could inject packets into the packet buffer maintained by the virtual switch, which stores packets that are sent from the VMs, have already been processed by the sketch and are waiting to be sent out by the NIC. In this case, the injected packets are sent out by the NIC, but are not seen (monitored) by the sketch. We find that the error can be as high as 40% even when only 5% packets are injected (figure not shown).

Takeaways. While sketch-based telemetry running in cloud virtual switches is promising due to performance and low footprint, existing solutions are designed in a benign non-adversarial setting. Consequently, an attacker can impact the integrity of telemetry results to enable further downstream goals (*e.g.,* avoid detection).

III. Requirements and Existing Solutions

Given the threats to sketch-based telemetry, we identify the security requirements that *any* trustworthy sketch-based telemetry must strive to achieve. Then, we discuss two practical system requirements. Finally, we explain why existing solutions cannot meet these requirements in our context.

A. Security Requirements

Suppose that in an ideal world, the sketch would output $Counters$ and $Heap$ to the controller, and in an unsecured world, the sketch would output $Counters'$ and $Heap'$ respectively. Our overarching goal is to achieve tamper evidence, meaning that one of two conditions occurs: Either *(i)* $Counters' = Counters$ and $Heap' = Heap$, or *(ii)* raise an alert if $Counters' \neq Counters$ or $Heap' \neq Heap$. In other words, we ensure that the adversary cannot tamper with the telemetry results, or if they do, the tampering can be immediately detected. To achieve this, we identify and formally define three necessary security requirements[2].

Sketch-compute-integrity. Since the attacker might manipulate the sketch computation procedure (*i.e.,* changing how the sketch updates the counters and the heap) , ① in Fig. 3), we need to ensure that the execution of sketches is not corrupted by the attacker at run-time. Specifically, for each input packet p, the sketch executes instructions I_{hash} to compute the hash, $I_{counter}$ to update the counter, and I_{heap} to update heap.

[2]As we will see, these are independently necessary as removing any of them will introduce opportunities for correctness violations.

Similarly in an unsecured world, I'_{hash}, $I'_{counter}$ and I'_{heap} are executed. We need to guarantee that

$$\begin{aligned} I_{hash} &= I'_{hash} \\ I_{counter} &= I'_{counter} \\ I_{heap} &= I'_{heap} \end{aligned} \tag{1}$$

or raise an alert when Eq. 1 does not hold.

Sketch-memory-integrity. Even if we guarantee Sketch-compute-integrity, an attacker can still use the memory attack (*i.e.,* modify the sketch counters or the heap , ② in Fig. 3) . Assume in an ideal world, given the memory update instructions I_{hash} and $I_{counter}$, the sketch will update the counters from $Counters_{before}$ to $Counters_{after}$ and update the heap from $Heap_{before}$ to $Heap_{after}$. Assume that in an unsecured world, the sketch will update the counters to $Counters_{after}{}'$ and update the heap to $Heap_{after}{}'$. We need to guarantee that

$$\begin{aligned} Counters_{after} &= Counters_{after}{}' \\ Heap_{after} &= Heap_{after}{}' \end{aligned} \tag{2}$$

or raise an alert when Eq. 2 does not hold.

Sketch-input-integrity. Even if we guarantee Sketch-compute-integrity and Sketch-memory-integrity, an attacker could still launch an input attack (③ in Fig. 3). Thus, we need an *sketch-input-integrity* to ensure that the sketch monitors every packet in order sent and received by the NIC.

We define the sketch input integrity requirement by taking two observations into account. First, some complex sketches [25], [52], [69] depend on packet order, so we want to guarantee that the sketch monitors the same sequence of incoming (outgoing) packets as the NIC receives (sends). Second, since sketches work based on packet header, we only need to check the packet header. For a sequence of packets P, we use $P[i].hdr$ to denote the header of the i-th packet in that sequence. We use $S_{in}, S_{out}, N_{in}, N_{out}$ to denote the incoming and outgoing packet sequence seen by the sketch and the NIC. Then, we need to ensure that

$$\begin{aligned} S_{in}[i].hdr &= N_{in}[i].hdr, \forall i \in N^+ \\ S_{out}[i].hdr &= N_{out}[i].hdr, \forall i \in N^+ \end{aligned} \tag{3}$$

or raise an alert when Eq. 3 does not hold. For N_{in}, N_{out}, we do not consider non-data packets (*e.g.,* control packets between NIC and controller).

B. System Requirements

In addition to security requirements, we also need:

Performance. With today's high link speeds, any telemetry system may experience a large volume of packets, so we need to support high processing throughput and low latency. Also, we want to maximize the (computation, memory and network) resource available to the customers and maintain low resource usage for the telemetry system.

Generality. We can consider creating solutions that use particular algorithmic properties of specific sketches for robustness. However, this would not generalize well across sketches and will restrict operators who often deploy different types of telemetry goals. Thus, our system needs to be agnostic to the specifics of sketch algorithms.

Solution	Cross Checking	Software Attestation	Secure Memory
Compute Integrity	✓	×	×
Memory Integrity	✓	×	✓
Input Integrity	✓	×	×
Performance	×	✓	✓
Generality	✓	✓	✓

TABLE I: *Summary of Strawman Solutions and Limitations.*

C. Existing Solutions and Limitations

Next, we discuss why seemingly natural strawman solutions from the trustworthy computing literature can not meet our requirements. We summarize our discussion in Table I.

First, we can cross check results by running the sketch with the same input at different locations and comparing the results to verify the telemetry result. While such an approach could satisfy our security requirements, it incurs high network and compute overhead and thus fails the performance requirement. For example, we could mirror traffic via the NIC (*e.g.,* up to 100Gbps) to another location (*e.g.,* a separate server or controller) [80], incurring prohibitively large computation and network overheads. We can reduce these overheads by mirroring only a subset of sampled packets. However, since sketches are probabilistic data structures (*i.e.,* the same input can generate different outputs at different times), precisely reconstructing the telemetry result with partial input (*i.e.,* sampled packets) is hard and we cannot distinguish inaccuracies in reconstruction vs. malicious behavior.

Second, we could consider *code attestation* techniques (e.g., [49]) to verify the integrity of the sketch code at boot time. However, code attestation cannot provide runtime *sketch-compute-integrity* as it is only a static checker that verifies the code before execution. Thus, attested sketch code can still be interrupted or corrupted during the runtime execution (*e.g.,* by modifying runtime libraries).

Third, we could leverage secure memory features such as AMD Secure Memory Encryption (SME) [3] to provide *sketch-memory-integrity*. SME leverages specialized hardware to encrypt data before writing it to the main memory. We could calculate and store MAC (message authentication code) along with the data in memory to ensure its integrity. However, *sketch-memory-integrity* alone is not sufficient for trustworthy telemetry as we still need to provide *sketch-compute-integrity* and *sketch-input-integrity*.

IV. Design Overview

In this section, we first discuss two roots of trust for ensuring the telemetry integrity: secure enclave and SmartNIC. Then, we explore the design space of leveraging these hardware bases and explain why some strawman designs fail. Finally, we describe our design choices.

A. Roots of Trust

Our design is based on two practical roots of trust that are available on modern server hardware.

An enclave is an integrated CPU feature that allocates a private region of memory for isolated runtime execution. Software running outside the enclave (including privileged software such as kernel or hypervisor) is not able to read or modify computation and data in the enclave. Enclaves have already been deployed in today's data centers [4], [5].

We impose restrictions on attacker capabilities so that an attacker cannot read or modify the code and data in the enclave,

similar to prior work [42], [66], [78]. We acknowledge that powerful attackers could use side channels in the enclave (*e.g.,* page fault [86], cache timing [35], [50], branch prediction [63]) to compromise the confidentiality of data stored inside the enclave. Note, however, that our work focuses on *integrity* and thus such side channels that impact confidentiality are an explicit non-goal for our work.

A SmartNIC is a programmable network interface card that can run custom code to process packets, such as network functions (*e.g.,* NAT) and network protocol offloading (*e.g.,* TLS, VXLAN). Many types of SmartNIC have already been deployed in today's data centers [47].

We consider the NIC to be trusted *i.e.,* an attacker cannot read or modify the code and data on a SmartNIC. This design choice is in line with prior work [76] and has several justifications. First, the NIC code could be signed by the operator and verified by the NIC before execution. Additionally, the NIC operates as standalone hardware with its own firmware, which is distinct from the hypervisor code. As a result, an attacker would need to discover separate vulnerabilities specific to the NIC to compromise its security. Moreover, the firmware is updated less frequently and typically has a smaller code base, reducing the potential attack surface.

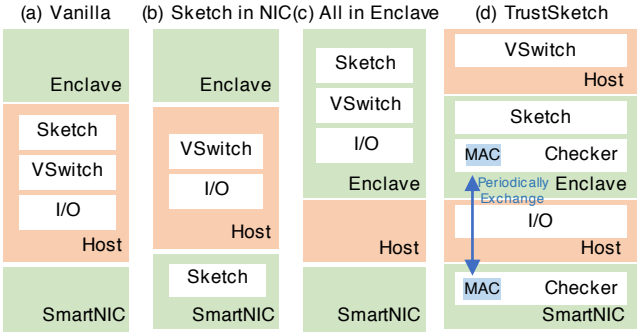
Constraints. Although trusted, both the enclave and SmartNIC have constraints in resources and functionality. For enclave, we consider the second generation of Intel SGX (SGXv2) [43], the latest enclave supported by Intel. Any data transferred between the enclave and the NIC must go through the unprotected host memory. This problem still exists for the next-generation enclave (*e.g.,* Intel TDX [12]). Moreover, instructions that change privilege levels (*i.e.,* syscall and int instructions) are not supported inside an enclave.

Commodity SmartNICs have limited computing and memory resources. They often only allow a limited number of instructions and memory accesses for each packet to support high line-rate packet processing. Also, the NIC resources are shared by other offloaded network functions (*e.g.,* NAT, TLS). So we want to use minimum NIC resources for network telemetry. Although some high-end SmartNICs (*e.g.,* Mellanox BlueField [72]) have more resources, a general framework is required to work with the commodity SmartNIC of limited resource.

B. Design Space

In general, a sketch-based telemetry system includes a virtual switch, a sketch, and an associated I/O module. The virtual switch in software is responsible for emulating the network switch function for the VMs, thus implementing operations such as forwarding and NAT. The sketch, as we describe in §II-B, is responsible for telemetry and contains a 2D counter array and a heap. The I/O module (*e.g.,* Intel DPDK [41]) is a software component that acts as a connecting bridge between network applications and the NIC. The I/O module manages packet buffers on the server and interacts with the NIC driver to send and receive packets.

Given these components, we can explore the design space of options that considers where these components could run in our server stack : host, enclave, or NIC. For example, the virtual switch and the sketch could run in the host memory, the enclave, or the NIC. Similarly, the I/O module could run in



Solution	(a)	(b)	(c)	(d)
Compute Integrity	×	✓	✓	✓
Memory Integrity	×	✓	✓	✓
Input Integrity	×	✓	×	✓
Performance	✓	×	✓	✓
Generality	✓	×	✓	✓

Fig. 5: *While various solution are possible in our design space, only* TRUSTSKETCH *satisfies all requirements.* (a) *the sketch-based telemetry as done today;* (b) *only the Sketch in the NIC as in [24], [91];* (c) *all related components in the enclave as in [78];* (d) TRUSTSKETCH *places only the sketch in the enclave and implements a input validation that runs between two checkers.*

the host memory or in the enclave. Note that the I/O module cannot fully run on the NIC because it interacts with drivers to support other software network applications.

Different design choices naturally entail trade-offs between security, performance and generality. To see why, consider two points from this design space. One design choice is to run the sketch on the NIC to achieve the three security requirements as we illustrate in Fig. 5(b). While it might be feasible to run some simple sketches on a commodity NIC, such as CountMin Sketch [40], NIC's resource constraints are typically prohibitive for real sketches. For instance, UnivMon [68] needs to update multiple counters and maintain multiple priority queues, resulting in an unbearable resource load for the NIC. Similarly, Hydra [70] uses up to tens of MBs for multidimensional telemetry, which are not available in typical commodity NICs. Also, virtual switch might update some packet header fields (*e.g.,* Network Address Translation). Only offloading the sketch to the NIC would cause the sketch to get incorrect packet header fields. Offloading the virtual switch to the NIC could solve the context problem, but the virtual switch would consume lots of computation and memory resource of the NIC.

An alternative design choice would be to run all three components in the enclave as we illustrate in Fig. 5(c). However, this would fail the *sketch-input-integrity* because packets go through unprotected host memory between the enclave and the SmartNIC and could be corrupted. Second, the original virtual switch and I/O module heavily rely on system calls that are not supported by the enclave. To address this limitation, prior work, such as SCONE [27], Graphene-SGX [37], have introduced shielded execution frameworks that enable the execution of unmodified applications inside the enclave. However, these frameworks do not provide enough low-level networking support [60] to run the I/O module inside the enclave. Furthermore, if we simply place all components inside the enclave, it would significantly increase the size of the trusted computing base (TCB) by a factor of 99x, as demonstrated in §VII-B.

C. Our Design: TRUSTSKETCH

Taking into account the constraints of the enclave and the NIC, we design TRUSTSKETCH to navigate the trade-off across security, performance, and generality.

TRUSTSKETCH provides compute and memory integrity by running only the sketch in the enclave. We observe that while trustworthy telemetry incorporates multiple components, only the sketch computation logic and the memory are crucial for the telemetry result. We also recognize a unique opportunity: the sketch has a small memory footprint and uses simple operations *e.g.,* hashing, additions; thus, it could fit in the resource budget of an enclave. Hence, we only put the sketch in the enclave and leave the virtual switch and the I/O module (*i.e.,* DPDK [41]) in host memory. The sketch in the enclave get access to packets managed by the I/O module through pointers to avoid packet copy (§VII-B). This choice provides the *sketch-compute-integrity* and *sketch-memory-integrity*, and minimizes enclave memory usage. Also, running the whole sketch in the enclave makes our solution general as one does not need to modify any sketch code.

TRUSTSKETCH provides input integrity by validating the I/O using the SmartNIC. While the enclave provides the *sketch-compute-integrity* and *sketch-memory-integrity*, it does not support direct and secure access to the NIC's memory, thus cannot provide *sketch-input-integrity*[3] (similar to Fig. 5(c)). Common solutions such as adding a digest in the each packet header [56] or constructing sketches in each end and comparing them [65], [90] do not offer *sketch-input-integrity* as they do not detect packet reorderings and deteriorate performance by incurring significant overhead and/or involving the controller.

To provide *sketch-input-integrity* at scale, TRUSTSKETCH leverages a domain-specific observation: re-orderings and packet drops cannot occur unless there is an attacker. Indeed, since there are not multiple paths between the enclave and the NIC, reordering are not possible. To avoid drops caused by congestion between the enclave and the NIC, the fastest component, namely, the NIC can be rate-limited to match the speed of the enclave.

Driven by these insights, TRUSTSKETCH runs a simple-yet-effective input validation with two checkers: one at the enclave and one at the NIC. The checkers would compute a message authentication code (MAC) over the packets they have seen and periodically exchange the MAC. To prevent the attacker from re-calculating the MAC, TRUSTSKETCH uses a shared key in the MAC that is negotiated between the enclave and the NIC at bootstrap stage. To make TRUSTSKETCH react independent of the traffic rate, the validation process is directed (separate for incoming and outgoing traffic) and time-triggered.

V. ENSURING INPUT INTEGRITY IN TRUSTSKETCH

At a high level, TRUSTSKETCH uses well-known best practices in using enclaves for compute and memory integrity and we defer specific implementation details to §VII. In this section, we focus on the harder problem of ensuring input integrity. In particular, we need to tackle practical challenges in synchronizing exchange of traffic summaries between the enclave and NIC and ensuring correct operation in the presence of multi-threaded operation, which is critical for performance.

[3]This is a problem also with the new generation enclave (*e.g.,* Intel TDX [12]).

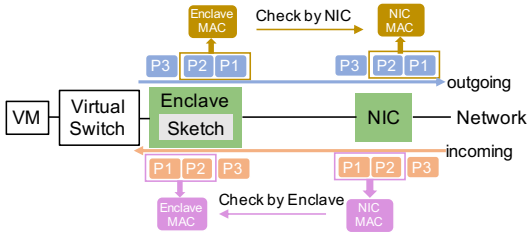


Fig. 6: *Strawman 2: calculate separate MACs for in and out direction based on predefined number of packets (2 in this figure) .*

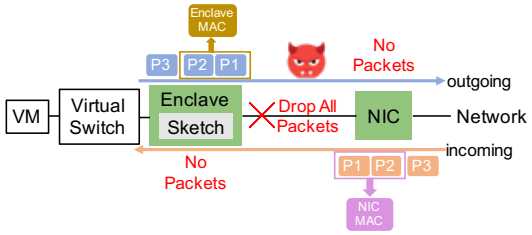


Fig. 7: *Strawman 2 fails: the attacker can silently drop all packets, while evading detection.*

A. Synchronizing MAC exchanges

As mentioned in §IV-C, the checkers periodically calculate a MAC on the packets they forward and compare their results. To be effective, the comparison must be *(i) precise* as false positives / negatives would degrade the performance or security of the system ; *(ii) timely* such that the detection delay is bounded; *(iii) lightweight* to incur minimum computation, memory, and network overhead.

Strawman solutions To see why this is challenging, we consider two strawman solutions and explain why they do not meet the requirements. First, one could calculate one MAC when T incoming and T outgoing packets (T is a predefined number) have been forwarded. This strawman would only work if the two directions are symmetric in rate. If the incoming rate is different from the outgoing rate, then either some packets will be left invalidated or the checkers will evaluate the MAC on different sets of packets.

From the previous strawman, we can conclude that one needs to maintain two summaries (one for each direction). Thus, for the second strawman we consider that each checker calculates one incoming MAC when they forward T incoming packets, and one outgoing MAC when they forward T outgoing packets. Then, they could exchange the incoming or outgoing MAC separately. Fig. 6 shows an example with $T = 2$ packets. However, an attacker could drop all packets between the enclave and the NIC, as shown in Fig. 7. This attack would not be detected because the NIC (enclave) would not receive any outgoing (incoming) MAC packet and thus no MAC mismatch would be detected.

Our approach TRUSTSKETCH builds on top of the second strawman solution but refrains from using a packet-number-based epoch. Instead, TRUSTSKETCH uses time-based epoch and in-band synchronization. Concretely, TRUSTSKETCH runs two timers, one at the enclave and one at the NIC. The timer would periodically fire out to signal the end of one epoch. For the outgoing packets, the *Enclave-Checker* would compute a MAC using all packets in that epoch (*e.g.,* there are 2 packets in outgoing direction in Fig. 8) and send the MAC to the *NIC-Checker* right after those packets.

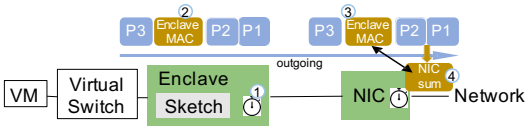


Fig. 8: *MAC exchange. For outgoing packets, 1. the enclave timer fires out; 2. the* Enclave-Checker *calculates a MAC; 3. the MAC arrives at the NIC as an in-band notification; 4. the* NIC-Checker *calculates a MAC and compares the two MACs. The incoming packets follow a symmetric procedure (not shown).*

Observing that even slight time drift between the two timers would cause the checkers to calculate the MAC on different sequences of packets, we need the two timers perfectly synchronized, which is infeasible. Even assuming perfectly synchronized timers, the set of packets inputted to the checkers might be different, as some packets are in transit between the enclave and the NIC. Our solution uses the MAC packet as an in-band synchronization notification, which tells the *NIC-Checker* when one epoch ends. Concretely, the timer at the enclave signals the end of an epoch for the outgoing direction (and triggers the enclave to send a MAC) and the timer at the NIC for the incoming direction (and triggers the NIC to send a MAC). When the *NIC-Checker* receives the MAC, it computes a new MAC and compares the two MACs. In this way, we provide synchronization between two checkers without the need for perfectly synchronized timers. The procedure for incoming packets is symmetric: *NIC-Checker* would compute a MAC and send it to the *Enclave-Checker*.

If any checker finds a MAC mismatch, it raises an alert, *i.e.,* notifies the controller. We discuss reactions for MAC mismatch in §VII. If any checker does not receive the MAC over a predefined time period, it also raises an alert. To ensure that an attacker cannot counterfeit or replay the MAC packet, we add a nonce-based cryptographic hash to the MAC packet.

B. Handling multithreading

Problem Modern NICs are built with multi-threads (*e.g.,* the Netronome NIC we use in our evaluation has 54 cores in total and 4 threads on each core) to process packets in parallel to maximize throughput. This would lead to two problems. First, there will be a mismatch between the packets on which the checkers will calculate the MAC, thus a false alert. Indeed, if we simply run one checker on each NIC thread, the NIC would generate N MACs at the end of each epoch per direction (assuming that the NIC has N threads in total). In contrast, the enclave checker processes the entire packet stream and would generate one MAC per direction, assuming that the enclave runs in one thread[4]. Then, there would be a MAC mismatch (1 vs. N) between the enclave and the NIC.

Second, there will be a mismatch between the order in which the checkers will calculate the MAC, thus a false alert. Indeed, since packets have different lengths and NIC threads share resources (*e.g.,* CPU, memory), packets might be processed out-of-order by different threads. Note that the NIC has a GRO (global reorder module) to ensure that the packets leave the NIC in the same order as they enter, but the NIC does not provide a guarantee about the packet processing order. This would also result in a MAC mismatch (packet order in MAC input), if the *NIC-Checker* processed packets in a different order than the enclave checker. Also, since we use the MAC packet as an in-band synchronization to signal the end of an epoch, if the *NIC-Checker* processes the MAC packet early or late, there would be a MAC mismatch due to the incorrect MAC input length.

[4]The problem still exists even if the enclave is running with multi-threads.

Strawman solutions Before we describe our solution, we consider two strawman solutions and show why they would not work. First, we could use only one thread on the NIC to process all packets and calculate one MAC. This would address the problem but would greatly degrade performance. We show in §VIII that using one thread would only achieve 0.1 Gbps throughput. Second, we could divide the original packet stream into N packet substreams at the NIC and the enclave. We could map a packet to a substream in a round-robin way: the first packet goes to the first substream, the second packet goes to the second substream, and so on. Then, each NIC thread would process one packet substream, resulting in N MACs in the NIC. Since packets mapped to the same thread would be processed in order, we would have solved the out-of-order problem. The *Enclave-Checker* would also calculate one MAC per packet substream. Both the NIC and the enclave would calculate N MACs each epoch. Unfortunately, this is not always a viable solution. Our Netronome NIC, for instance, hides the NIC-specific packet steering logic from the programmer, so we cannot control which packet is processed by which NIC thread.

Our approach We build on the second strawman solution by explicitly appending a tag to the packets and using the reorder buffer to reconstruct consistent packet substreams between the enclave and the NIC. Both the *Enclave-Checker* and the *NIC-Checker* maintain a global packet sequence counter and $2M$ buffers (they maintain M substreams for each direction where M is a configurable parameter, independent of the total number of NIC threads N). The NIC counter and buffers are protected by locks to prevent race conditions between threads. We optimize the buffer size by using streaming MAC (§VII-A) to achieve a small memory footprint.

When an outgoing packet arrives at the enclave, the *Enclave-Checker* updates the counter, uses the packet sequence number to decide which substream the packet belongs to, copies the packet header to that buffer (as mentioned in §III-A, TRUSTSKETCH only needs to protect the packet header for *sketch-input-integrity*), tags the packet with the sequence number, and forwards the packet to the *NIC-Checker*. Upon arrival of the packet to the *NIC-Checker*, the latter uses the number in the packet tag as the packet's sequence number to decide which substream the packet belongs to, copies the packet header to that buffer, removes the tag, and sends the packet out (to the virtual switch or the network). The procedure for incoming packets is symmetrical: *NIC-Checker* would update the counter in the NIC, and use it as the packet sequence number.

Our solution also solves the out-of-order processing problem. Fig. 9 shows an example: incoming packets P_1, P_2 are processed out of order. P_2 is processed first and gets a tag 1, and P_1 gets a tag 2. When the *Enclave-Checker* receives P_1, P_2, it uses the tag to reconstruct the same substreams as the substreams in the NIC. Note that due to out-of-order processing, the packets belong to different substreams in two directions (h_1, h_2 are copied to different buffer positions). In case the MAC packet is processed out-of-order, we include in the MAC packet the total number of packets in that epoch, so

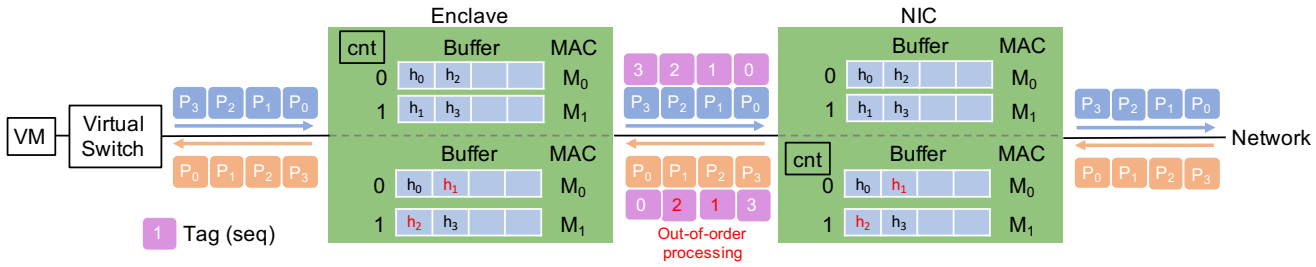


Fig. 9: TRUSTSKETCH *appends a tag (in pink) to the packets to explicitly declare the order at which they were processed (may be different from the order they arrived). These tags are used by reorder buffers to reconstruct consistent packet substreams between the enclave and the NIC. The incoming packets,* P_1, P_2*, are processed by the NIC threads out-of-order and have tags* $2, 1$*. The* Enclave-Checker *reconstructs the same packet substreams based on the tags.*

that the *NIC-Checker* could wait until it receives all packets in that epoch before it computes the MAC.

VI. SECURITY ANALYSIS

Using a case-based qualitative analysis and a model-driven analysis, we show that the design of TRUSTSKETCH achieves the correctness requirements mentioned in §III-A in contrast to other solutions, such as using an unmodified telemetry architecture or naively using enclaves (e.g., Safebricks [78]).

A. Qualitative Analysis

First, we qualitatively explain why TRUSTSKETCH protects against attack strategies in §II-C and attacks towards TRUSTSKETCH design (*e.g.,* DoS attack toward the enclave).

We observe that, by construction, TRUSTSKETCH prevents sketch-compute-attack and sketch-memory-attack by placing the entire sketch (code and data) in the enclave. This builds directly on the integrity properties that enclaves provide against software stack attacks [71]. Note that side-channel attacks on enclaves [35], [50], [63], [86] and SmartNIC [81], [92] are not relevant in our context, since our goal is only the integrity and not the confidentiality of the telemetry results. Second, TRUSTSKETCH can detect the sketch-input-attack thanks to input validation. If an attacker injects, drops, modifies, or reorders the packet sequence between the enclave and the NIC, the MACs calculated by the enclave and the NIC would show inconsistency.

A potential concern is that an attacker can attack the input validation mechanism itself by injecting, dropping, modifying, or replaying the MAC packet. Also, the attacker could reorder the MAC packet with datapath packets. However, injecting, modifying, or replaying a MAC packet would be detected because the MAC would show inconsistency. Dropping the MAC packet would cause the checker not to receive the MAC packet on time, resulting in an alert due to a timeout. Reordering the MAC packet with datapath packets would mess up synchronization and lead to an inconsistency between the MACs calculated by the enclave and the NIC.

Another potential concern is a DoS attack against the enclave. The attacker could send many packets from a VM to the virtual switch to overwhelm the enclave. This might lead to some packet drops at the virtual switch if the enclave processing rate is lower than the DoS packet rate. Even in this case, though, the telemetry result reported by the sketch would still be correct because the NIC would only send packets that have been processed (tagged) by the enclave.

B. Model Driven Analysis

We use Alloy [58], a declarative language used for system modeling to define system components and their expected behaviors for our model-driven analysis. Alloy implements an analyzer based on first-order logic that provides automatic correctness verification or provides counterexamples if the correctness is not satisfied. Next, we discuss how we model the system components in a cloud server, various designs from the design space, and our end-to-end design.

Model Description First, we model three basic elements of the servers in the cloud: code, memory, and packet. For example, in our model, all *Memory* objects have two attributes: *memoryPosition* and *security*, which indicates where the piece of memory is and whether it is compromised by the attacker. Then, we model three key entities in the telemetry system: a Sketch, a NIC, and a packet buffer between the Sketch and the NIC. *seq* is provided by Alloy to model a sequence of objects. We use *seq* to model the packet sequence. We also model the security constraints of each component. We only show two security constraints here for brevity: (1) if the attacker does not have root access, we assume that all programs and memory are secure. (2) if the packet buffer between the sketch and the NIC is secure, we assume that the sketch would get the same packet sequence as the NIC for both the incoming and outgoing directions. Since the Alloy model cannot model the time, for Input Validation, we model the packet sequence on a per-epoch basis. That is, if the *sketch-input-integrity* is correct for the packet stream in each epoch, we can inductively conclude that this property would hold for the entire packet stream.

Design Refinement We used our Alloy model as an integral part of our design workflow to identify potential blind spots in addition to validating that our design is robust against a broad spectrum of integrity attacks. For instance, an early version of our design used a packet-number-based epoch instead of the time-based epoch to synchronize the MAC exchange (§V-A). Using our Alloy model, we identified a subtle but important counterexample (as we illustrate in Fig. 7): the packet-number-based epoch would not detect input integrity violation if the attacker drops all packets between the enclave and the NIC.

Design Validation First, we verify that TRUSTSKETCH meets the three security requirements defined in §III-A. To that end, we set a check scope of 10. Therefore, the Alloy Analyzer will enumerate up to 10 instances of each *sig* to see if there is a counterexample that violates security requirements. Alloy does not find any counterexamples; thus TRUSTSKETCH meets our

Attack Type	Example Attack	Vanilla	Safebricks	TRUSTSKETCH
Compute	Modify runtime library	✓	×	×
Memory	Modify the counter	✓	×	×
	Modify the heap	✓	×	×
Input	Inject packets	✓	✓	×
	Drop packets	✓	✓	×
	Modify packet header	✓	✓	×

TABLE II: *Alloy model attack result: among the six attacks we model, some attacks are effective against Vanilla and Safebricks; none is effective against* TRUSTSKETCH.

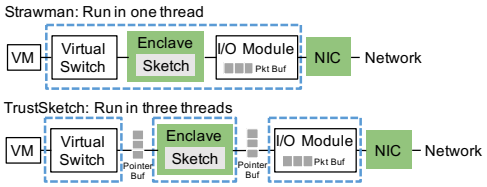


Fig. 10: TRUSTSKETCH *(lower) leverages that the three components can run in parallel and assigns them in different threads. This parallelism saves costly transitions between the enclave and host memory which would occur if only one thread is used (upper).*

security requirements.

Second, we model six attacks and check if those attacks could escape detection by three sketch systems[5]:*(i)* Vanilla: existing unsecure sketch system; *(ii)* Safebricks [78]: a previous work uses enclave to protect critical functions (in our case, the sketch); and *(iii)* TRUSTSKETCH. Table II shows the attack results: Vanilla and Safebricks are vulnerable to some attacks, while TRUSTSKETCH prevents those attacks.

VII. IMPLEMENTATION

In this section, we discuss two implementation optimizations and our end-to-end prototype.

A. Optimizations

Reducing enclave transition overhead. As shown in Fig. 10, a strawman solution is to run packet processing logic (virtual switch, Sketch and I/O Module) in one thread. However, this would involve transitions between host memory (virtual switch and I/O Module) and enclave memory (sketch), which would incur a high overhead due to the cost of saving and restoring the enclave state.

To reduce the overhead of the enclave transition, previous work [60], [78] runs the host and enclave logic in separate threads so that there will be no transitions between host memory and enclave memory. We take a similar approach: running virtual switch, Sketch, and I/O Module in three threads. Note that our solution uses two more cores than strawman solutions. However, this cost could be amortized by running multiple enclave (sketch) threads. All packet data is stored in a packet buffer managed by the I/O Module and threads pass packet pointers through pointer buffers to avoid packet copying.

Reducing memory usage. A naive way to implement MAC is to concatenate all packet headers in one epoch as input to generate the output $hash$. However, this requires all packet headers buffered in the checkers, wasting a large amount of memory resources. To reduce memory usage, we use a streaming MAC, which maintains an internal state S and keeps updating it $S' = F(X, S, k)$ when the input string X (*i.e.,* packet headers) arrives. At exchange time, the streaming MAC would generate a final hash based on the internal state $hash = G(S)$.

Note that this is not a conflict with the reorder buffer mechanism mentioned in §V-B. If using a naive MAC, the reorder buffer needs to be large enough to store all the packet headers used to compute the MAC. By using a streaming MAC, we can use a smaller buffer size and hence save memory usage.

[5]Since Alloy does not support modeling the alert mechanism of our system, we use the analyzer to check if there is an attack without triggering the alert.

B. Prototype

NIC. We use the Netronome Agilio® CX 1x40GbE SmartNIC [15] and implement the *NIC-Checker* in P4 [34] and Micro-C (a subset of C code supported by the Netronome SmartNIC). The SmartNIC has 54 cores in total and 8 threads on each core. We use all threads available on the NIC ($N = 54 \times 8$) to process packets and run *NIC-Checker*. The choice of M (the total number of packet substreams) is a trade-off between throughput and memory usage. A smaller M would consume fewer memory resources in the enclave and the NIC, but each NIC thread will have to process more packets than it can handle, resulting in lower throughput. In our implementation, we use $M = 64$.

Sketch implementation in the software. We use Open vSwitch [77] to manage virtual network and use DPDK [41] as the fast I/O Module, which is a popular kernel bypassing packet processing library used by many prior works [66], [78]. To allow the sketch (in the enclave) to access the packets stored in host memory, we follow the paradigm of SEC-IDS [60] and patch some glue I/O code (*i.e.,* DPDK Mbuf module) in the enclave.

Zero copy. We achieve zero packet copy by passing pointers between modules. Take incoming packets as an example. When the NIC receives a packet, it writes the packet to a region of untrusted host memory managed by the I/O module. Pointers are passed to the checker, the sketch, and the virtual switch so that they can access the packets. Note that the monitored data (i.e., packet headers) is copied to the enclave before the hashing and sketching operations. Thus, the adversary cannot change the headers between the time the enclave computes the hash and when it computes the sketch.

MAC. We use SipHash as the stream-MAC and adopt an open source implementation of SipHash [18]. We compute the hash over 5 tuples (source IP address, destination IP address, source port, destination port, protocol). Thus, we guarantee the integrity of those five fields, but the sketch could use any subset of those for flow keys.

Tag. We put tag between the ethernet header and the IP header. We use a one bit in tag to distinguish the MAC packets from regular datapath packets so that the NIC can invoke different processing functions for them correspondingly. We do not explicitly protect the flag bit from an attacker as she has no incentive to change it. If an attacker sets this bit on a regular datapath packet, the checker would take it as a MAC packet, check the MAC and find a mismatch. If an attacker

unsets this bit for a MAC packet, then the MAC packet would be processed as a datapath packet by the checker. Since the original MAC packet would be missing afterward, the checker will eventually raise an alert.

Timer. TRUSTSKETCH runs two timers at the enclave and the NIC to signal the MAC exchanges for integrity checking. We require the timer to provide accurate timing differences, but do not need absolute timestamps or synchronization between the two timers. For the enclave part, we implement the counting thread from prior work [82] that provides a cycle count. The counting thread is running in the enclave so it is also secure.

For the NIC part, the SmartNIC we use does not provide a timing API [83]. So we simulate the timer based on packet number: the timer would fire out when the number of packets received by the NIC reaches a threshold. As we mentioned in §V-A, a packet number-based epoch could be attacked. Here, we only use it as a simulated timer to verify the epoch synchronization mechanism. For TRUSTSKETCH to be deployed in the real world, a real timer is needed. For example, there are other SmartNICs [7] that provide a timing API.

We set the epoch length to 1s, so there would be a MAC exchange every second. We have evaluated that performance is not sensitive to the epoch length (not shown). We use two counters (C_{pass} and C_{fail}) in the enclave and in the NIC to track how many MAC checks have passed and failed.

Reactions for MAC mismatch. Upon detection of a MAC mismatch indicating the presence of an attacker, TRUSTSKETCH must operate in a timely manner to limit the impact of the attack. Here, we consider two natural alternatives. First, we can use a secure channel between the NIC or the enclave to notify the cloud operator and defer policy actions to the operator. We assume that this channel is set up during bootstrap, which is typical in many modern software-defined data centers. In addition, we also provide hooks for a faster response in the data path, *i.e.,* in the enclave or NIC. For example, TRUSTSKETCH can react to MAC mismatch immediately by stopping further damage; *e.g.,*, to isolate a compromised server from inflicting further damage.

Key management. There is the risk of integrity violations if we reuse the same key for a long time. To avoid this, we envision periodically refreshing the key (say every 10 million packets) by using a stronger cryptographic key generation algorithm (*e.g.,* HOTP [87]). Note that this does not require additional synchronization between the enclave and the NIC.

TCB size. TRUSTSKETCH in the enclave part consists of 5K LoC. In comparison, OVS consists of ~364K LoC, and the DPDK library consists of ~131K LoC. Therefore, placing OVS and DPDK in host memory greatly reduces the TCB size by 99x.

VIII. EVALUATION

We perform a wide range of experiments to evaluate the effectiveness of TRUSTSKETCH and demonstrate that:

- **Security:** Today's systems are vulnerable to simple attacks. TRUSTSKETCH detects the attacks in less than one second.
- **Performance:** Compared to existing unsecure systems, TRUSTSKETCH incurs low overhead in terms of throughput (7%) and latency (6μs).

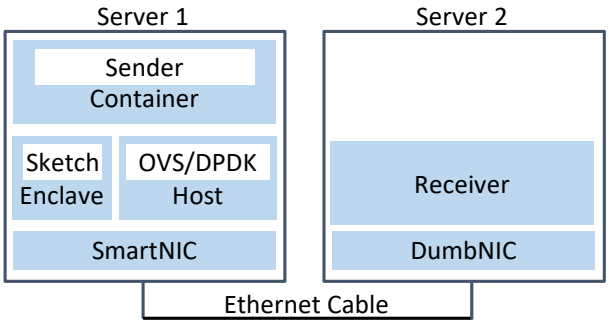


Fig. 11: *Evaluation Testbed*[6]

Task	Sketch	Memory (KB)
Heavy Hitters	Count-Min Sketch [40]	200
	Count Sketch [36]	200
	CocoSketch [91]	500
Cardinality	HyperLogLog [48]	5
Flow size distribution	MRAC [59]	250
Heavy change	FlowRadar [64]	600
General	UnivMon Sketch [68]	800
	Elastic Sketch [88]	750

TABLE III: *Sketch parameters for evaluation*

- **Generality:** We implement 8 state-of-the-art sketches and demonstrate that TRUSTSKETCH is applicable to different sketches without accuracy degradation.

A. Experimental Methodology

Testbed. As shown in Fig. 11, our testbed has two commodity servers: both have 24-core 3.00GHz Intel Xeon Gold 5317 CPU with 512 GB RAM and a Netronome SmartNIC. Server 1 enables Intel SGXv2 [43] and runs TRUSTSKETCH. We run Docker [73] on Server 1 to create containers. We run MoonGen [44] in a container to generate realistic packets packets *i.e.,* according to a packet trace. Server 2 runs a DPDK receiver and the SmartNIC on Server 2 is configured as a dumbNIC. The two servers are directly connected via a 40Gbps link. We enable jumbo frames in Open vSwitch to allow the tag to be added to large packets (making them larger than the default MTU of 1500 bytes).

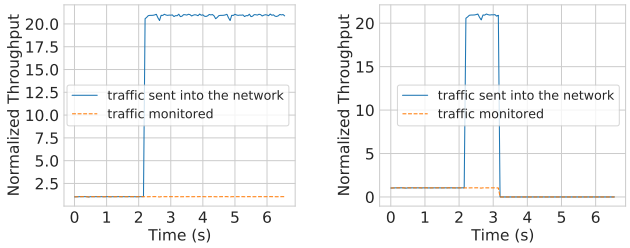
Workloads. We use four types of workload to represent various network conditions: (1) Data Center: data center packet traces UNI1 and UNI2 from [32]. (2) CAIDA: Anonymous Internet traces [6] collected from a commercial backbone link. (3) Min-sized: synthetic traffic with min-sized packets (64B). This workload represents high line-rate stress tests (4) Max-sized: synthetic traffic with max-sized packets (1500B).

Sketches and Parameters. We implement 8 state-of-the-art sketches for a variety of telemetry tasks as shown in Table III.

Metrics. We use 5-tuple as the flow key. For telemetry metrics, we report $relative\_error = \frac{|t-t_{real}|}{t_{real}}$, where t_{real} is the ground truth of a metric and t is the measured value. For throughput and latency, we report median and use the error bar to show a 95% confidence interval.

Systems in Comparison. We compare the following system designs: (1) **Vanilla**: Existing sketch-based implementation

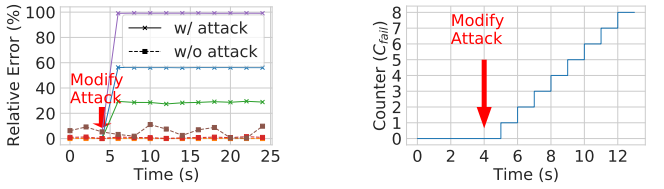
[6] TRUSTSKETCH monitors bidirectional traffic (incoming and outgoing of the server). We also switch the sender and receiver to test traffic in the other direction. For brevity, we only report the results of the outgoing traffic (the numbers of the incoming traffic are similar).



(a) Vanilla: fails to report the attack traffic volume

(b) TRUSTSKETCH: detects the attack in less than 1s and shuts off all traffic

Fig. 12: *Inject-Packet-Attack*



(a) Vanilla: the accuracy of telemetry metrics degrades after the attack is launched.

(b) TRUSTSKETCH: the increase of counter value shows that TRUSTSKETCH detects the attack in less than 1s.

Fig. 13: *Modify-Packet-Header-Attack*

without any security mechanisms; (2) **Safebricks**: we reimplement Safebricks [78] with running sketches in the enclave; and (3) **TRUSTSKETCH**: End-to-end TRUSTSKETCH design and implementation.

B. End-to-End Validation With Real Attacks

We examine the security of our design by testing our end-to-end prototype against real attacks. The compute and memory integrity is guaranteed by the enclave by construction. Thus, in the interest of brevity, we do not explicitly test against Sketch-Compute-Attack and Sketch-Memory-Attack. Instead, we focus on testing Sketch-Input-Attack which depends on our input validation. We implement two Sketch-Input-Attacks and check if TRUSTSKETCH can preserve Sketch-input-integrity under the corresponding attacks.

Inject-Packet-Attack. We run a separate MoonGen [44] *pktgen* application on *Server 1* to send malicious traffic. In this case, the pktgen bypasses the sketch in the enclave and creates unmonitored outgoing traffic. As shown in Fig. 12, the self-generated traffic is forwarded without being monitored by Vanilla. On the other hand, TRUSTSKETCH detects the unmonitored traffic and raises an alert in one second. The operator can make control decisions on how to handle this traffic, *e.g.,* dropping all packets to minimize the risk of attack.

Modify-Packet-Header-Attack. We launch a modify-packet-header attack by modifying the outgoing packet headers when the packets are buffered in the virtual switch buffer (those packets have already been processed by the sketch and waited to be sent out by the NIC). By modifying the packet headers, the sketch would monitor incorrect packet stream and hence generate incorrect telemetry results. We calculate the accuracy of three telemetry metrics (L2 Norm, Entropy, Cardinality) every five seconds. As shown in Fig. 13(a), the telemetry accuracy of Vanilla degrades greatly after the attack is launched. On the other hand, TRUSTSKETCH could immediately detect the attack (Fig. 13(b)): after only one second, C_{fail} (the counter tracking how many MAC checks have failed) on the NIC starts to increase[7], indicating that the NIC correctly detects this attack.

C. Performance

Next, we show the overall performance of TRUSTSKETCH.

Throughput. To measure the throughput of TRUSTSKETCH, we use MoonGen [44] to generate high-speed packet streams based on traces via a VM in Server 1 and send packets to a DPDK receiver in Server 2. Fig. 14 shows that compared to Vanilla, TRUSTSKETCH incurs 7% overhead.

Latency. We use *chrony* [38] to synchronize the system clocks between two servers and measure the one-way delay. When MoonGen packet generator sends packets from a VM in Server 1, the sending rate is limited to 80% of the maximum system throughput (as shown in Fig. 14). For each packet, a timestamp is added to the packet header before it leaves the VM. When the DPDK receiver on Server 2 receives the packet, it extracts the sending timestamp from the packet and calculates the elapsed time. As shown in Fig. 15, TRUSTSKETCH incurs 6μs latency compared to Vanilla. Compared to Vanilla, even Safebricks adds about 4μs to the overall latency due to the cost of context switches between the host memory and the enclave memory.

Accuracy. We calculate the accuracy of the telemetry metrics and demonstrate that TRUSTSKETCH could achieve the same accuracy as Vanilla as shown in Fig. 16 (we only show the accuracy result for UnivMon Sketch for brevity).

Optimization. When implement TRUSTSKETCH, we use two optimization techniques. Fig. 17 shows the throughput gain of using multiple NIC threads(§V-B), and removing enclave transition(§VII-A).

NIC Resource Usage. We calculate the NIC CPU usage based on how many cores it takes to achieve $40Gbps$. It takes 9 cores without running input validating and 10 cores with running input validating. Given that the NIC has 54 cores in total, the CPU utilization overhead is around 1.9%. We compute the NIC memory usage based on the size of all variables we use for the input validation mechanism. We use 104KB, only 0.53% of the total NIC on-chip memory (19.2MB).

IX. RELATED WORK

Sketch-based telemetry. A number of techniques have been proposed to improve the accuracy, generality and performance of sketch-based telemetry [54], [67], [68], [88], [91]. For example, UnivMon adopts universal sketching theory to build a general sketch to estimate a wide range of traffic statistics. CocoSketch extends the canonical SpaceSaving algorithm [74] to support unbiased estimation over all flows. Our telemetry framework with CPU and SmartNIC can adapt to different kinds of sketches for different telemetry tasks.

Other aspects of secure telemetry. OblivSketch [61] uses oblivious data structures and algorithms in the control plane to ensure the privacy of the telemetry statistics during query time. However, it only improves the security of the telemetry tool in the control plane and assumes that the data plane is trustworthy. Our work focuses on the data plane and the integrity of the

[7]Netronome provides a tool to read NIC variable values from command line.

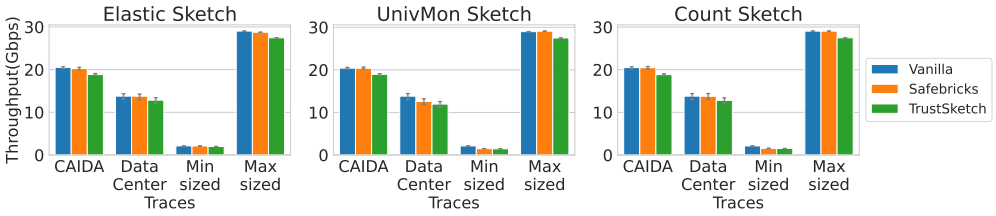


Fig. 14: *Throughput comparison: compared to Vanilla,* TRUSTSKETCH *incurs 7% overhead.*

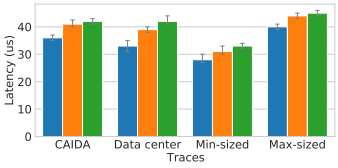


Fig. 15: *Latency comparison: compared to Vanilla,* TRUSTSKETCH *incurs 6μs latency.*

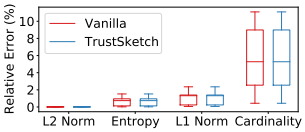


Fig. 16: TRUSTSKETCH *achieves the same accuracy as Vanilla.*

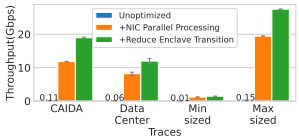


Fig. 17: *Throughput with different* TRUSTSKETCH *optimizations applied.*

measurements (i.e., not confidentiality) and is complementary to their work.

Secure Enclaves for Network Functions. LightBox [42], Safebricks [78], S-NFV [84], SGX-Box [51] are frameworks that improve security for general virtualized network functions. However, they could not guarantee Sketch-input-integrity. An attacker could insert, drop, or modify packets between the telemetry tool and the NIC and remain undetected.

Verifiable fault localization and measurements. DynaFL [90] could identify compromised and misconfigured routers in the data plane. Other work on trustworthy measurements [26] focuses on verifying the performance measurements reported by each network domain. However, these works focus on wide-area setting with router-specific constraints. In contrast, our work focuses on providing a trustworthy software telemetry tool in the cloud.

X. DISCUSSION

Before we conclude, we discuss limitations and alternative usage model for TRUSTSKETCH.

Limitations of TRUSTSKETCH First, we note that for *sketch-input-integrity*, we only protect the path between the enclave and the SmartNIC, so we only protect the packets that access the network. An attacker could still corrupt the traffic between the sketch and a VM. For instance, the attacker could send packets from one VM and spoof the source IP address to pretend that the packets are sent by another VM. Second, an attacker could also corrupt the traffic between VMs (*e.g.,* using one VM to send traffic to another VM on the same physical machine to launch a DoS attack). We cannot guarantee the correctness of the telemetry result for traffic between VMs on the same physical machine because those packets do not go through the NIC.

Alternative cloud usage model Some cloud customers rent clusters of bare metal servers [1] from cloud providers and build their own virtual private cloud. In this scenario, the cloud provider still manages the datacenter network between the servers, but cloud customers will manage the whole software stack on the bare metal, including VM, telemetry tool, virtual switch, and hypervisor. In this case, cloud customers can use TRUSTSKETCH to run trustworthy telemetry to better manage their virtual private clouds, but the scope of threats of interest in these deployments may be different (e.g., to detect compromised components rather than to account per se).

Deployability concerns with enclaves. There have been concerns about the long-term viability of SGX. For example, SGX is deprecated on "desktop" or "client" machines in the 11th and 12th generations of Intel Core processors [10]. However, on server platforms, SGX is currently supported and is expected to have ongoing support in the foreseeable future [21]. More generally, the enclave-specific requirements of TRUSTSKETCH are minimal and can potentially be supported in other enclave technology as well; e.g., Intel TDX [12]). As such, we believe that our architecture and design which combine an enclave and a SmartNIC to ensure trustworthy telemetry will also apply to future server technologies.

XI. CONCLUSIONS

Our work is motivated by the observation that existing software sketch telemetry frameworks can be circumvented by cloud attackers, effectively deceiving downstream management tasks. To address this, we take a first-principles approach and formally define correctness properties for trustworthy sketch-based telemetry. We present a practical design called TRUSTSKETCH, which achieves correctness, high performance and is also general *i.e.,* supports many classes of sketches. Therefore, TRUSTSKETCH can serve as a basis for future trustworthy software sketch telemetry deployment.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their valuable feedback. This work was supported in part by NSF grants CNS-2107086, CNS-2106214, and CPS-2111751.

REFERENCES

[1] Amazon ec2 bare metal. https://aws.amazon.com/about-aws/whats-new/2019/02/introducing-five-new-amazon-ec2-bare-metal-instances/.

[2] Amazon Web Services (AWS). https://aws.amazon.com/.

[3] AMD Secure Memory Encryption (SME). https://developer.amd.com/sev/.

[4] AWS Nitro Enclaves. https://aws.amazon.com/cn/ec2/nitro/nitro-enclaves/.

[5] Azure announces next generation Intel SGX confidential computing VMs. https://techcommunity.microsoft.com/t5/azure-confidential-computing/azure-announces-next-generation-intel-sgx-confidential-computing/ba-p/2839934.

[6] CAIDA Trace. https://www.caida.org/catalog/datasets/monitors/passive-equinix-nyc/.

[7] Cisco Nexus SmartNIC. https://www.cisco.com/c/en/us/products/interfaces-modules/nexus-smartnic/index.html.

[8] Google Cloud Platform. https://cloud.google.com/.

[9] Hackers exploiting saltstack vulnerability hit data centers. https://www.datacenterknowledge.com/security/hackers-exploiting-saltstack-vulnerability-hit-data-centers.

[10] Intel Core Processors 12th Generation. https://cdrdv2.intel.com/v1/dl/getContent/655258.

[11] Intel is halting development of the networking chip it got from Barefoot Networks. https://www.bizjournals.com/sanjose/news/2023/01/26/intel-halts-development-of-tofino-switch-chips.html.

[12] Intel Trust Domain Extensions. https://www.intel.com/content/www/us/en/developer/articles/technical/intel-trust-domain-extensions.html.

[13] Kubernetes. https://kubernetes.io/.

[14] Microsoft Azure. https://azure.microsoft.com/en-us/.

[15] Netronome smartnic. https://www.netronome.com/products/smartnic/overview/.

[16] openstack. https://www.openstack.org/.

[17] Overview of Data Transfer Costs for Common Architectures. https://aws.amazon.com/blogs/architecture/overview-of-data-transfer-costs-for-common-architectures/.

[18] SipHash. https://github.com/veorq/SipHash.

[19] [v5,1/2] member: implement nitrosketch mode - patchwork. https://patches.dpdk.org/project/dpdk/patch/20220916030317.3111820-2-leyi.rong@intel.com/.

[20] Vmware esxi successful vm escape at geekpwn2018 security patch. https://www.virtualizationhowto.com/2018/11/vmware-esxi-successful-vm-escape-at-geekpwn2018-security-patch/.

[21] Will SGX be deprecated? https://github.com/intel/linux-sgx/issues/760.

[22] Xen hypervisor patched for privilege escalation and information leak flaws - the new stack. https://thenewstack.io/privilege-escalation-information-leak-flaws-patched-xen-hypervisor/.

[23] A. Agache, M. Brooker, A. Iordache, A. Liguori, R. Neugebauer, P. Piwonka, and D.-M. Popa. Firecracker: Lightweight virtualization for serverless applications. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 419–434, Santa Clara, CA, Feb. 2020. USENIX Association.

[24] A. Agarwal, Z. Liu, and S. Seshan. Heterosketch: Coordinating network-wide monitoring in hetero-geneous and dynamic networks. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, Renton, WA, Apr. 2022. USENIX Association.

[25] M. Apostolaki, A. Singla, and L. Vanbever. *Performance-Driven Internet Path Selection*, page 41–53. Association for Computing Machinery, New York, NY, USA, 2021.

[26] K. Argyraki, P. Maniatis, and A. Singla. Verifiable network-performance measurements. In *Proceedings of the 6th International COnference*, Co-NEXT '10, New York, NY, USA, 2010. Association for Computing Machinery.

[27] S. Arnautov, B. Trach, F. Gregor, T. Knauth, A. Martin, C. Priebe, J. Lind, D. Muthukumaran, D. O'Keeffe, M. L. Stillwell, D. Goltzsche, D. Eyers, R. Kapitza, P. Pietzuch, and C. Fetzer. SCONE: Secure linux containers with intel SGX. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 689–703, Savannah, GA, Nov. 2016. USENIX Association.

[28] E. Assaf, R. Ben-Basat, G. Einziger, and R. Friedman. Pay for a sliding bloom filter and get counting, distinct elements, and entropy for free. In *Proc. of IEEE INFOCOM*, 2018.

[29] W. Bai, S. S. Abdeen, A. Agrawal, K. K. Attre, P. Bahl, A. Bhagat, G. Bhaskara, T. Brokhman, L. Cao, A. Cheema, R. Chow, J. Cohen, M. Elhaddad, V. Ette, I. Figlin, D. Firestone, M. George, I. German, L. Ghai, E. Green, A. Greenberg, M. Gupta, R. Haagens, M. Hendel, R. Howlader, N. John, J. Johnstone, T. Jolly, G. Kramer, D. Kruse, A. Kumar, E. Lan, I. Lee, A. Levy, M. Lipshteyn, X. Liu, C. Liu, G. Lu, Y. Lu, X. Lu, V. Makhervaks, U. Malashanka, D. A. Maltz, I. Marinos, R. Mehta, S. Murthi, A. Namdhari, A. Ogus, J. Padhye, M. Pandya, D. Phillips, A. Power, S. Puri, S. Raindel, J. Rhee, A. Russo, M. Sah, A. Sheriff, C. Sparacino, A. Srivastava, W. Sun, N. Swanson, F. Tian, L. Tomczyk, V. Vadlamuri, A. Wolman, Y. Xie, J. Yom, L. Yuan, Y. Zhang, and B. Zill. Empowering azure storage with RDMA. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 49–67, Boston, MA, Apr. 2023. USENIX Association.

[30] Z. Bar-Yossef, T. S. Jayram, R. Kumar, D. Sivakumar, and L. Trevisan. Counting distinct elements in a data stream. In *Proc. of RANDOM*, 2002.

[31] R. B. Basat, G. Einziger, R. Friedman, M. C. Luizelli, and E. Waisbard. Constant time updates in hierarchical heavy hitters. In *Proc. of ACM SIGCOMM and CoRR/1707.06778*, 2017.

[32] T. Benson, A. Akella, and D. A. Maltz. Network traffic characteristics of data centers in the wild. In *Proceedings of the 10th ACM SIGCOMM Conference on Internet Measurement*, IMC '10, pages 267–280, New York, NY, USA, 2010. ACM.

[33] T. Benson, A. Anand, A. Akella, and M. Zhang. MicroTE: Fine Grained Traffic Engineering for Data Centers. In *Proc. of ACM CoNEXT*, 2011.

[34] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker. P4: Programming protocol-independent packet processors. *SIGCOMM Comput. Commun. Rev.*, 44(3):87–95, jul 2014.

[35] F. Brasser, U. Müller, A. Dmitrienko, K. Kostiainen, S. Capkun, and A.-R. Sadeghi. Software grand exposure: Sgx cache attacks are practical. In *Proceedings of the 11th USENIX Conference on Offensive Technologies*, WOOT'17, page 11, USA, 2017. USENIX Association.

[36] M. Charikar, K. Chen, and M. Farach-Colton. Finding frequent items in data streams. In *Proc. of ICALP*, 2002.

[37] C. che Tsai, D. E. Porter, and M. Vij. Graphene-SGX: A practical library OS for unmodified applications on SGX. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 645–658, Santa Clara, CA, July 2017. USENIX Association.

[38] Chrony. https://chrony.tuxfamily.org/.

[39] G. Cormode and S. Muthukrishnan. An Improved Data Stream Summary: The Count-Min Sketch and Its Applications. *J. Algorithms*, 2005.

[40] G. Cormode and S. Muthukrishnan. An improved data stream summary: the count-min sketch and its applications. *Journal of Algorithms*, 55(1):58–75, 2005.

[41] Data Plane Develpment Kit. https://www.dpdk.org/.

[42] H. Duan, C. Wang, X. Yuan, Y. Zhou, Q. Wang, and K. Ren. Lightbox: Full-stack protected stateful middlebox at lightning speed. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS '19, page 2351–2367, New York, NY, USA, 2019. Association for Computing Machinery.

[43] M. El-Hindi, T. Ziegler, M. Heinrich, A. Lutsch, Z. Zhao, and C. Binnig. Benchmarking the second generation of intel sgx hardware. In *Proceedings of the 18th International Workshop on Data Management on New Hardware*, DaMoN '22, New York, NY, USA, 2022. Association for Computing Machinery.

[44] P. Emmerich, S. Gallenmüller, D. Raumer, F. Wohlfart, and G. Carle. Moongen: A scriptable high-speed packet generator. In *Proc. of ACM IMC*, 2015.

[45] FD.io. Vector packet processing, 2018.

[46] A. Feldmann, A. Greenberg, C. Lund, N. Reingold, J. Rexford, and F. True. Deriving traffic demands for operational ip networks: Methodology and experience. *IEEE/ACM Transactions On Networking*, 9(3):265–279, 2001.

[47] D. Firestone, A. Putnam, S. Mundkur, D. Chiou, A. Dabagh, M. Andrewartha, H. Angepat, V. Bhanu, A. Caulfield, E. Chung, H. K. Chandrappa, S. Chaturmohta, M. Humphrey, J. Lavier, N. Lam, F. Liu, K. Ovtcharov, J. Padhye, G. Popuri, S. Raindel, T. Sapre, M. Shaw, G. Silva, M. Sivakumar, N. Srivastava, A. Verma, Q. Zuhair, D. Bansal, D. Burger, K. Vaid, D. A. Maltz, and A. Greenberg. Azure accelerated networking: Smartnics in the public cloud. In *Proceedings of the 15th USENIX Conference on Networked Systems Design and Implementation*, NSDI'18, page 51–64, USA, 2018. USENIX Association.

[48] P. Flajolet, É. Fusy, O. Gandouet, and F. Meunier. Hyperloglog: the analysis of a near-optimal cardinality estimation algorithm. 2007.

[49] T. Garfinkel, M. Rosenblum, and D. Boneh. Flexible OS support and applications for trusted computing. In *9th Workshop on Hot Topics in Operating Systems (HotOS IX)*, Lihue, HI, May 2003. USENIX Association.

[50] J. Götzfried, M. Eckert, S. Schinzel, and T. Müller. Cache attacks on intel sgx. In *Proceedings of the 10th European Workshop on Systems Security*, EuroSec'17, New York, NY, USA, 2017. Association for Computing Machinery.

[51] J. Han, S. Kim, J. Ha, and D. Han. Sgx-box: Enabling visibility on encrypted traffic using a secure middlebox module. In *Proceedings of the First Asia-Pacific Workshop on Networking*, APNet'17, page 99–105, New York, NY, USA, 2017. Association for Computing Machinery.

[52] T. Holterbach, E. C. Molero, M. Apostolaki, A. Dainotti, S. Vissicchio, and L. Vanbever. Blink: Fast connectivity recovery entirely in the data plane. In *Proc. of USENIX NSDI*, 2019.

[53] Q. Huang, X. Jin, P. P. C. Lee, R. Li, L. Tang, Y.-C. Chen, and G. Zhang. Sketchvisor: Robust network measurement for software packet processing. In *Proc. of ACM SIGCOMM*, 2017.

[54] Q. Huang, P. P. Lee, and Y. Bao. Sketchlearn: Relieving user burdens in approximatemeasurement with automated statistical inference. In *Proc. of ACM SIGCOMM*, 2018.

[55] Q. Huang, H. Sun, P. P. C. Lee, W. Bai, F. Zhu, and Y. Bao. Omnimon: Re-architecting network telemetry with resource efficiency and full accuracy. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '20, page 404–421, New York, NY, USA, 2020. Association for Computing Machinery.

[56] IP Security (IPsec) and Internet Key Exchange (IKE) Document Roadmap. https://datatracker.ietf.org/doc/html/rfc6071.

[57] N. Ivkin, R. B. Basat, Z. Liu, G. Einziger, R. Friedman, and V. Braverman. I know what you did last summer: Network monitoring using interval queries. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 3(3):1–28, 2019.

[58] D. Jackson. Alloy: A new technology for software modelling. In J.-P. Katoen and P. Stevens, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 20–20, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg.

[59] A. Kumar, M. Sung, J. J. Xu, and J. Wang. Data streaming algorithms for efficient and accurate estimation of flow size distribution. In *Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '04/Performance '04, page 177–188, New York, NY, USA, 2004. Association for Computing Machinery.

[60] D. Kuvaiskii, S. Chakrabarti, and M. Vij. Snort intrusion detection system with intel software guard extension (intel sgx), 2018.

[61] S. Lai, X. Yuan, J. K. Liu, X. Yi, Q. Li, D. Liu, and S. Nepal. Oblivsketch: Oblivious network measurement as a cloud service. In *NDSS*, 2021.

[62] A. Lall, V. Sekar, M. Ogihara, J. Xu, and H. Zhang. Data streaming algorithms for estimating entropy of network traffic. In *Proc. of ACM SIGMETRICS/Performance*, 2006.

[63] S. Lee, M.-W. Shih, P. Gera, T. Kim, H. Kim, and M. Peinado. Inferring fine-grained control flow inside sgx enclaves with branch shadowing. In *Proceedings of the 26th USENIX Conference on Security Symposium*, SEC'17, page 557–574, USA, 2017. USENIX Association.

[64] Y. Li, R. Miao, C. Kim, and M. Yu. Flowradar: A better netflow for data centers. In *Proceedings of the 13th Usenix Conference on Networked Systems Design and Implementation*, NSDI, pages 311–324, Berkeley, CA, USA, 2016. USENIX Association.

[65] Y. Li, R. Miao, C. Kim, and M. Yu. Lossradar: Fast detection of lost packets in data center networks. In *Proceedings of the 12th International on Conference on Emerging Networking EXperiments and Technologies*, CoNEXT '16, page 481–495, New York, NY, USA, 2016. Association for Computing Machinery.

[66] G. Liu, H. Sadok, A. Kohlbrenner, B. Parno, V. Sekar, and J. Sherry. Don't yank my chain: Auditable NF service chaining. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 155–173. USENIX Association, Apr. 2021.

[67] Z. Liu, R. Ben-Basat, G. Einziger, Y. Kassner, V. Braverman, R. Friedman, and V. Sekar. Nitrosketch: Robust and general sketch-based monitoring in software switches. In *Proceedings of the ACM Special Interest Group on Data Communication*, pages 334–350. ACM, 2019.

[68] Z. Liu, A. Manousis, G. Vorsanger, V. Sekar, and V. Braverman. One sketch to rule them all: Rethinking network flow monitoring with univmon. In *Proc. of ACM SIGCOMM*, 2016.

[69] Z. Liu, S. Zhou, O. Rottenstreich, V. Braverman, and J. Rexford. Memory-efficient performance monitoring on programmable switches with lean algorithms. *CoRR*, abs/1911.06951, 2019.

[70] A. Manousis, Z. Cheng, R. B. Basat, Z. Liu, and V. Sekar. Enabling efficient and general subpopulation analytics in multidimensional data streams, 2022.

[71] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagaonkar. Innovative instructions and software model for isolated execution. In *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy*, HASP '13, New York, NY, USA, 2013. Association for Computing Machinery.

[72] Mellanox BlueField-2 SmartNIC. https://diaway.com/files/PB_BlueField-2_SmartNIC_ETH.pdf.

[73] D. Merkel. Docker: Lightweight linux containers for consistent development and deployment. *Linux J.*, 2014(239), Mar. 2014.

[74] A. Metwally, D. Agrawal, and A. E. Abbadi. Efficient computation of frequent and top-k elements in data streams. In *Proc. of ICDT*, 2005.

[75] G. Nychis, V. Sekar, D. G. Andersen, H. Kim, and H. Zhang. An empirical evaluation of entropy-based traffic anomaly detection. In *ACM IMC, 2008*.

[76] H. Oh, A. Ahmad, S. Park, B. Lee, and Y. Paek. Trustore: Side-channel resistant storage for sgx using intel hybrid cpu-fpga. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, CCS '20, page 1903–1918, New York, NY, USA, 2020. Association for Computing Machinery.

[77] B. Pfaff, J. Pettit, T. Koponen, E. Jackson, A. Zhou, J. Rajahalme, J. Gross, A. Wang, J. Stringer, P. Shelar, K. Amidon, and M. Casado. The Design and Implementation of Open vSwitch. In *Proc. of USENIX NSDI*, 2015.

[78] R. Poddar, C. Lan, R. A. Popa, and S. Ratnasamy. Safebricks: Shielding network functions in the cloud. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 201–216, Renton, WA, Apr. 2018. USENIX Association.

[79] A. Ramachandran, S. Seetharaman, N. Feamster, and V. Vazirani. Fast monitoring of traffic subpopulations. In *Proceedings of the 8th ACM SIGCOMM conference on Internet measurement*, pages 257–270, 2008.

[80] J. Rasley, B. Stephens, C. Dixon, E. Rozner, W. Felter, K. Agarwal, J. Carter, and R. Fonseca. Planck: Millisecond-scale monitoring and control for commodity networks. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, SIGCOMM '14, pages 407–418, New York, NY, USA, 2014. ACM.

[81] F. Schellenberg, D. R. Gnad, A. Moradi, and M. B. Tahoori. An inside job: Remote power analysis attacks on fpgas. In *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1111–1116, 2018.

[82] M. Schwarz, S. Weiser, D. Gruss, C. Maurice, and S. Mangard. Malware guard extension: Using sgx to conceal cache attacks. In M. Polychronakis and M. Meier, editors, *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 3–24, Cham, 2017. Springer International Publishing.

[83] R. Shashidhara, T. Stamler, A. Kaufmann, and S. Peter. FlexTOE: Flexible TCP offload with Fine-Grained parallelism. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 87–102, Renton, WA, Apr. 2022. USENIX Association.

[84] M.-W. Shih, M. Kumar, T. Kim, and A. Gavrilovska. S-nfv: Securing nfv states by using sgx. In *Proceedings of the 2016 ACM International Workshop on Security in Software Defined Networks & Network Function Virtualization*, SDN-NFV Security '16, page 45–48, New York, NY, USA, 2016. Association for Computing Machinery.

[85] V. Sivaraman, S. Narayana, O. Rottenstreich, S. Muthukrishnan, and J. Rexford. Heavy-hitter detection entirely in the data plane. In *Proc. of ACM SOSR*, 2017.

[86] J. Van Bulck, N. Weichbrodt, R. Kapitza, F. Piessens, and R. Strackx. Telling your secrets without page faults: Stealthy page table-based attacks on enclaved execution. In *Proceedings of the 26th USENIX Conference on Security Symposium*, SEC'17, page 1041–1056, USA, 2017. USENIX Association.

[87] M. View, D. M'Raihi, F. Hoornaert, D. Naccache, M. Bellare, and O. Ranen. HOTP: An HMAC-Based One-Time Password Algorithm. RFC 4226, Dec. 2005.

[88] T. Yang, J. Jiang, P. Liu, Q. Huang, J. Gong, Y. Zhou, R. Miao, X. Li, and S. Uhlig. Elastic sketch: Adaptive and fast network-wide measurements. In *Proc. of ACM SIGCOMM*, 2018.

[89] M. Yu, L. Jose, and R. Miao. Software defined traffic measurement with opensketch. In *Proc. of USENIX NSDI*, 2013.

[90] X. Zhang, C. Lan, and A. Perrig. Secure and scalable fault localization under dynamic traffic patterns. In *2012 IEEE Symposium on Security and Privacy*, pages 317–331, 2012.

[91] Y. Zhang, Z. Liu, R. Wang, T. Yang, J. Li, R. Miao, P. Liu, R. Zhang, and J. Jiang. Cocosketch: high-performance sketch-based measurement over arbitrary partial key query. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, pages 207–222, 2021.

[92] M. Zhao and G. E. Suh. Fpga-based remote power side-channel attacks. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 229–244, 2018.