OblivSketch: Oblivious Network Measurement as a Cloud Service

Shangqi Lai*, Xingliang Yuan*, Joseph K. Liu*, Xun Yi†, Qi Li‡§, Dongxi Liu¶ and Surya Nepal¶
*Faculty of Information Technology, Monash University, Australia
†School of Computer Science and Information Technology, RMIT University, Australia
‡Institute for Network Sciences and Cyberspace, Tsinghua University, China
§ Beijing National Research Center for Information Science and Technology (BNRist), China
¶Data61, CSIRO, Australia
Email: {shangqi.lai, xingliang.yuan, joseph.liu}@monash.edu, xun.yi@rmit.edu.au
qli01@tsinghua.edu.cn, {dongxi.liu, surya.nepal}@data61.csiro.au

***Abstract*—Network function virtualisation enables versatile network functions as cloud services with reduced cost. Specifically, network measurement tasks such as heavy-hitter detection and flow distribution estimation serve many core network functions for improved performance and security of enterprise networks. However, deploying network measurement services in third-party multi-tenant cloud service providers raises critical privacy and security concerns. Recent studies demonstrate that leaking and abusing flow statistics can lead to severe network attacks such as DDoS, network topology manipulation and poisoning, etc.**

In this paper, we propose OblivSketch, an oblivious network measurement service using Intel SGX. It employs hardware enclave for secure network statistics generation and queries. The statistics are maintained in newly designed oblivious data structures inside the SGX enclave and queried by data-oblivious algorithms to prevent data leakage caused by access patterns to the memory of SGX. To demonstrate the practicality, we implement OblivSketch as a full-fledge service integrated with the off-the-shelf SDN framework. The evaluations demonstrate that OblivSketch consumes a constant and small memory space (6MB) to track a massive amount of flows (from 30k to 1.45m), and it takes no more than 15ms to respond six widely adopted measurement queries for a 5s-trace with 70k flows.

I. Introduction

Network Function Virtualisation (NFV) decouples network functions from hardware appliances and pushes forward the deployment of software network functions in the cloud [6], [23], [66]. Enterprises can subscribe to versatile network functions from cloud service providers with high scalability and reduced cost. Among others, network measurement tasks [49], [77] such as estimating flow frequencies, tracking heavy-hitters, and counting distinct flows are crucial to improved performance and security of enterprise networks. They serve numerous core network functions such as traffic engineering, load balancing, and anomaly detection.

Despite this growing adoption, the provisioned network measurement services in third-party multi-tenant cloud service providers raise critical privacy and security concerns. Those pivotal services analyse global flow statistics collected from enterprise networks. Such information (e.g., flow counts and network topology) can be proprietary and private to an enterprise [43] and should be kept confidential at any time from the cloud service provider or other co-tenants. Moreover, flow statistics are important auxiliary information which can be exploited by several network attacks. Recent research has demonstrated that leaking and abusing network statistics can facilitate DDoS attacks [75], network topology manipulation and poisoning [37], [51], etc.

To guard network measurement services in the cloud, one direction is to design cryptographic protocols to evaluate network measurement tasks over encrypted flow data. Relevant attempts have been made via secure multi-party computation (SMC) [7], [14], [22], [43], [78] and homomorphic encryption [11]. Although these protocols provide provable security without hardware assumptions, they confront obstacles on unsatisfactory performance in networked contexts. For example, an SMC-based protocol takes over 10 minutes to extract heavy-hitters from only 4 thousand flows [43].

A practical alternative for securing network measurement services is to resort to hardware-assisted security, i.e., Intel SGX, a trusted execution environment. Due to its advantages on functionality and performance, SGX has recently been applied to develop a wide spectrum of secure networked applications and systems, e.g., IDS [27], [58], load balancer [31], [58], and firewall [27], [58]. At first glance, migrating existing implementation of network measurements into the SGX enclave would solve the problem. Unfortunately, several security, performance, and deployment challenges are yet to be resolved.

A. Technical Challenges

Challenges I: How to perform data-oblivious network measurement tasks? In practice, two data structures are commonly implemented to enable network measurement tasks, i.e., a *sketch* to estimate flow sizes and a *list* to cache flow IDs within an epoch of measurement [38], [49]. For instance, a network operator can use flow IDs from the list to retrieve flow sizes from the sketch for heavy-hitter detection. A naive

Network and Distributed Systems Security (NDSS) Symposium 2021
21-25 February 2021, Virtual
ISBN 1-891562-66-5
https://dx.doi.org/10.14722/ndss.2021.24330
www.ndss-symposium.org

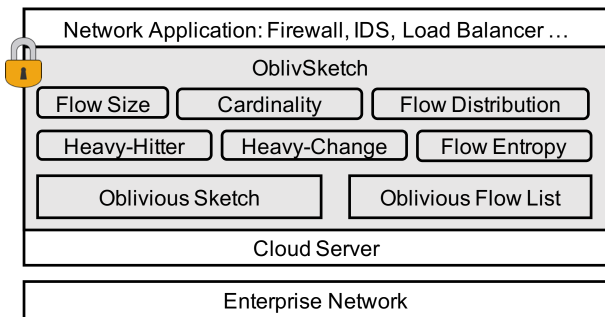

Fig. 1. OblivSketch handles network measurement queries via SGX-enabled oblivious primitives. It prevents leakages of flow statistics from traditional data breaches as well as memory access pattern side-channels against SGX.

solution for protecting flow sizes and IDs is to move the above data structures into the enclave; then network measurement tasks are performed inside it. However, it is not sufficient to guarantee the privacy of flow data. It is known that adversaries may exploit the memory access side-channels to recover the data even stored in enclaves [9], [13], [32], [45], [46], [60], [69], [76]. In our context, the memory access against the sketch and flow list is not data-oblivious. Furthermore, measurements require algorithms like sorting and swapping, and these algorithms implemented in the SGX standard library are also not data-oblivious. The above data-dependent access or execution patterns would be leveraged to infer the flow data and statistics.

Challenges II: How to perform oblivious network measurements for large networks efficiently? Practical network measurement services handle a large number of flows. A sketch is a randomised data structure which is able to track massive flows in low memory footprint; millions of flow sizes can be cached in a KB-level sketch with high accuracy [21]. Obviously, the memory consumption of sketch is negligible to the enclave. But keeping the list with all flow IDs inside would exceed the memory limit of SGX (96MB). As a result, this treatment will trigger SGX paging and lead to long delay on data access, i.e., can be 5× slower and even worse for memory-intensive tasks [65].

A straightforward approach to avoid paging is to store the encrypted flow list in the untrusted memory, where existing ORAM-like indexes based on SGX (e.g., Oblix [53] and ZeroTrace [59]) can be adopted to carry out secure and oblivious access of data. Those primitives relieve the enclave from the memory shortage but incurring new performance bottleneck. That is, transitions between the enclave and untrusted part are costly because they require encryption/decryption and memory I/O operations when passing data in between. A recent study [72] shows that such transition takes 3 - 10× higher delay compared with memory read/write within the enclave.

B. Contributions

In this paper, we propose OblivSketch, an oblivious network measurement service via Intel SGX (see Figure 1). OblivSketch can protect private flow data from the cloud and other co-tenants while performing network measurement tasks. It integrates newly proposed oblivious data structures and algorithms into the enclave to address powerful adversaries who can compromise the software stack and exploit memory access pattern leakages against SGX. Besides, the oblivious data structures are carefully customised to fit memory and I/O restrictions of the enclave for stringent performance requirements in network functions. We also make non-trivial system efforts to implement a full-fledged service that integrates with the Software-Defined Network (SDN) framework with six representative measurement tasks, i.e., flow size estimation, heavy-hitter detection, heavy-change detection, cardinality estimation, flow distribution estimation, and flow entropy estimation [49], [77].

Design Choice. Tackling the challenges above is non-trivial. We have made thoughtful design choices and built new oblivious primitives from the ground up to resolve those challenges simultaneously. First of all, we decide to deploy the oblivious data structures and algorithms for network measurements inside the enclave. The reason is that storing the encrypted flow list outside avoids SGX paging, but can still induce long query latency due to the adoption of ORAM for obfuscating external memory access. Note that even the flow data is moved inside the enclave, we also need to ensure that the computation and access against internal data structures are data-oblivious to defeat memory access side-channels [1], [25], [29], [53], [59].

Approaches. Starting from the above choice, we craft a new and compact oblivious data structure dedicated to network measurements, which can handle a large number of flows in a secure and scalable manner. Our insight is that not all flow IDs are needed during measurements. Particularly, network measurement tasks may analyse the sizes of all flows, but they only ask for the flow IDs of heavy flows, which are known to be rare within a certain period of time [5]. To this end, we consider using a fixed-size flow list to keep the flow ID of all heavy flows, along with a Count-Min sketch [21] that only keeps the sizes of other flows. Since the size of the flow list is small and fixed, linear-scan can be applied to obliviously insert and query the flow ID. To take the dynamics of network flows into consideration, we devise a secure eviction operation that can obliviously evict the flow from the flow list into the sketch. To make the sketch data-oblivious, we adapt the structure of Path ORAM[1] [63]. Note that the stash (an ORAM client data structure) is accessed via linear-scan, as suggested by existing SGX-based ORAM structures [53], [59]. But unlike prior works, the data in the ORAM is no longer encrypted by the ORAM protocol, since the entire ORAM is maintained inside the enclave. More details are provided in Section V-B.

To shield the entire process of network measurements, we further analyse the query algorithm of each measurement task and recognise several sub-routines that leak the access pattern, i.e., sorting, conditional-branching, swapping and direct memory accessing. We then transform the implementation of these sub-routines to corresponding trace-oblivious versions, i.e., sorting network [4], oblivious selector, oblivious swapping and linear-scan. More details can be found in Section V-C.

Requirements of Service Deployment. Only implementing the above primitives is not sufficient to deploy our network measurement service for enterprise networks. To demonstrate the practicality of our design, we develop a full-fledged service that can integrate with the modern network framework, i.e., SDN. There are two practical requirements for our design goal. First, our proposed primitives only harden the data security inside the enclave. The communications between other parties

[1]We explain the reason of adapting Path ORAM in Section III-B

of SDN (i.e., data planes and network applications) and the enclave need to be secured to protect the flow data from being collected to being processed. Second, the above requirement expects to be realised in a minimally intrusive way. Modifications in software switches and network applications should not affect other functionalities of SDN. The development efforts of using our service should be lightweight.

Implementation Efforts. OblivSketch provides an enclave deployed in the cloud. It receives the encrypted local statistics sent by the switch periodically. Upon receiving the encrypted query from network applications like load balancer and stateful firewall, the enclave processes the corresponding measurement tasks and returns the encrypted result to the application.

To fulfil the security requirement on flow data transmission, we design a *secure memory channel* and a *secure gateway* which jointly offer a transparent communication service between the enclave and other parties (i.e., switches, applications) of OblivSketch. Messages of our service will pass through the gateway to the memory channel on the measurement service, which can be assessed in an exit-less fashion by the enclave. We also make careful security considerations like hiding header information and payload sizes in the above designs (see Section VI-B for details).

We implement the data plane via Open vSwitch (OVS) with 180 lines of code modification. We also provide a library for network applications and software switches to invoke the secure gateway via only one API. For the controller, we integrate our design with the OpenFlow protocol and provide a library with three APIs for the controller to access the secure memory channel. Another 120 lines of code are added to let the controller process requests/responses to OblivSketch.

Snapshot of Results. We conduct extensive evaluations over OblivSketch on two CAIDA datasets [15] with 4 million and 8.9 million flows, respectively. OblivSketch features a constant and small memory consumption (6MB) for keeping the flow statistics within the enclave. We also evaluate the performance of each oblivious primitive invoked in OblivSketch. Our results indicate that the proposed oblivious data structure can insert and query a flow within 20μs. Moreover, all the oblivious algorithms, i.e., oblivious sorting, oblivious selector, oblivious swapping and linear-scan, take less than 12ms to finish under a 600KB oblivious data structure. Last, with 70k flows in our sketch (typical for the backbone trace in a 5s-period [15]), our service can respond to measurement queries in a timely fashion, i.e., no more than 15ms for each query. OblivSketch incurs 2 - 8$\times$ overhead compared to the non-oblivious baseline. When paging is triggered on the baseline, OblivSketch outperforms the baseline by 15$\times$ at most. Finally, OblivSketch is highly scalable: querying on a larger dataset does not introduce extra query delay on measurement tasks.

II. Related Work

Software-Centric Private Network Measurements and Routing Functions. There exist several protocols without relying on secure hardware which can securely analyse and monitor encrypted traffic data. Most of these existing designs are built upon secure multiparty computation (SMC) techniques [7], [14], [22], [43], [78]. They distribute the secret shares of the local statistics to a set of untrusted-but-non-colluded servers to compute an aggregated statistical result and respond to queries. However, the above protocols may incur long latency to process a query on a large number of flows. As a generic secure computation technique, SMC can be used to implement secure inter-domain routing protocols [2], [18], [35], which ensure that each domain cannot learn the routing information (e.g., routing tables and policies) of other domains. These protocols target on different network functions in routing, which are not our focus.

Another line of research adopts differential privacy [11], [28] to protect local statistics privacy against untrusted service providers. They add noise on local statistics and ask authorised parties [11] or SMC protocols [28] to aggregate the statistics. The untrusted service provider can publish the aggregated result but can hardly learn private information about local statistics. Those designs normally bring accuracy loss for aggregation tasks and do not protect the results by default.

Hardware-Assisted Secure Networked Systems. Our work is also related to secure networked systems based on Intel SGX [27], [31], [36], [58], [67]. With the help of trusted hardware, these systems can securely process network traffic for a variety of network functions, e.g., IDS [27], [31], [36], [67], firewall [27], [31], [58], load balancer [31], [58], NAT [58], etc. We note that the above systems do not consider mitigating side-channel attacks against Intel SGX.

SGX Side-Channels and Data-Oblivious Systems. In the past few years, many attacks targeting SGX have been developed. They can be divided into several categories, and one of the broad categories is the attacks on memory access patterns. This includes the attacks using page fault [13], [69], [76], cache timing [9], [32], [60], branch predication [46], and memory bus [45]. To mitigate these attacks, some system-level mitigations have been put forward (e.g., [42], [61]). Unfortunately, it could be difficult to address all sorts of the existing and future side-channels on memory access patterns with system patches, because that involves extra system-level work and may lead to new bugs in the system [41].

Recent studies [1], [19], [25], [29], [30], [53], [59] tend to design data-oblivious systems to mitigate the memory access pattern leakage. Specifically, those systems adopt oblivious data structures (e.g., ORAM [63]) and algorithms to hide the access pattern on data as well as the code, which are the core sources exploited by the above attacks. A wide range of applications have been implemented via oblivious systems, such as file system [1], database [29], collaborative analytics [25], and indexing system [53], [59]. Recent works propose generic programming frameworks [19], [30] that can compile any algorithm as an oblivious circuit running in the enclave. However, all the above systems are not designed and optimised for network measurements.

There are other side-channels such as speculative execution [12], [17], [50], communication patterns [55], power analysis [54] and others [34], [41]. We note that these attacks can be mitigated by some complementary works [8], [16], [20], [56]. More discussions are given in Section IV-B.

Plaintext Network Measurements. In the plaintext domain, extensive work [3], [38], [49], [77] has been done to enable

efficient and accurate network measurements over large-scale networks. These systems only work for unencrypted data.

III. Background

A. Intel SGX

Intel SGX [40] is a set of CPU instructions that offer a trusted execution environment (TEE) to the user application running upon the Intel CPU. SGX separates an application into an untrusted and trusted part, and it executes the trusted part in an isolated environment known as an *enclave*. The code and data within the enclave are isolated in a protected memory region called the *enclave page cache* (EPC). The other processes on the same CPU, including OS and hypervisor, cannot access the enclave memory and tamper its content. SGX also offers *remote attestation* (RA), which enables remote service providers (e.g., public clouds) to prove to a client that their enclave code is executed unaltered in the remote device. Also, RA establishes a secure channel between the client and the enclave to transmit secrets. We will discuss the threat model of SGX in Section IV-B. More details about Intel SGX can be found in [24].

B. Path ORAM

Path ORAM [63] is an ORAM protocol that allows a client to outsource data to the remote storage and access the data without leaking access patterns to the remote server. In this protocol, the server memory is arranged as a full binary tree with N tree nodes; each node contains Z blocks with equal size B. If a node has less than Z valid blocks, the node will be padded with dummy blocks to ensure that each node has a fixed size ($Z \times B$). The client maintains two data structures: a stash S, which hosts all blocks that have not yet been written to the server (can be new blocks or the blocks read from the server); a position map $\mathsf{position}$, which keeps the mapping between blocks and the leaf nodes in the binary tree.

The Path ORAM protocol comprises of four algorithms:

- $(T, \mathsf{position}, S) \leftarrow \mathsf{ORAM.Init}(N, Z, B)$: Given the tree node number N, node size Z and block size B, the server allocates $N \times Z \times B$ memory space and arranges it as a full binary tree T with $\lceil \log_2 N \rceil$ levels. The client generates an empty position map $\mathrm{position}$, a stash S and an encryption key k.
- $b \leftarrow \mathsf{ORAM.ReadBlock}(T, \mathsf{position}, bid)$: On input the binary tree T, the position map $\mathsf{position}$ and a block ID bid, the client gets the leaf node $\mathsf{lf} \leftarrow \mathsf{position}[bid]$ from $\mathsf{position}$. Then, it fetches all the blocks from the root of T to lf and inserts all these blocks into the stash S. The client finally reads the block b with the given bid from S and decrypts it with k. Meanwhile, it assigns a new random leaf node id to bid in $\mathsf{position}$ to ensure that the next access to this block goes to a random path.
- $\mathsf{ORAM.WriteBlock}(T, \mathsf{position}, S)$: To write the blocks in S back to T, the client gets the leaf node $\mathsf{lf} \leftarrow \mathsf{position}[bid]$ for each bid inside S. Later, it constructs nodes in the path between the specific leaf node lf and the root node. Specifically, for each block in S, the client scans the node from the leaf node level lf to the root node level and tries to fit the block in a node on these levels. If a node at the current level has enough space, the client uses k to encrypt the block with a randomised encryption scheme (AES-CBC) and evicts the encrypted block from the stash to that node. Otherwise, the block remains in the stash. Finally, the nodes are written back to the memory on the server.
- $\mathsf{ORAM.Access}(\mathsf{Op}, T, \mathsf{position}, S, bid, b)$: All Path ORAM accesses (Read/Write) consists of the above two operations. Particularly, the client firstly runs $\mathsf{ORAM.ReadBlock}(T, \mathsf{position}, bid)$ and gets a block from S. Then, if $\mathsf{Op} = \mathsf{Read}$, it copies the block to b; otherwise, it copies b to the block. Finally, the client runs $\mathsf{ORAM.WriteBlock}(T, \mathsf{position}, S)$ to write some blocks from S to T.

In the Path ORAM protocol, each access incurs path read/write operations in a tree path specified by the client. Since the accessed block will be re-assigned to a random tree path, the adversary on the server can only see a sequence of random accesses on the Path ORAM tree which is independent of real memory access patterns.

Path ORAM is one of the most efficient ORAM schemes in terms of computational cost. Since OblivSketch keeps the ORAM client and server in the enclave with limited size (see Section V-B for details), the storage cost is more important than the bandwidth cost. Thus, we do not consider the scheme such as SSS ORAM [64], which is bandwidth-efficient but occupying more memory. Moreover, we do not use secure multiparty computation and pursue an optimised ORAM circuit in this work, so we also exclude circuit ORAM [70].

C. Network Measurement and Sketches

We aim to design a secure network measurement service that supports general network measurement tasks based on network flows. Like the plaintext systems [38], [77], each flow in this paper is identified by a unique flow ID, which is a 5-tuple, i.e., protocol, source IP, source port, destination IP, destination port. We focus on collecting the following common flow-based network statistics, which is extensively studied in the plaintext literature [38], [77]. Note that these works are all based on sketches due to its high efficiency and high fidelity.

- **Flow size estimation**: reports an estimated size of any flow upon on a given flow ID.
- **Heavy-hitter detection**: reports the flow IDs with top-k flow sizes.
- **Heavy-change detection**: reports the flow IDs whose sizes are changed (increase or decrease) beyond a pre-set threshold T between two adjacent time windows.
- **Cardinality estimation**: reports the number of distinct flow IDs.
- **Flow distribution estimation**: reports the distribution of flow sizes.
- **Flow entropy estimation**: reports the entropy of flow sizes.

In this work, we employ the Count-Min Sketch (CMSketch) [21] to conduct measurement tasks. CMSketch offers a summarised view on the network statistics with a theoretical guarantee on its memory consumption and accuracy, i.e., A

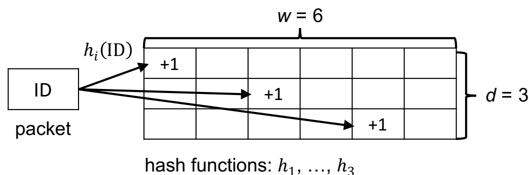

Fig. 2. Example of Count-Min Sketch.

40KB CMSketch with ten hash functions and 2000 32-bit counters achieves 0.01% error at the probability of 0.01%. As shown in Figure 2, a CMSketch comprises d arrays with w counters as well as d independent hash functions. For each new packet, the sketch hashes its flow ID (i.e., the 5-tuple) to a counter in each of the d rows and then increases the counters accordingly. To query flow size with a given flow ID, the sketch returns the minimum of d hashed counters in each row as an estimation. Intuitively, CMSketch supports the rest of measurement tasks by collecting all the per-flow size estimation. Note that other existing counter-based sketches [38], [77] can readily be implemented in our service. In this work, we use CMSketch to demonstrate our design, which is one of the widely adopted sketches for network measurements.

IV. Service Overview

A. Architecture

We consider that enterprises outsource the network measurement service to the cloud service provider. As shown in Figure 3, our targeted scenario comprises an enterprise network with switches, a cloud server with the network measurement service and network applications subscribed to by enterprises. Network applications like NAT, load balancer, and stateful firewall require network statistics to operate [44]. The cloud service maintains statistics of the entire enterprise network, i.e., a high-value target for adversaries [26], [43]. OblivSketch aims to hide flow statistics from the cloud server while providing essential network measurement services.

The workflow of OblivSketch is outlined below. The enterprise deploys standard traffic monitoring gadgets on each switch and maintains a set of counters associated with the flow. Before providing local statistics, the switch runs the remote attestation[2] (see Section III-A) to attest the integrity of remote modules in the server. That also establishes a secure channel between each switch and the enclave in the server, which allows the switch to share its secret key to the enclave. The cloud server periodically receives the encrypted local statistics from switches and passes them to a statistic module inside the enclave. This module maintains a small amount of memory as the oblivious sketch to aggregate the monitoring results, which provides a "one-big-switch" abstraction [44] for the whole network. After that, network operators can specify the monitoring tasks according to network applications. When an application asks for the statistics, it runs the attestation to check the integrity of the enclave and establish a secure channel in between. The application also sends its secret key to the enclave via the above secure channel. Upon receiving the query from an application, the statistic module queries the sketch and obliviously performs the measurement task to get the result. The encrypted result is returned to the application.

[2]Switches do not require to equip with SGX-enabled hardware for the remote attestation. They rely on Intel's attestation service to verify the enclave.

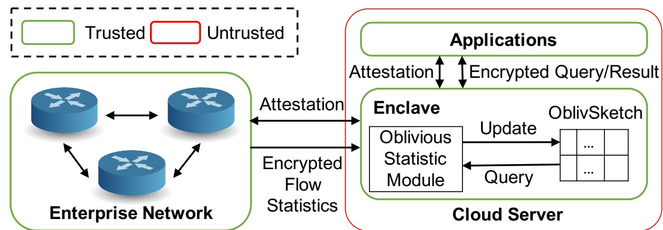

Fig. 3. Service Overview of OblivSketch.

B. Threat Model

In our scenario, essential statistical flow data regarding enterprise networks is supplied to untrusted third-party services. Therefore, enterprises expect to protect the private flow statistics from the cloud or other co-tenants. Besides, exposure of flow statistics can lead to severe consequences, e.g., creating opportunities and surfaces for emerging network attacks [10], [37], [51], [74], [75]. For example, revealing the network topology, flow statistics and traffic forwarding behaviours makes DDoS attacks much easier to launch. We aim to keep flow statistics confidential throughout the network measurement tasks, which is crucial to the security of networks.

Meanwhile, we address a powerful adversary under Intel SGX, who fully controls the user programs and even OS of the cloud server. He can see the entire memory trace of the network measurement service, and obtain and tamper any communication between the enterprise network and the cloud, or the network measurement service and network applications. The only exception is that the adversary cannot access the processor and collect information from it. The goal of OblivSketch is to operate network measurement tasks *obliviously* in the presence of the above adversary. The term "obliviously" indicates that the adversary cannot learn the underlying contents of network statistics and query results from the memory access pattern during the entire procedure of network measurement services. The above security level mitigates a combination of software side-channel attacks on cache [9], page table [13], [76], and branch history [46] under generic TEEs, including SGX.

There exist some other side-channel attacks against TEEs. Particularly, those attacks are based on speculative execution [12], [17], [50], communication patterns [55], denial-of-service [34], power analysis [54] and vulnerabilities in implementations [41]. Those attacks are out of the scope of OblivSketch. Nonetheless, existing countermeasures such as resolving race-conditions in hyperthreading [16], data shuffling/padding [55], [56] and SGX code analysis tool [20] can be readily applied to handle these attacks. Moreover, hardware enclaves can be updated in the microcode or hardware-level [8], [24] to tackle the above attacks.

We do not assume threats on software switches since enterprises can configure their switches as "only approachable via a strict policy" [62]. We do not focus on malicious applications such as infiltrating shared data stores to get sensitive information or disrupting other services [26], [68]. We assume that applications are subscribed and authenticated by network operators who can decide which application(s) can access measurement results [6].

C. Strawman Design

We first present a strawman design and highlight why it fails to meet both the security and efficiency requirements. A naive approach of designing a network statistic module is to maintain a flow list consisting of $(\mathrm{ID}, \mathrm{occurrence})$-tuple within the enclave. In detail, this module first extracts the ID in the $(\mathrm{ID}, \mathrm{counter})$-tuple collected from enterprise networks and puts it in the flow list. Then, the module computes the hash of ID and refers to the counter to update the sketch. At the end of each epoch, the statistic module uses all received flow IDs to query the sketch and derive the occurrence of each flow. The above module can enable the measurement tasks in Section III-C with trivial operations.

Limitations. Although the strawman design is easy to implement, it faces security and efficiency issues. First, it cannot reach the desired security level due to the current limitation of SGX. The reason is that the access on the sketch or flow list is deterministic as it is based on hash functions. Thus, adversaries can conduct inference attacks [9], [13], [32], [45], [46], [60], [69], [76] from the above access pattern within the enclave. Those attacks allow adversaries to gain rich information about network statistics. Second, storing a flow list with all flow IDs can be prohibitively burdensome to the enclave memory. Recall that SGX only has 96 MB protected memory for applications, if an application consumes more than 96 MB memory, the extra memory will be loaded on-demand via EPC paging.

V. PRIMITIVES

We first overview OblivSketch's core primitives to illustrate how they achieve data-obliviousness while ensuring the efficiency of network measurements. First of all, we design an oblivious data structure, named *oblivious sketch*, to support all relevant data accesses obliviously with a small, constant memory cost. The oblivious sketch consists of two parts: an OCMSketch is a CMSketch stored within the Path ORAM structure, and an oblivious bucket (OBucket) is a fixed-size flow list storing the flow ID and size of heavy flows. Both parts are kept inside the enclave.

In OCMSketch, the counters of CMSketch are stored in the Path ORAM tree. To access these counters within the enclave, we deploy an oblivious Path ORAM client, which runs normal ORAM access operations on the tree but leveraging linear-scan to access the stash of Path ORAM. As a result, the access on OCMSketch is fully oblivious in the view of untrusted servers. OBucket contains a fixed number of entries, and these entries are also placed in the Path ORAM tree. Each entry has multiple buckets to store $(\mathrm{ID}, \mathrm{counter})$-tuples. The enclave retrieves the corresponding entry and uses linear-scan to obliviously access the target bucket when accessing it. We also design an oblivious eviction strategy to capture the dynamics of network flows. The eviction performs linear-scan on the updated entry and also obliviously evicts a flow, which will be inserted into OCMSketch (see Section V-B for details).

Our proposed sketch natively supports the oblivious flow size query. For other queries, we utilise data-oblivious algorithms to compute over the data fetched from the sketch. In particular, we leverage oblivious sorting to enable heavy-hitter detection. For heavy-change, cardinality, flow distribution and flow entropy, we employ linear-scan algorithms for both OCMSketch and OBucket to ensure these queries are data-oblivious (see Section V-C for details).

Algorithm 1 OREAD

Input: The key-value list C; the search key k
Output: A value $C[k]$, 0 if $C[k]$ does not exist in C
OREAD(C, k)

1: $res \leftarrow 0$
2: **for** i = 1 : $|C|$ **do**
// if the current $C[i].key = k$, assign $C[i].value$ to res; otherwise, keep the recent res
3: $res \leftarrow oselector(C[i].value, res, C[i].key = k)$
4: **End for**
5: **Return** res

Algorithm 2 OSWAP

Input: The input value x and y; the control bit b
OSWAP(x, y, b)

// if $b = 1$, run $t \leftarrow x, x \leftarrow y, y \leftarrow t$ obliviously; otherwise, run $t \leftarrow y, x \leftarrow x, y \leftarrow y$ obliviously
1: $t \leftarrow oselector(x, y, b)$
2: $x \leftarrow oselector(y, x, b)$
3: $y \leftarrow oselector(t, y, b)$

We next provide the detailed design of the oblivious network measurement primitives. In the rest of the section, we use an oblivious operation $oselector(x, y, b)$ which runs $(x \,\&\, b) \mid (y \,\&\, \neg b)$ to select x or y as the output obliviously. Specifically, $oselector(x, y, b)$ outputs x if the control bit $b = 1$; otherwise, it outputs y. We also utilise the bitonic sort [4] with $O(n \log^2(n))$ complexity of sorting n elements. This sorting algorithm executes a fixed-sequence of comparisons for any given size of inputs (i.e., trace-oblivious) so we use it in OblivSketch (*osort*). In addition, some basic oblivious algorithms like oblivious reading (OREAD), swapping (OSWAP) are given in Section V-A.

A. Basic Algorithms

Algorithm 1 outlines the oblivious read procedure in the stash of Path ORAM. This algorithm enables the client of Path ORAM to access the stash obliviously.

Algorithm 2 outlines the oblivious swapping algorithm. It ensures that each element will be accessed equivalently during the swap process, and it is unknown for the adversary whether the input x and y are swapped. We employ this algorithm in OBucket (see Algorithm 4) to obliviously swap the input flow ID and the minimal flow ID in each entry.

B. Oblivious Sketch

We present the construction of the oblivious sketch, i.e., OCMSketch and OBucket. For each part, we explain the underlying data structure and the corresponding algorithms for insertion and query.

Oblivious CMSketch. OCMSketch exactly follows the CMSketch algorithm to insert and query flow information. The only difference is that we customise a Path ORAM tree to store the counters of CMSketch. In particular, we do not specifically encrypt the data in the Path ORAM tree. Instead, we rely on the

Algorithm 3 OCMSketch

Input: The number of counters w; the number of hash functions d; ORAM parameter Z and B

Output: The Path ORAM structure $(T, \mathsf{position}, S)$; a hash function set $\{H_i\}_{i=1}^d$;

SETUP(w, d, Z, B)

1: $(T, \mathsf{position}, S) \leftarrow \mathsf{ORAM.Init}(d \times w, Z, B)$
2: Choose d hash functions that map their inputs to [1, w], [w+1, 2w], ..., $[(d-1) \times w + 1, d \times w]$, respectively
3: **Return** $(T, \mathsf{position}, S)$ and $\{H_i\}_{i=1}^d$

Input: The flow ID *flowID* and its value v; a hash function set $\{H_i\}_{i=1}^d$; the Path ORAM structure $(T, \mathsf{position}, S)$

INSERT(*flowID*, v, $\{H_i\}_{i=1}^d$, $(T, \mathsf{position}, S)$)

1: **for** i = 1 : d **do**
2: Compute $bid \leftarrow H_i(flowID)$
3: Retrieve b from $\mathsf{ORAM.Access}(\mathsf{Read}, T, \mathsf{position}, S, bid, b)$
4: $b.value \leftarrow b.value + v$
5: $\mathsf{ORAM.Access}(\mathsf{Write}, T, \mathsf{position}, S, bid, b)$
6: **End for**

Input: The flow ID *flowID*; a hash function set $\{H_i\}_{i=1}^d$; the Path ORAM structure $(T, \mathsf{position}, S)$

Output: An estimated flow size $\hat{v}$

QUERY(*flowID*, $\{H_i\}_{i=1}^d$, $(T, \mathsf{position}, S)$)

1: $\hat{v} \leftarrow$ INT.MAX
2: **for** i = 1 : d **do**
3: Compute $bid \leftarrow H_i(flowID)$
4: Retrieve b from $\mathsf{ORAM.Access}(\mathsf{Read}, T, \mathsf{position}, S, bid, b)$
5: $\hat{v} \leftarrow \mathrm{MIN}(\hat{v}, b.value)$
6: **End for**
7: **Return** $\hat{v}$

SGX memory encryption engine [40] to protect the memory space allocated for OCMSketch. Note that linear-scan is also data-oblivious to access the sketch, but Path ORAM is more efficient as its complexity is $\mathcal{O}(\log n)$, while linear-scan is $\mathcal{O}(n)$ when there are n data blocks to be stored.

The original Path ORAM protocol considers its client as a trusted party, so the client-side routine is not oblivious. But in OCMSketch, the statistic module plays the role of the Path ORAM client, and the module is a part of the enclave. When the statistic module updates OCMSketch or retrieves results for queries, the client has a deterministic process based on the block ID when accessing the blocks in the stash, which leaks the pattern about those IDs. Hence, it is important to re-design the client to make it data-oblivious. Therefore, we replace the above process with linear-scan. After fetching blocks from a tree path to the stash, the statistic module will find the requested block by visiting the entire stash (cf. Algorithm 1). This approach raises a concern about its efficiency: if there are many blocks in the stash, the linear-scan will be expensive. To address this issue, we use a fixed-size stash rather than an unlimited stash. As in [63], one can use a stash with 105 blocks at most by tolerating a negligible false positive rate (2^{-128}) when each tree node contains five blocks ($Z = 5$).

As shown in Algorithm 3, the ORAM is initialised with $d \times w$ tree nodes (same size as the plaintext CMSketch). Also, the hash function maps its input to a value between 1 to $d \times w$, which is treated as the block ID bid for the ORAM to read and write blocks. Such a mapping enables OCMSketch to access

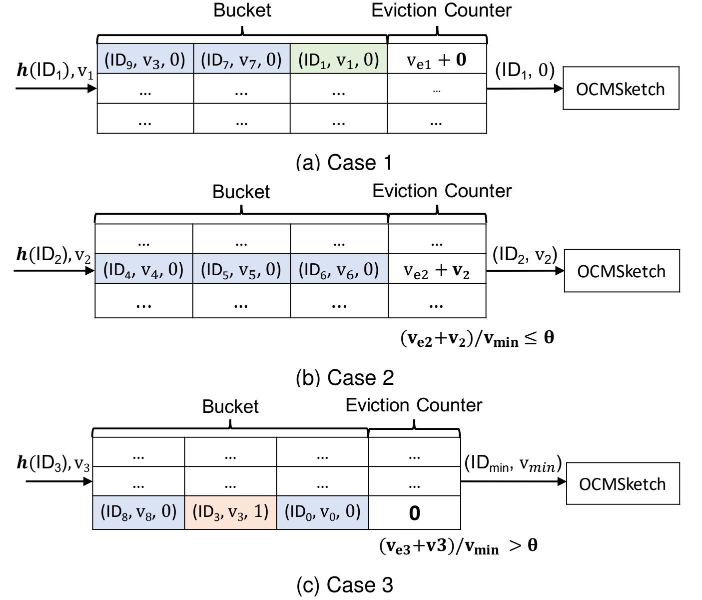

Fig. 4. The structure of OBucket and the eviction when inserting a new flow. During insertion, OBucket scans all coloured buckets in a selected entry. Then, it evicts a flow based on the eviction counter and availability of buckets. The above three figures cover the case when (a) an empty bucket is available; (b) no available bucket and cannot find a small flow to evict; (c) no available bucket but one smaller flow is evicted to fill the new flow (**bold** in counters denotes changes after the operation).

each underlying counter value obliviously since it is now stored in the Path ORAM tree. The detailed Path ORAM operations are given in Section III-B.

Oblivious Bucket. Since only the flow IDs of heavy flows are needed by common measurement tasks [77], we devise *OBucket* to keep those flow IDs. As shown in Figure 4, the data structure is split into multiple entries, and each entry has multiple buckets to store the flow information. Each bucket consists of three elements: the flow ID, its count and a tag indicating whether the eviction has happened on it. Note that OBucket keeps multiple buckets within the same entry to reduce the collision rate when inserting bulk flow information. Note that this strategy has been employed in plaintext measurement systems such as [77].

To make OblivSketch adaptive to the dynamics of the flow size, the last bucket of each entry is reserved for storing an '*eviction count* (v_e)'. It is increased when OblivSketch fails to insert a flow into the bucket. Once the ratio of $v_{eviction}$ to the minimum size in the entry v_{min} exceeds a threshold θ, the corresponding flow ID$_{min}$ will not be considered as a heavy flow and will be moved to OCMSketch.

To ensure the obliviousness of OBucket, all entries are also kept in Path ORAM, and the selected entry will be fully scanned when inserting or querying a given flow. The above eviction covers three cases as shown in Figure 4: 1) there is an empty bucket, or the flow ID$_1$ exists on a bucket. Then, OBucket updates the bucket and outputs (ID$_1$, 0); 2) there is no empty bucket, and OblivSketch cannot evict a bucket because of $\frac{v_{e2}+v_2}{v_{min}} \leq \theta$. Then, OBucket updates the eviction bucket by adding v_2 and outputs (ID$_2$, v_2); 3) there is no empty bucket, but OblivSketch finds a bucket to evict ($\frac{v_{e3}+v_3}{v_{min}} > \theta$). OBucket replaces the (ID$_{min}$, v_{min}) with (ID$_3$, v_3) and sets tag_3 to 1. Then it resets v_{e3} to 0 and outputs (ID$_{min}$, v_{min}).

Algorithm 4 Oblivious Bucket

Input: The number of entries d; the number of buckets for each entry L; the bucket size s; ORAM parameter Z
Output: the Path ORAM structure $(T, \text{position}, S)$; a hash function set H;
SETUP(d, L, s, Z)

1: $(T, \text{position}, S) \leftarrow \text{ORAM.Init}(d, Z, L \times s)$
2: Choose a hash function that maps its input to [1, d]
3: **Return** $(T, \text{position}, S)$ and H

Input: The flow ID *flowID* and its value v; a hash function H; the OBucket parameter L; the Path ORAM structure $(T, \text{position}, S)$; the swap threshold θ
Output: An evicted flow ID $swap_key$ and its value $swap_value$
INSERT(*flowID*, v, H, L, $(T, \text{position}, S)$, θ)

1: Compute $bid \leftarrow H(flowID)$
2: Retrieve b from $\text{ORAM.Access}(\text{Read}, T, \text{position}, S, bid, b)$
// flag indicating whether the insertion is successful
3: $success \leftarrow false$
4: $min_val \leftarrow b[0].value$
// scan the entire entry and try to fill one bucket of it
5: **for** i = $1 : L-1$ **do**
// flag indicating whether the current bucket is empty
6: $empty \leftarrow (b[i].key = 0)$ // true or false
// if the current bucket is empty and no successful insertion, flowID and value will be inserted; otherwise, the current bucket remains unaltered
7: $b[i].key \leftarrow b[i].key | (flowID \& (empty \wedge \neg success))$
// if flowID is not found, $b[i].value+ = 0$, else $b[i].value+ = v$
8: $found \leftarrow (b[i].key = flowID)$
9: $b[i].value \leftarrow b[i].value + found \ \& \ v$
// update min_val if the b[i].value is smaller; otherwise, keep the current min_val
10: $min_val \leftarrow oselector(min_val, b[i].value, min_val \leq b[i].value)$
// insertion is successful if an empty bucket or the repeated flowID is found
11: $success \leftarrow success \vee (empty \vee found)$
12: **End for**
13: $swap_key \leftarrow ID$
// if insertion is successful, set swap value to 0 (no swap is needed); otherwise, set it to the input flow value v
14: $swap_value \leftarrow oselector(0, v, success)$
// if insertion is successful, $b[L].value \leftarrow b[L].value + 0$, else $b[L].value \leftarrow b[L].value + v$
15: $b[L].value \leftarrow b[L].value + v \ \& \ \neg success$
//swapping is needed when no successful insertion happened and the eviction count is large enough
16: $swap \leftarrow \neg success \ \& \ (\frac{b[L].value}{min_val} > \theta)$
17: $swap_success \leftarrow false$
// scan the entire entry and substitute the bucket with the minimal flow size
18: **for** i = $1 : L-1$ **do**
19: $target \leftarrow (min_val = b[i].value)$
// perform swap if: swapping is needed, current bucket contains the minimal flow size and no swap was performed
20: $\text{OSWAP}(b[i].key, swap_key, \neg swap_success \wedge swap \wedge target)$
21: $\text{OSWAP}(b[i].value, swap_value, \neg swap_success \wedge swap \wedge target)$
// set tag to 1 if the current bucket is swapped
22: $b[i].tag \leftarrow oselector(1, b[i].tag, \neg swap_success \wedge swap \wedge target)$
// swap is successful if swap is needed and the current bucket contains min_val
23: $swap_success \leftarrow swap_success \vee (swap \wedge traget)$
24: **End for**
// if swap is successful, reset $b[L].value$, else keep the recent $b[L].value$
25: $b[L].value \leftarrow oselector(0, b[L].value, swap_success)$
26: $\text{ORAM.Access}(\text{Write}, T, \text{position}, S, bid, b)$
27: **Return** $swap_key$, $swap_value$

Input: The flow ID *flowID*; a hash function H; the Obucket parameter L; the Path ORAM structure $(T, \text{position}, S)$
Output: A bucket entry f
QUERY(*flowID*, H, L, $(T, \text{position}, S)$)

1: Compute $bid \leftarrow H(flowID)$
2: Retrieve b from $\text{ORAM.Access}(\text{Read}, T, \text{position}, S, bid, b)$
3: $f.key \leftarrow flowID$, $f.value \leftarrow 0$, $f.tag \leftarrow 0$
// scan the entire entry and obliviously get the query result
4: **for** i = $1 : L-1$ **do**
5: $f.value \leftarrow oselector(b[i].value, f.value, b[i].key = flowID)$
6: $f.tag \leftarrow oselector(b[i].tag, f.tag, b[i].key = flowID)$
7: **End for**
8: **Return** f

As shown in Algorithm 4, OBucket has three algorithms. The SETUP algorithm initialises the Path ORAM data structure with a pre-defined size of each bucket.

The INSERT algorithm can obliviously insert one flow into OBucket. In particular, the algorithm first computes the hash of the given flow and finds the corresponding entry from the ORAM. Then, it scans the selected entry and tries to insert the flow into one of the buckets in the entry. During this process, each bucket in the entry will be accessed, and which also obliviously finds the minimal flow in that bucket (line 5 - 12 in INSERT). The above process decides whether the flow is inserted or not, and this result is stored in a buffer obliviously (line 13 - 14 in INSERT). Later, the algorithm scans the entry again and obliviously swaps (cf. OSWAP in Algorithm 2) the input flow and minimal flow. Note that the flow is only swapped if it is not inserted and the ratio between the eviction counter and minimal flow size is larger than θ. Otherwise, the input flow is outputted as a result (line 18 - 27 in INSERT).

The QUERY algorithm can obliviously query a flow size in OBucket. Similar to INSERT, it computes the hash of the given flow and finds the corresponding entry. Then, it scans the whole entry and obliviously assigns the value and tag based on the input flow ID and the key of each bucket. The output flow will be filled with a positive value if it is found in the entry. Otherwise, it is set to 0.

Overall Protocol. We summarise the insertion and query protocols of the oblivious sketch as follows:

- **Insertion**: Upon receiving the flow information ((ID, counter)-tuples), OblivSketch computes the hash of the flow ID and retrieves the entry to insert in from the Path ORAM tree. Then it runs linear-scan on the entry to insert the flow information (cf. INSERT in Algorithm 4). After insertion, OBucket outputs a flow, which is inserted into OCMSketch (As in Figure 4).
- **Query**: Upon receiving the flow ID to query, OblivSketch computes the hash of the flow ID and gets the entry to search in OBucket. Then, it runs linear-scan on the entry to find the flow information (cf. QUERY in Algorithm 4). It finally retrieves a bucket from OBucket. Later, it queries OCMSketch with

the same ID to get $\hat{v}$ and sums the value from the bucket with $oselector(\hat{v}, 0, f.tag)$, i.e., accumulating the value from OCMSketch if eviction had happened.

The above procedure guarantees obliviousness during insertion and query on the sketch. The adversary who can observe memory traces sees the same (indistinguishable) access pattern for each insertion or query. However, he cannot determine where the actual flow information (i.e., in OBucket, OCMSketch or both) is and the relationship between any two flows (i.e., whether they are hashed into the same bucket/counter or not). The security definition and proof of the oblivious sketch are given in Appendix B.

Protocol Complexity. The complexities of insertion and query in OblivSketch comprise the insertion and query complexities on its two data structures (OCMSketch and OBucket). For each data structure, the insertion and query operations involve the following three essential operations: 1) hash function evaluation; 2) Path ORAM access[3]; 3) Linear-scan, and the complexities of them are: 1) $\mathcal{O}(1)$; 2) $\mathcal{O}(\log N)$, where N is the number of tree nodes in ORAM; 3) $\mathcal{O}(|X|)$, where $|X|$ is the size of the array X to be scanned.

For OCMSketch, its insertion and query protocols follow a similar procedure (see Algorithm 3): It firstly evaluates d hash functions. Then, it leverages these hash values as the key to retrieve the corresponding blocks from the ORAM; those blocks will be evicted to the ORAM after access (d ORAM accesses). For each retrieved blocks, OCMSketch should scan the stash S to read (for query) or update (for insertion) it to ensure the obliviousness. Thus, the time complexity of the above process is $\mathcal{O}(d + d \log N + d|S|)$.

The insertion and query on OBucket (see Algorithm 4) only need to evaluate one hash function and perform one ORAM access to retrieve an entry from the ORAM. Also, it scans the stash once to get the target entry. On the other hand, for each retrieved entry b, OBucket scans all buckets on that entry to hide the read/update pattern. Thus, the complexity of the above process is $\mathcal{O}(1 + \log N + |S| + |b|) \approx \mathcal{O}(\log N + |S| + |b|)$.

Accuracy. We briefly compare the accuracy of the oblivious sketch and the strawman. In particular, the accuracy of the above approaches is bounded by the number of packets stored in CMSketch. Since our design keeps heavy flows in OBucket, the number of packets in CMSketch is noticeably reduced. Therefore, the accuracy of our new protocol is highly improved compared with the strawman. We note that the above strategy, i.e., storing heavy and light flows separately to increase the accuracy, has been adapted in plaintext systems [49], [77]. We refer readers to [77] for the theoretical accuracy and proof of this improved approach.

C. Oblivious Network Measurement

We now present how OblivSketch supports the measurement tasks obliviously via the proposed primitives and operations.

Flow Size Estimation. It is straightforward to estimate the flow size with the query protocol of the oblivious sketch.

[3]The read/write operations in Path ORAM are the same process [63].

Heavy-Hitter Detection. To detect heavy-hitters, OblivSketch scans the whole OBucket and copies the flow ID and value into a map. Then, it calls *osort* to sort the values in the map and return the top-n (pre-defined parameter) as the heavy-hitters.

Heavy-Change Detection. To detect the heavy-changes, OblivSketch keeps the OBuckets of two consecutive epochs. These two OBuckets are scanned and copied to two maps, respectively. Later, OblivSketch takes all flows from the current epoch (denoted as (k_1, v_1)) and the flows from previous epoch (denoted as (k_2, v_2)) and computes $v' = oselector(|v_1 - v_2|, 0, k_1 = k_2)$. If v' is larger than the pre-defined threshold, it is then added into the heavy-change list.

The oblivious algorithms of the following three tasks are adapted from the native implementation in plaintext systems [77]. All tasks perform linear-scan on OCMSketch to get a counter distribution array $\{n\} = (n_0, n_1, ..., n_{255})$, where n_i is the number of counter with value i (cf. OCMSKETCHSCAN in Algorithm 6). Then, they use the distribution array to compute the following metrics.

Cardinality Estimation. OblivSketch leverages the linear counting algorithm [73] (see Algorithm 5) to estimate the number of flows from the distribution array. Then, it scans OBucket to count the number of flows. The sum of the above two results is the cardinality of network flows.

Flow Distribution Estimation. The distribution array reflects the flow size distribution of OCMSketch. OblivSketch updates it with the flow size in OBucket to get the flow size distribution of the current epoch. As shown in Algorithm 6 (OBUCKETSCAN), the flow size is recomputed with the flow information in OBucket. The new value is updated into the distribution array after dynamically resizing the array.

Flow Entropy Estimation. After computing the distribution, it is uncomplicated to compute the entropy based on the distribution. In particular, OblivSketch uses the distribution set $\{n\}$ to compute $m = \Sigma_{i=1}^{|\{n\}|} n_i$, and the entropy is computed as $-\Sigma_{i=1}^{|\{n\}|} \frac{n_i}{m} \log \frac{n_i}{m}$. The above process accesses the whole distribution set, so it is also an oblivious process.

Security. All the above protocols protect data access patterns with the help of the oblivious primitives. Our service only reveals which data structure is accessed (OBucket or OCMSketch) and how the enclave accesses this data structure (i.e., linear-scan or ORAM access). Given the pre-set parameters, all insertions and queries in every measurement epoch will result in the same data access patterns disregarding the change of network statistics. Therefore, the adversary can only infer the type of tasks running in the enclave but cannot learn the flow data or statistics. We provide a detailed security analysis on each measurement task in Appendix C.

VI. IMPLEMENTATION

In this section, we show how OblivSketch can be deployed as a plug-in service for modern network infrastructures. We incorporate OblivSketch with the SDN framework.

A. Implementation Overview

We first overview our implementation to show how it is seamlessly integrated with SDN. As shown in Figure 5a,

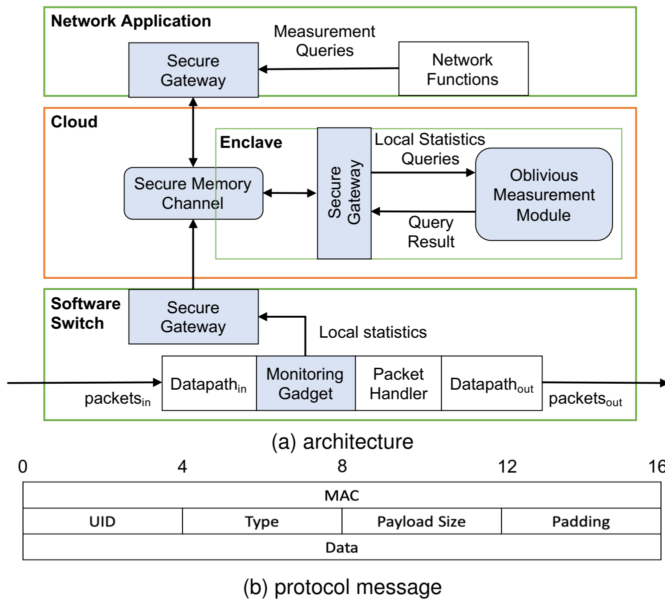

Fig. 5. The service implementation overview: (a) presents the architecture of OblivSketch service; (b) presents the formation of protocol messages transmitting in OblivSketch.

OblivSketch's implementation is involved with a network measurement daemon in the cloud, network applications running upon it and software switches. The network measurement daemon computes the statistics of network flows and handles queries from applications; the switch is responsible for collecting and reporting local statistics to the measurement daemon; the network applications query measurement statistics from the service to fulfil their requests.

Network Measurement Daemon. Our service runs a daemon program which comprises three components: a *secure memory channel* in its untrusted part, the *secure gateway* and *OblivSketch primitives* inside the enclave. The shared memory channel is a managed memory space sharing between the untrusted part and trusted part. In particular, it pairs with the secure gateway located in the switch, application and enclave. The gateway and shared memory channel jointly provide a transparent encryption and decryption service for all ingress and egress messages. In addition, this channel enables the network measurement daemon to communicate with the switch and network applications without exiting the enclave.

Software Switch. The software switch using our service consists of the ordinary SDN switch with the monitoring gadget and secure gateway. The monitoring gadget is deployed on each software switch and maintains a set of counters associated with the flows. It sends the local statistics in the form of $(\mathrm{ID}, \mathrm{counter})$ to the daemon via the secure gateway. To eliminate redundancy, the monitoring gadget is set to monitor the ingress packets only. Each switch also equips with the secure gateway, which transparently encrypts and transmits the encrypted statistics to the secure memory channel.

Network Applications. Network applications also utilise a secure gateway to communicate with the network measurement daemon. An application can send its query to the gateway and retrieve the corresponding result from it. All communication through it is encrypted.

Deployment. To deploy OblivSketch, the first step is to start up the adapted software switch and the network measurement daemon. In this work, we choose Open vSwitch [48] as our software switch due to its popularity. After that, switches run remote attestation to attest the integrity of remote modules in the enclave. Remote attestation also establishes a secure channel between each switch and the enclave, which is used to securely share the encryption key for the secure gateway and secure memory channel. When an application starts, it runs attestation (either local or remote, depending on where the application is) with the enclave and shares the encryption key via the secure channel.

Communication. We customise an application header to assist the network measurement daemon in deciding the measurement types and processing the updates from the switch. The structure of that header is in Figure 5b. It includes the following fields: 1) *MAC*: the MAC digest of a message; 2) *UID*: the unique identifier of an application, it is set to -1 if the message is from switches; 3) *Type*: the message type (discussed below); 4) *Payload Size*: the length of Data; 5) *Data*: the payload of the message. The switches and applications employ the above formatted message to communicate with the daemon, i.e., committing statistics, issuing queries and receiving responses. The message types are defined below to carry those messages: 1) STAT: sent by switches and encapsulates the encrypted statistics as payload; 2) FLOW_SIZE, HEAVY_HITTER, HEAVY_CHANGE, CARD, DIST, ENTROPY: sent by the application, requesting the result of the corresponding measurement type; the controller uses the same data type to reply; 3) STOP: sent by the administration application to shut down the daemon. Note that this header can be used independently when OblivSketch is deployed as an independent service. Also, it can integrate with the SDN protocols such as OpenFlow [57] as the request/response body of original protocols.

B. Secure Memory Channel

The goals of implementing a secure memory channel are to 1) minimise the transition cost between the enclave and the untrusted parts of the service; 2) hide the message information including the header from other untrusted parts of the service. To achieve the above goals, the secure memory channel is allocated on the untrusted memory space but shared with the enclave. The enclave thus enjoys an exitless fashion when sending and receiving messages from other parts of the service. Meanwhile, this channel maintains a pre-shared key in the enclave and leverages it to provide a transparent encryption/decryption service to the OblivSketch primitives. The primitives can read messages from and write messages to the channel without manually calling cryptographic operations.

Figure 6 illustrates the architecture of the secure memory channel. The secure gateway takes charge of encapsulating, decapsulating the message and performing cryptographic operations. Each switch and application will deploy a gateway that is paired with the one resided inside the enclave. The secure memory channel also implements a shared memory pool to allocate memory space for messages in advance. All the secure gateways request memory from the pool to mitigate the allocation cost on-the-fly.

Another core component in the secure memory channel is two queues for the messages: one for transmitting (TX)

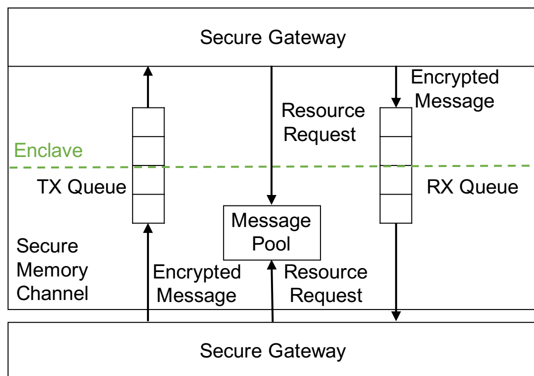

Fig. 6. The architecture of secure memory channel.

and the other for receiving (RX). These queues are initialised in the untrusted memory with the *user_check* parameter [39] provided by SGX SDK. The data structure with the *user_check* parameter can be shared as a raw pointer address accessible by the enclave and untrusted domain. In the secure memory channel, the pointers of TX and RX queues are shared among all secure gateways, including the one within the enclave. The secure gateway within the enclave will read the TX queue for any incoming messages and decrypt it into EPC for further processing. It also encrypts the processed query and puts it on the RX queue as the query output. Conversely, the secure gateways outside put encrypted updates/queries into the TX queue and receive results from the RX queue. The above process helps reduce the transition cost because the enclave no longer invokes an *ocall* when it sends a message out.

After initialisation, the secure gateways establish either a remote connection (TCP socket) or a local connection (in shared memory) to the secure memory channel. Each time the gateway sends messages, it firstly acquires memory from the message pool. Then, it segments and encrypts the message including the header via AES-GCM, and stores the MAC digest generated from AES-GCM in the MAC field of the header. The secure memory channel stands by to receive the message from the connection, fill it into the acquired memory space and push the message into the queue. Upon receiving a read request, the gateway decrypts the incoming message in the queue to the trusted memory space. Finally, it returns an emptied message to the pool for future use.

Security. Besides the security guarantee offered by our oblivious primitives, our service can thwart the adversary who targets the communication channel. In this work, proactive adversaries modifying the message will be detected via authenticated encryption. Particularly, the secure memory channel ensures that the adversary only learns the MAC digest of a message but nothing more. This is because the whole message, including the header, is encrypted. Also, the message is segmented to a pre-set size given by the message pool. Thus, the adversary capturing the unencrypted TCP packets cannot obtain the real size of a message.

C. Integration with the SDN Framework

We integrate OblivSketch with Open vSwitch (OVS) [48], one of the most widely used software switches. For the deployment of OVS, we embed the monitoring gadget into the datapath, which is responsible for receiving and routing all the packets. Upon receiving a packet, the monitoring gadget extracts the flow ID from the packet and puts it in a buffer. At the end of each monitoring epoch, it sends the local statistics to the secure gateway. To optimise the performance of OblivSketch, our service utilises the DPDK [47] datapath of OVS. The DPDK datapath allows the user-space program access to the NIC buffer directly, which eliminates the overhead from the memory copy and context switch between the kernel and user space. Meanwhile, our implementation aggregates the same flow ID on the local statistics before sending to the network measurement service. This treatment reduces the number of oblivious operations in the enclave by $10\times$ and highly improves the performance of sketch generation.

TABLE I. THE STATISTICS OF TEST TRACE WITH DIFFERENT TIME INTERVALS (K: THOUSAND, M: MILLION).

Intervals	1s	5s	10s	30s	60s	120s	240s
CAIDA1	30k	70k	100k	190k	280k	440k	690k
CAIDA2	42k	112k	175k	370k	590k	908k	1.45m

We slightly modified the secure gateway. It adds an OpenFlow header with the OFPT_EXPERIMENTER type upon the message described in Section VI-B. Then, it relies on the original network connections between switches/controller or applications/controller to send the OpenFlow packet to the controller. The controller will extract messages from OpenFlow packets, acquire memory space from the secure memory channel and transmit messages to the enclave for sketch generation and query processing. Upon receiving a result from the RX queue, the controller packs it with the OpenFlow header and sends it back to the application.

We implement the service depicted above[4]. The implementation consists of roughly 2700 lines of C/C++ code. This includes 300 lines of code modification in OVS, a library for applications and switches to access the secure gateway with one API call, and a library with three APIs for the controller to access the secure memory channel. Note that our realisation and implementation on the network measurement service can integrate with the existing SDN controller directly. As mentioned, it is an independent daemon program which can be deployed in the controller OS and approached by the secure gateway.

VII. EVALUATION

A. Setup

Platform. We deploy our service in an SGX-enabled workstation equipped with Intel Core i7-8850H 2.60GHz CPU (6 cores with multi-threading) and 32GB RAM. Both the service daemon and software switches are deployed on the above workstation, and the secure gateway establishes the local connection with the secure memory channel (i.e., shared memory). We also test our service after integrating it with the test-controller provided by OVS [57]. To eliminate the interference from other SDN messages in the evaluation, we implement a simulated switch, which only reads the offline traffic dump and sends it to the daemon periodically. Namely, it does not participate in any network communication such as routing, forwarding packets. We also deploy another secure gateway locally to simulate the application and send measurement queries to the daemon.

[4]Source code: https://github.com/MonashCybersecurityLab/measurement

Dataset. We use two one-hour traces collected in Equinix-nyc (CAIDA1) and Equinix-chicago (CAIDA2) monitor from CAIDA [15]. CAIDA1 includes 1.56 billion packets and 4 million flows with distinct sources (source IP), while CAIDA2 has 1.83 billion packets and 8.9 million flows. We divide the trace into various measurement time intervals (1s, 5s, 10s, 30s, 60s, 120s and 240s). The average number of flows for each interval is given in Table I. And we follow plaintext systems [77] to use 5s as the default measurement interval.

Baseline. We implement two baseline services for comparison. The first baseline (strawman baseline) adopts the strawman protocol described in Section IV-C to substitute the OblivSketch protocol in Figure 5, while the other components in OblivSketch remain unchanged. We compare the performance of the baseline with OblivSketch to show that our oblivious design is practical in the real-world network. Our comparison also shows that OblivSketch outperforms the strawman when the paging is triggered, as it brings enormous delays on tasks as shown in Section VII-C. Also, the performance of the heavy-hitter and heavy-change detection in OblivSketch is better than the strawman even if the paging is not triggered, because of OblivSketch keeps few flow IDs than that in the strawman.

The second one (SMC baseline) implements [43] to support flow size and heavy-hitter queries. As in [43], the SMC baseline equips two daemons that collect the secret sharing of local statistics and sort/merge them into a flow list with $(\mathrm{ID}, \mathrm{counter})$-tuple (as a list of secret shares). Then, it uses a garbled circuit to scan the entire list to get the flow size. For heavy-hitter queries, it implements the bitonic sorting as a circuit which sorts the flow list and returns the top-n result as the heavy-hitters. Thus, the SMC baseline is also data-oblivious. The SMC baseline is implemented with C/C++ and emp-toolkit (an optimised garbled circuit library) [71].

Sketch Settings. We set the default sketch size to 600KB for both the strawman and oblivious designs (src IP as flow ID). For the CMSketch in the strawman protocol, we follow the recommendation in [33] to use three hash functions. For the oblivious sketch, we use an 8-bit counter and one hash function in OCMSketch, as well as seven flow buckets in each entry of OBucket [77]. To ensure the accuracy of the heavy-hitter and heavy-change detection, we set OBucket to 150KB (see Section VII-D for a detailed analysis). OBucket has 2400 entries stored in a Path ORAM structure, and each ORAM node has five 64-byte blocks (64-byte is the size of entries in OBucket). In OCMSketch, we initialise a Path ORAM with 450K nodes, and each node has five 1-byte blocks. Also, we accept a negligible false positive rate (2^{-128}) in Path ORAM and the resulting size of the stash is 105. As mentioned in Section V-B, this setting noticeably improves the efficiency of the ORAM while retaining its security guarantee. We note that the memory consumption is much larger than the pre-defined parameter (600KB) due to the padding and position map of Path ORAM occupies extra space. Nonetheless, this is a fair comparison since most of the blocks in the ORAM are dummy blocks, and thus the valid blocks are still 600KB in total.

Remark: The above settings are dedicated to the large-scale network trace dataset we used (CAIDA dataset is collected from backbone networks). If the service is deployed in relatively small networks such as enterprise networks, the network administrator can use a smaller sketch. The query delay can be further reduced with the new sketch settings.

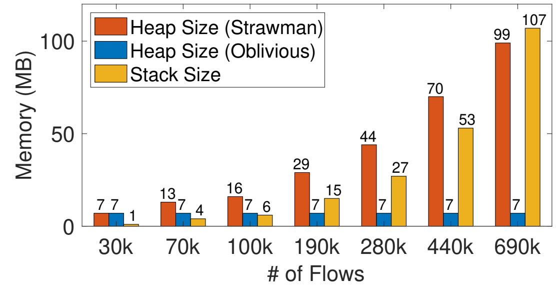

Fig. 7. The enclave memory consumption in strawman and OblivSketch.

Task Optimisations. For the measurement tasks like cardinality, flow distribution and flow entropy, processing those queries involve heavy computational costs, while their results are unchanged within an epoch. We observe that the flow distribution and flow entropy are estimated based on a counter distribution array. In addition, the cardinality query routine is highly overlapped with the counter distribution array generation process, so they can be calculated together. The process is costly as it performs linear-scan on OBucket and OCMSketch (see Algorithm 6). To improve the performance, OblivSketch precomputes the distribution array and cardinality when generating the sketch and saves it within the enclave. This optimisation introduces an additional 181 - 185ms delay upon precomputing the array. However, it brings a 50 - 170× speed-up on the query of the above three queries.

Evaluation Goals. The *major goal* of our evaluations is to show *OblivSketch can answer measurement queries before the current epoch elapsed.* This ensures that OblivSketch can provide a real-time measurement result regarding the underlying network. It is also consistent with the design goal of plaintext measurement systems [49], [77]. Meanwhile, we present a comparison between the strawman and OblivSketch for each individual query. The result demonstrates how our design mitigates the security and efficiency issues of employing Intel SGX with an acceptable cost from oblivious primitives.

The rest of this section is organised as follows: Section VII-B compares the memory usage of OblivSketch and the strawman. Then, Section VII-C demonstrates the micro benchmark of each oblivious primitives as well as the delay and throughput of each measurement task. Finally, Section VII-D examines the accuracy of OblivSketch.

B. Memory Management and Usage

Our first evaluation compares the enclave memory consumption of OblivSketch with the strawman design. In particular, we generate the sketch on the trace with different measurement time intervals. Here, we use the SGX Enclave Memory Measurement Tool (EMMT) [40] to track the memory usage within the enclave.

Figure 7 demonstrates the memory consumption in the heap and stack of the enclave, respectively. In particular, the heap keeps the computed statistics, and it resides in the memory for future queries; The stack will be released after the generation process, so it does not affect the performance of queries.

We can see that OblivSketch features a constant memory consumption on the heap (7MB), where the sketch is located.

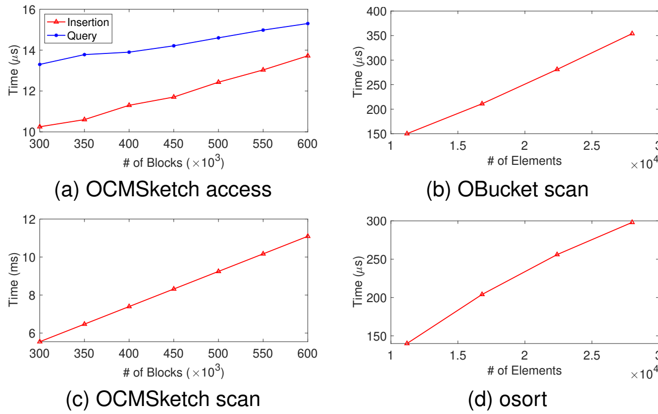

Fig. 8. Latency of oblivious data structure and function.

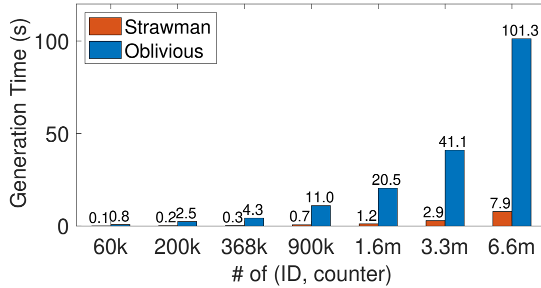

Fig. 9. Sketch generation time in strawman and OblivSketch.

On the other hand, the strawman design incurs an increasing demand for the heap memory when it computes long-term statistics because it keeps all flow IDs in the heap. The stack of the enclave is used to keep the incoming statistics from software switches, so it is a fixed value for a given measurement time interval.

Figure 7 also indicates that the memory consumption of the strawman design exceeds the SGX memory limit during the sketch generation when the measurement interval is 120s (123MB). This causes an extra delay for paging in the generation process. Obliviously, if the time interval is increased to 240s, the size of strawman's heap (99MB) also exceeds the SGX limit, and this leads to a noticeable delay on the following measurement queries. Compared to the strawman, although OblivSketch surpasses the memory limit when the time interval is 240s, it still achieves practical performance when processing queries, because paging will not be triggered after releasing the stack. More evaluation results about the generation cost and query delay are given in Section VII-C.

C. Measurement Performance

Micro Benchmarks on Primitives. We begin by evaluating the performance of our oblivious data structure and functions.

1) *oblivious sketch insertion and query*: The oblivious sketch consists of an OBucket and an OCMSketch. In the following evaluation, we evaluate their insertion and query performance separately. The overall insertion and query delay of the oblivious sketch can be simply computed by accumulating the access time on each part.

For OBucket, the access time is fixed under the fixed bucket size of each entry. In our default setting (150KB, 7 buckets per entry), it takes $10\mu s$ to insert and $3\mu s$ to query OBucket.

For OCMSketch, the insertion and query time depends on the performance of the underlying Path ORAM. Thus, we vary the size of the sketch (i.e., the number of blocks in the ORAM) and report the operation time under different block sizes. We observe from the result (Figure 8a) that our customised ORAM can respond the insertion and query within $20\mu s$, which means OblivSketch can access it frequently with a low delay. We also observe that the insertion is slightly slower than the query. This delay is consistent with our design, where the insertion needs to retrieve the corresponding counter for update and then write it back, while the query only reads the counter value.

2) *OCMSketch scan*: To evaluate the linear-scan cost, we run a sum function on both OBucket and OCMSketch to sum all values in these two parts. The evaluation results are listed in Figure 8b and Figure 8c. We find that the linear-scan on OBucket is finished within $400\mu s$ as it contains fewer elements. On the other hand, linear-scan on OCMSketch can take 5 - 10ms, which is the heaviest operation among all oblivious functions. Therefore, we further consider reducing the frequency of invoking this function. In the query delay evaluation, we will demonstrate that we can reduce the invocation of the linear-scan algorithm after applying the optimisation mentioned in Section VII-A. It improves the performance of queries (i.e., cardinality, flow distribution and flow entropy) that rely on linear-scan.

3) *osort*: Figure 8d evaluates the performance of *osort*. The number of elements in the table is corresponding to the size of OBucket (100KB, 150KB, 200KB and 250KB) in the protocol. The result shows that the *osort* can sort OBucket within $300\mu s$. Note that only a 200KB OBucket can achieve 100% precision and recall, so the evaluation reflects the performance of OblivSketch in practice.

4) *oselector*: The *oselector* can be finished within $0.01\mu s$, and the cost is negligible in our service.

Sketch Generation Time. As shown in Figure 9, both strawman and oblivious designs generate the sketch before the next statistics coming in. By doing so, they are able to answer the measurement query within each epoch. Again, the result demonstrates the impact of paging. We can see that the generation time increases disproportionally to the number of flows if the memory exceeds the SGX limit. For the strawman design, the generation time is 2.5 - 2.8$\times$ longer than the one on a smaller trace, even if the trace only enlarges by 2$\times$. The same phenomenon happens for OblivSketch if paging is triggered: the generation time slows down by 2.8$\times$ with a 2$\times$ larger trace. Nonetheless, we highlight that OblivSketch can process more flow information per epoch before paging is on, i.e., the strawman needs paging to process 3.3m flows, but OblivSketch does not. The above performance is sufficient for OblivSketch to monitor very large-scale networks [15].

Query Delay. We present the query delay incurred by each measurement task in Figure 10. First, our result further confirms the impact of paging during the query. In the strawman, the query delay surges when the sketch size exceeds the limit of the enclave. It makes the query slower than the corresponding oblivious query although it is theoretically faster. Next, we analyse the results of different measurement queries:

1) Flow Size: Both the strawman and OblivSketch cost a constant time when querying the flow size of a given flow ID

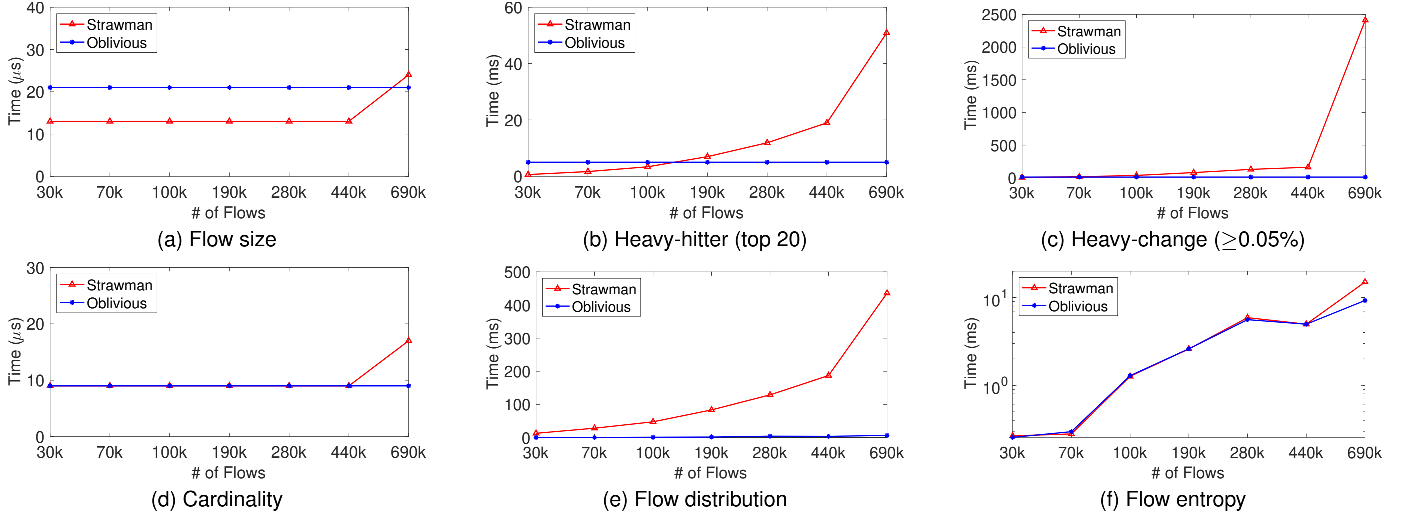

Fig. 10. The query delay of each measurement task.

(see Figure 10a). That is because our design is based on the sketch, and it has a constant query complexity as mentioned in Section III-C. Moreover, our service is 14% faster than the strawman when paging is triggered.

2) Heavy-Hitter: The query delay of heavy-hitter queries is in Figure 10b. We can see that the strawman design requires more time to compute the result while OblivSketch requires a constant time to process the query. This is because of our protocol has a fixed-size OBucket as the input of the sorting algorithm. Furthermore, in OBucket, the number of elements to sort is much smaller compared to the total number of flows as we can see in Section VII-A. Therefore, OblivSketch outperforms the strawman design, even if the complexity of the sorting algorithm in the strawman ($\mathcal{O}(n)$, n is the number of elements to sort) is lower than OblivSketch ($\mathcal{O}(n \log^2 n)$).

3) Heavy-Change: Similar to the heavy-hitter detection, heavy-change queries in OblivSketch take a constant time to process (see Figure 10c). In the strawman, the query delay increases with the size of the heap. In addition, it needs to linear-scan and compares two heaps, which is memory-intensive. Hence, the strawman triggers paging with a significant delay (15×).

4) Cardinality: The cardinality query of OblivSketch is much slower than that in the strawman since OblivSketch needs linear-scan on OBucket and OCMSketch to get the result, while the strawman only invokes the standard C++ function on the map. After applying the optimisation in Section VII-A, the query performance can be improved to the same magnitude as in the strawman (9μs).

5) Flow Distribution: As shown in Figure 10e, the distribution query delay is proportional to the number of flows due to the linear-scan algorithm in both implementations. Fortunately, the precompute strategy significantly boosts its performance: The enclave only needs to return the distribution array back now. It only involves encryption/decryption and transmission costs, which are negligible comparing to the linear-scan time.

6) Flow Entropy: Both implementations rely on the flow distribution to compute the entropy. Since the flow distribution is precomputed in OblivSketch, OblivSketch and the strawman achieve similar performance (see Figure 10f) except when paging is triggered (the strawman is 60% slower).

TABLE II. THE MEASUREMENT TASK THROUGHPUT (QUERY/EPOCH) COMPARISON BETWEEN THE STRAWMAN AND OBLIVSKETCH.

measurement interval: 5s

Measurement Tasks	Flow Size	Heavy-Hitter	Heavy-Change	Cardinality	Flow Distribution	Flow Entropy
Strawman	372317	2858	305	537792	172	17286
OblivSketch	112875	473	217	263376	13242	7981

measurement interval: 240s

Measurement Tasks	Flow Size	Heavy-Hitter	Heavy-Change	Cardinality	Flow Distribution	Flow Entropy
Strawman	9672583	4563	96	13655411	533	15460
OblivSketch	6606942	27700	12720	15416198	614	14993

After integrating OblivSketch with the SDN framework, we re-run the sketch generation time and query delay evaluations. We realise that the SDN framework introduces a 5ms extra delay when processing the above measurement queries. This extra cost results from the network round-trip communication as well as the protocol processing procedure in the controller.

Throughput. To confirm the practicality of our service, we measure the throughput of measurement tasks. Here, the throughput we measured is the number of queries that can be answered for the current epoch after generating the sketch. For each measurement task, we compare its throughput result between OblivSketch and the strawman. Table II presents the throughput result for OblivSketch and the strawman. In the first table, all the results are collected under the default settings (see Section VII-A for details). Our results show that even though the throughput of OblivSketch decreases to some extent, it can answer a significant number of queries. Note that the network management applications (i.e., routing, load-balancing, anomaly detection) will neither change the network parameter nor scan for anomaly frequently [52]. Hence, the throughput loss in OblivSketch is affordable: it is still viable to respond to a large number of queries (from 200 to 260k for the default settings) while it highly improves the security of measurement tasks.

The second table shows the throughput performance when the measurement interval is 240s, where the paging is triggered

TABLE III. The query delay of datasets with different sizes

measurement interval: 5s

Measurement Tasks	Flow Size	Heavy-Hitter	Heavy-Change	Cardinality	Flow Distribution	Flow Entropy
CAIDA1	20μs	5ms	10ms	9μs	179μs	301μs
CAIDA2	20μs	5ms	10ms	9μs	181μs	317μs

measurement interval: 240s

Measurement Tasks	Flow Size	Heavy-Hitter	Heavy-Change	Cardinality	Flow Distribution	Flow Entropy
CAIDA1	20μs	5ms	10ms	9μs	6ms	10ms
CAIDA2	20μs	5ms	10ms	9μs	6ms	10ms

TABLE IV. The runtime performance of the SMC baseline

Measurement Tasks	Generation	Flow Size	Heavy-Hitter
CAIDA1	65s	1.3s	18s
CAIDA2	70s	2.1s	32s

for the strawman. Under this setting, all the tasks throughput of OblivSketch are close to or higher than that in the strawman except the flow size. For the memory-intensive tasks like heavy-change, OblivSketch's throughput is 132.5× higher than the strawman. This result illustrates the impact of paging in a large network, and OblivSketch can effectively eliminate paging to achieve better performance for the long-term statistics.

Scalability. We leverage two CAIDA datasets with a different number of flows to demonstrate the scalability of OblivSketch. The evaluation is running with two measurement intervals: the first one is 5s, where CAIDA2 has 60% more flows than CAIDA1. The second interval is 240s, where the number of flows in CAIDA2 is doubled comparing to CAIDA1 (see Table I). The other settings (sketch parameters) remain unchanged. As shown in Table III, even the number of flows is doubled in CAIDA2, the query delay of OblivSketch does not change. The reason is that the query delay of OblivSketch is only affected by the sketch setting but not the traffic size. Note that the only exceptions are flow distribution and flow entropy queries since a larger measurement interval indicates a longer distribution array to be processed.

Compare with the SMC Baseline. Table IV illustrates the query delay of the SMC baseline when querying a 5s-trace. The delay includes the circuit generation and evaluation time. The table shows that the SMC baseline takes 1.3 - 2.1s when executing flow size queries. This is much slower than OblivSketch, which only needs 20μs. Thus, the SMC baseline can answer 2 - 3 queries only, and it cannot handle the query if there are more than three applications submitting flow size queries at the same time. For heavy-hitter queries, the SMC baseline requires 18 - 32s to respond a query on the trace of a 5s-interval. It indicates that the SMC baseline cannot respond to any heavy-hitter query before the current epoch elapsed. Note that the above result does not consider the generation time of the flow list, which is about 65 - 70s. After taking the generation time into consideration, the SMC baseline cannot answer any query within the 5s-epoch.

D. Accuracy

Finally, we perform a comparison of the accuracy on the flow size, heavy-hitter and heavy-change. We choose these three tasks because we observe that the measurement tasks in OblivSketch either rely on OBucket only (heavy-hitter and heavy-change) or the flow size (other tasks). Hence, the accuracy results of the selected tasks can depict the accuracy characteristics of OblivSketch precisely. We compare the accuracy of OblivSketch with the strawman as well as the plaintext sketches, i.e., CMSketch [21] and ElasticSketch [77]. Note that the CMSketch-based solution is exactly the same as the strawman except that it does not run within the enclave. Also, ElasticSketch [77] adopts the same design philosophy (split heavy flows and light flows and count them with different data structures) as OblivSketch. We perform the evaluation upon the CAIDA1 dataset. The evaluation shows that, with comparable memory usage, OblivSketch achieves higher fidelity than the strawman/CMSketch.

TABLE V. Accuracy comparison (ARE) between plaintext sketches, strawman and OblivSketch. We fix the size of OBucket to 150 KB.

Sketch size (KB)	200	400	600	800	1000
Strawman/CMSketch	6.93	2.84	1.61	1.04	0.73
OblivSketch	4	0.84	0.48	0.33	0.26
ElasticSketch [77]	3.97	0.81	0.44	0.31	0.27

We use the following metrics to measure the accuracy [77]:

- Average Relative Error (ARE): ARE can evaluate the accuracy of flow size estimations. It is computed as $\frac{1}{n}\Sigma_{i=1}^{n}\frac{|f_i-\hat{f}_i|}{f_i}$, where n is the number of flows, f_i is the flow size and $\hat{f}_i$ is the estimated flow size. If all estimated size is exactly the same as the real size, ARE is 0.
- F1 score: F1 score can evaluate the accuracy of the heavy-hitter and heavy-change detection. It is computed as $\frac{2\times PR\times RR}{PR+RR}$, where PR is the precision rate (the ratio of correct flows in the reported flows) and PP is the recall rate (the ratio of correct flows reported among all correct flows). If F1 score is 1, the estimation result is exactly the same as the real result.

For the accuracy of the heavy-hitter and heavy-change, OBucket achieves the same accuracy level as [77] when they use the same size of memory to keep heavy flows. When they use 150KB memory (our default setting, 19200 buckets) to keep the heavy flows, F1 score reaches 1 (100% accuracy). Hence, we fix the default setting on OBucket and evaluate the accuracy of the flow size estimation. In the flow size estimation, we vary the size of sketch from 200KB to 1000KB (note that only the OCMSketch of OblivSketch and the light part of [77] will be increased). We found that OblivSketch achieves better performance in accuracy: as shown in Table V, the ARE of OblivSketch is 3.5× smaller than the strawman/CMSketch under the default settings. Even though the sketch size is set to 1000KB, the ARE of OblivSketch is still 2.8× lower than the strawman/CMSketch. On the other hand, the accuracy of OblivSketch is close to [77] (less than 0.1 difference in ARE), whereas OblivSketch provides advanced security features.

VIII. Conclusion

In this paper, we propose OblivSketch, which is an oblivious and efficient network measurement service based on hardware enclaves. OblivSketch contributes customised oblivious data structures and algorithms that can integrate with the hardware enclaves to support a wide range of important network measurement tasks. OblivSketch service is integrated into

the existing SDN framework with minimised modifications. Finally, we leverage large-scale network traces to demonstrate its practicality.

ACKNOWLEDGMENT

The authors would like to thank the anonymous reviewers for their valuable comments and constructive suggestions. The work was supported in part by the Monash University Postgraduate Publications Award, the Data61-Monash Collaborative Research Project (D61 Challenge: E01), the ARC Discovery Projects (DP180102199, DP200103308), the NSFC Grant (61572278) and the BNRist Grant (BNR2020RC01013).

REFERENCES

[1] A. Ahmad, K. Kim, M. I. Sarfaraz, and B. Lee, "OBLIVIATE: A Data Oblivious Filesystem for Intel SGX," in *NDSS*, 2018.

[2] G. Asharov *et al.*, "Privacy-Preserving Interdomain Routing at Internet Scale," *Proceedings on Privacy Enhancing Technologies*, vol. 2017, no. 3, pp. 147–167, 2017.

[3] R. B. Basat, G. Einziger, S. L. Feibish, J. Moraney, and D. Raz, "Network-Wide Routing-Oblivious Heavy Hitters," in *ANCS'18*, 2018.

[4] K. Batcher, "Sorting Networks and their Applications," in *ACM SJCC'68*, 1968.

[5] T. Benson, A. Akella, and D. Maltz, "Network Traffic Characteristics of Data Centers in the Wild," in *ACM IMC'10*, 2010.

[6] T. Benson, A. Akella, A. Shaikh, and S. Sahu, "CloudNaaS: A Cloud Networking Platform for Enterprise Applications," in *ACM SoCC'11*, 2011.

[7] D. Bogdanov, L. Kamm, S. Laur, and V. Sokk, "Rmind: A Tool for Cryptographically Secure Statistical Analysis," *IEEE Transactions on Dependable and Secure Computing*, vol. 15, no. 3, pp. 481–495, 2016.

[8] T. Bourgeat, I. Lebedev, A. Wright, S. Zhang, and S. Devadas, "MI6: Secure Enclaves in a Speculative Out-of-Order Processor," in *MICRO'19*, 2019.

[9] F. Brasser *et al.*, "Software Grand Exposure: SGX Cache Attacks Are Practical," in *USENIX WOOT'17*, 2017.

[10] P. Bright, "Can a DDoS Break the Internet? Sure… Just Not All of It," https://arstechnica.com/information-technology/2013/04/can-a-ddos-break-the-internet-sure-just-not-all-of-it/ [online], 2013.

[11] J. W. Brown, O. Ohrimenko, and R. Tamassia, "Haze: Privacy-Preserving Real-Time Traffic Statistics," in *ACM SIGSPATIAL'13*, 2013.

[12] J. V. Bulck *et al.*, "Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution," in *USENIX Security'18*, 2018.

[13] J. V. 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 *USENIX Security 17*, 2017.

[14] M. Burkhart, M. Strasser, D. Many, and X. Dimitropoulos, "SEPIA: Privacy-Preserving Aggregation of Multi-Domain Network Events and Statistics," in *USENIX Security'10*, 2010.

[15] "The CAIDA UCSD Anonymized Internet Traces," https://www.caida.org/data/passive/passive_dataset.xml [online], CAIDA, 2018.

[16] G. Chen *et al.*, "Racing in Hyperspace: Closing Hyper-Threading Side Channels on SGX with Contrived Data Races," in *IEEE S&P'18*, 2018.

[17] ——, "SgxPectre Attacks: Stealing Intel Secrets from SGX Enclaves via Speculative Execution," in *IEEE EuroS&P'19*, 2019.

[18] M. Chiesa, D. Demmler, M. Canini, M. Schapira, and T. Schneider, "SIXPACK: Securing Internet eXchange Points Against Curious onlooKers," in *CoNEXT'17*, 2017.

[19] J. I. Choi *et al.*, "A Hybrid Approach to Secure Function Evaluation using SGX," in *AsiaCCS'19*, 2019.

[20] T. Cloosters, M. Rodler, and L. Davi, "TeeRex: Discovery and Exploitation of Memory Corruption Vulnerabilities in SGX Enclaves," in *USENIX Security'20*, 2020.

[21] G. Cormode and S. Muthukrishnan, "An Improved Data Stream summary: The Count-Min Sketch and its Applications," *Journal of Algorithms*, vol. 55, no. 1, pp. 58–75, 2005.

[22] H. Corrigan-Gibbs and D. Boneh, "Prio: Private, Robust, and Scalable Computation of Aggregate Statistics," in *USENIX NSDI'17*, 2017.

[23] P. Costa, M. Migliavacca, P. Pietzuch, and A. L. Wolf, "NaaS: Network-as-a-Service in the Cloud," in *USENIX HotICE'12*, 2012.

[24] V. Costan and S. Devadas, "Intel SGX Explained," Cryptology ePrint Archive, Report 2016/086, 2016.

[25] A. Dave, C. Leung, R. A. Popa, J. E. Gonzalez, and I. Stoica, "Oblivious Coopetitive Analytics Using Hardware Enclaves," in *EuroSys'20*, 2020.

[26] V. H. Dixit, A. Doupé, Y. Shoshitaishvili, Z. Zhao, and G.-J. Ahn, "AIM-SDN: Attacking Information Mismanagement in SDN-datastores," in *ACM CCS'18*, 2018.

[27] H. Duan *et al.*, "LightBox: Full-stack Protected Stateful Middlebox at Lightning Speed," in *ACM CCS'19*, 2019.

[28] T. Elahi, G. Danezis, and I. Goldberg, "PrivEx: Private Collection of Traffic Statistics forAnonymous Communication Networks," in *ACM CCS'14*, 2014.

[29] S. Eskandarian and M. M. Zaharia, "ObliDB: Oblivious Query Processing for Secure Databases," *Proceedings of the VLDB Endowment*, vol. 13, no. 2, pp. 169–183, 2019.

[30] S. Felsen, Á. Kiss, T. Schneider, and C. Weinert, "Secure and Private Function Evaluation with Intel SGX," in *ACM CCSW'19*, 2019.

[31] D. Goltzsche *et al.*, "EndBox: Scalable Middlebox Functions using Client-Side Trusted Execution," in *IEEE DSN'18*, 2018.

[32] J. Götzfried, M. Eckert, S. Schinzel, and T. Müller, "Cache Attacks on Intel SGX," in *EuroSec'17*, 2017.

[33] A. Goyal, H. D. III, and G. Cormode, "Sketch Algorithms for Estimating Point Queries in NLP," in *EMNLP-CoNLL'12*, 2012.

[34] D. Gruss *et al.*, "Another Flip in the Wall of Rowhammer Defenses," in *IEEE S&P'18*, 2018.

[35] D. Gupta *et al.*, "A New Approach to Interdomain Routing Based on Secure Multi-Party Computation," in *HotNets'12*, 2012.

[36] J. Han, S. Kim, J. Ha, and D. Han, "SGX-BOX: Enabling Visibility on Encrypted Traffic Using a Secure Middlebox Module," in *APNet'17*, 2017.

[37] S. Hong, L. Xu, H. Wang, and G. Gu, "Poisoning Network Visibility in Software-Defined Networks: New Attacks and Countermeasures," in *NDSS*, 2015.

[38] Q. Huang *et al.*, "SketchVisor: Robust Network Measurement for Software Packet Processing," in *ACM SIGCOMM'17*, 2017.

[39] "Refining The Enclave with Proxy Functions," https://software.intel.com/content/www/us/en/develop/articles/intel-software-guard-extensions-tutorial-part-7-refining-the-enclave.html [online], Intel, 2016.

[40] "Intel Software Guard Extensions (Intel SGX)," https://software.intel.com/en-us/sgx [online], Intel, 2019.

[41] J. J. Lee *et al.*, "Hacking in Darkness: Return-Oriented Programming against Secure Enclaves," in *USENIX Security'17*, 2017.

[42] J. J. Seo *et al.*, "SGX-Shield: Enabling Address Space Layout Randomization for SGX Programs," in *NDSS*, 2017.

[43] N. A. Jagadeesan *et al.*, "A Secure Computation Framework for SDNs," in *HotSDN'14*, 2014.

[44] N. Kang, Z. Liu, J. Rexford, and D. Walker, "Optimizing the "One Big Switch" Abstraction in Software-Defined Networks," in *CoNEXT'13*, 2013.

[45] D. Lee, D. Jung, I. T. Fang, C.-C. Tsai, and R. A. Popa, "An Off-Chip Attack on Hardware Enclaves via the Memory Bus," in *IEEE S&P'20*, 2020.

[46] S. Lee *et al.*, "Inferring Fine-grained Control Flow Inside SGX Enclaves with Branch Shadowing," in *USENIX Security'17*, 2017.

[47] "DPDK: Data Plan Development Kit," https://www.dpdk.org [online], Linux Foundation, 2020.

[48] "Open vSwitch," https://www.openvswitch.org [online], Linux Foundation, 2020.

[49] Z. Liu, A. Manousis, G. Vorsanger, V. Sekar, and V. Braverman, "One Sketch to Rule Them All: Rethinking Network Flow Monitoring with UnivMon," in *ACM SIGCOMM'16*, 2016.

[50] M. M. Schwarz *et al.*, "ZombieLoad: Cross-Privilege-Boundary Data Sampling," in *ACM CCS'19*, 2019.

[51] E. Marin, N. Bucciol, and M. Conti, "An In-depth Look Into SDN Topology Discovery Mechanisms: Novel Attacks and Practical Countermeasures," in *ACM CCS'19*, 2019.

[52] J. Medved, R. Varga, A. Tkacik, and K. Gray, "OpenDaylight: Towards a Model-Driven SDN Controller Architecture," in *IEEE WoWMoM'14*, 2014.

[53] P. Mishra, R. Poddar, J. Chen, A. Chiesa, and R. A. Popa, "Oblix: An Efficient Oblivious Search Index," in *IEEE S&P'18*, 2018.

[54] K. Murdock *et al.*, "Plundervolt: Software-based Fault Injection Attacks against Intel SGX," in *IEEE S&P'20*, 2020.

[55] O. Ohrimenko, M. Costa, C. Fournet, C. Gkantsidis, M. Kohlweiss, and D. Sharma, "Observing and Preventing Leakage in MapReduce," in *ACM CCS'15*, 2015.

[56] O. Oleksenko, B. Trach, R. Krahn, M. Silberstein, and C. Fetzer, "Varys: Protecting SGX Enclaves from Practical Side-Channel Attacks," in *USENIX ATC'18*, 2018.

[57] "OpenFlow Switch Specification," https://www.opennetworking.org/wp-content/uploads/2014/10/openflow-spec-v1.4.0.pdf [online], Open Networking Foundation, 2013.

[58] R. Poddar, C. Lan, R. Popa, and S. Ratnasamy, "Safebricks: Shielding Network Functions in the Cloud," in *USENIX NSDI'18*, 2018.

[59] S. Sasy, S. Gorbunov, and C. Fletcher, "ZeroTrace: Oblivious Memory Primitives from Intel SGX," in *NDSS*, 2018.

[60] M. Schwarz, S. Weiser, D. Gruss, C. Maurice, and S. Mangard, "Malware Guard Extension: Using SGX to Conceal Cache Attacks," in *DIMVA'17.*, 2017.

[61] M.-W. Shih, S. Lee, T. Kim, and M. Peinado, "T-SGX: Eradicating Controlled-Channel Attacks Against Enclave Programs," in *NDSS*, 2017.

[62] E. Siron, "What is the Hyper-V Virtual Switch and How Does it Work?" https://www.altaro.com/hyper-v/the-hyper-v-virtual-switch-explained-part-1/ [online], 2020.

[63] E. Stefanov *et al.*, "Path ORAM: An Extremely Simple Oblivious RAM Protocol," in *ACM CCS'13*, 2013.

[64] E. Stefanov, E. Shi, and D. Song, "Towards Practical Oblivious RAM," in *NDSS'12*, 2012.

[65] M. Taassori, A. Shafiee, and R. Balasubramonian, "VAULT: Reducing Paging Overheads in SGX with Efficient Integrity Verification Structures," in *ACM ASPLOS'18*, 2018.

[66] "Nokia Makes its Network Functions Available on AWS," https://telecoms.com/500959/nokia-makes-its-network-functions-available-on-aws/ [online], Telecoms, 2019.

[67] B. Trach *et al.*, "Shieldbox: Secure Middleboxes using Shielded Execution," in *SOSR'18*, 2018.

[68] B. E. Ujcich *et al.*, "Cross-App Poisoning in Software-Defined Networking," in *ACM CCS'18*, 2018.

[69] W. Wang *et al.*, "Leaky Cauldron on the Dark Land: Understanding Memory Side-Channel Hazards in SGX," in *ACM CCS'17*, 2017.

[70] X. Wang, H. Chan, and E. Shi, "Circuit ORAM: On Tightness of the Goldreich-Ostrovsky Lower Bound," in *ACM CCS'15*, 2015.

[71] X. Wang, A. J. Malozemoff, and J. Katz, "EMP-toolkit: Efficient MultiParty Computation Toolkit," https://github.com/emp-toolkit [online], 2016.

[72] O. Weisse, V. Bertacco, and T. Austin, "Regaining Lost Cycles with HotCalls: A Fast Interface for SGX Secure Enclaves," in *ACM ISCA'17*, 2017.

[73] K.-Y. Whang, B. Vander-Zanden, and H. Taylor, "A Linear-Time Probabilistic Counting Algorithm for Database Applications," *ACM Transactions on Database Systems*, vol. 15, no. 2, pp. 208–229, 1990.

[74] F. Xiao *et al.*, "Unexpected Data Dependency Creation and Chaining: A New Attack to SDN," in *IEEE S&P'20*, 2020.

[75] L. Xu, J. Huang, S. Hong, J. Zhang, and G. Gu, "Attacking the Brain: Races in the SDN Control Plane," in *USENIX Security'17*, 2017.

[76] Y. Xu, W. Cui, and M. Peinado, "Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems," in *IEEE S&P'15*, 2015.

[77] T. Yang *et al.*, "Elastic Sketch: Adaptive and Fast Network-Wide Measurements," in *ACM SIGCOMM'18*, 2018.

[78] Y. Zhang, A. Steele, and M. Blanton, "PICCO: A General-Purpose Compiler for Private Distributed Computation," in *ACM CCS'13*, 2013.

Algorithm 5 Linear Counting

Input: The distribution array $\{n\} = (n_0, n_1, ..., n_{255})$ of OCMSketch; The OCMSketch parameter w

Output: An estimated cardinality

LINEARCOUNTING($\{n\}$, w)

1: Initialise $sum \leftarrow 0$
2: **for** i = 0 : 255 **do**
3: $card{+}= n_i$
4: **End for**
5: $ratio \leftarrow (w - card)/w$
6: **Return** $-w \times \ln(ratio)$

APPENDIX A
OBLIVIOUS ALGORITHMS FOR OCMSKETCH AND OBUCKET

Linear Counting of OCMSketch. Algorithm 5 is the native linear counting algorithm in [73]. It leverages a hash function to put all elements into an array and then refers to the number of unused counters in the array to estimate the size (cardinality) of the dataset. This algorithm sums all values in the distribution array to learn how many counters are accessed in the current epoch and deduces the unused counters (line 2 - 4). This information allows the enclave to compute the maximum likelihood estimation of the cardinality in OCMSketch (line 5 - 6). Linear counting is data-oblivious because it always scans the entire distribution array and does the same computation to get the cardinality.

Linear-Scan of OCMSketch and OBucket Algorithm 6 outlines how OblivSketch scans OBucket and OCMSketch to get the flow distribution, respectively. In each part, the algorithm aims to scan the entire data structure and return a distribution array. Particularly, OCMSKETCHSCAN solely scans the blocks in Path ORAM to get the distribution array. On the other hand, OBUCKETSCAN aims to get the entire flow distribution of the network. Therefore, it takes as input the distribution array from OCMSKETCHSCAN and scans the buckets in OBucket. For each flow ID in OBucket, it queries OCMSketch to get the actual size of the flow (line 3 - 4 in OBUCKETSCAN). It then updates the distribution array coordinately (line 5 - 7 in OBUCKETSCAN). To ensure its obliviousness, we design RESIZE (line 5 in OBUCKETSCAN). This function leverages an oblivious map to scale up the array when needed. Hence, the memory access on $\{n\}$ is randomised even if OblivSketch accesses the same position in $\{n\}$ since the array is moved to a new place.

APPENDIX B
SECURITY OF OBLIVIOUS SKETCH

In this section, we formalise the security model of the oblivious bucket and provide a proof sketch.

The security of the oblivious sketch is defined under the real/ideal paradigm: In the real world, the adversary interacts with a real oblivious sketch $\mathcal{S}_{Real}$, while in the ideal world, the adversary interacts with a simulator $\mathcal{S}_{Sim}$ of the oblivious

Algorithm 6 Linear Scan

Input: The Path ORAM tree T; the ORAM parameter Z and N
Output: A distribution array $\{n\} = (n_0, n_1, ..., n_{255})$
OCMSKETCHSCAN(T, Z, N)

1: Initialise $\{n\}$ with 0 in all position
2: **for** i = 1 : $Z \times N$ **do**
3: $j \leftarrow T[i].value$
4: $n_j += oselector(0, 1, T[i].bid = -1)$
5: **End for**
6: **Return** $\{n\}$

Input: The distribution array $\{n\} = (n_0, n_1, ..., n_{255})$; the Path ORAM tree of OBucket T_H; the ORAM parameter Z and N of OBucket; The OBucket parameter L; The OCMSketch hash function set $\{H_i\}_{i=1}^d$; The OCMSketch data structure $(T_L, \mathsf{position}_L, S_L)$
Output: An updated distribution array $\{n\}$
OBUCKETSCAN($\{n\}$, T_H, Z, N, L, $\{H_i\}_{i=1}^d$, $(T_L, \mathsf{position}_L, S_L)$)

1: **for** i = 1 : $Z \times N$ **do**
2: **for** j = 1 : $L - 1$ **do**
3: $v_{cur} \leftarrow$ OCMSKETCH.QUERY($T_H[i][j].key, \{H_i\}_{i=1}^d, (T_L, \mathsf{position}_L, S_L)$)
4: $v_{new} \leftarrow T_H[i][j].value + oselector(v_{cur}, 0, T[i][j].tag)$
5: RESIZE($\{n\}$, $oselector(v_{new}, \{n\}.size(), v_{new} > \{n\}.size())$)
6: $n_{v_{cur}} -= 1$
7: $n_{v_{new}} += 1$
8: **End for**
9: **End for**
10: **Return** $\{n\}$

sketch. For both experiments, the adversary can initialise the oblivious sketch and supply any number of flow insertions and queries. Then, the adversary can observe the memory access on server including the that protected by enclaves. The following theorem states the security of the oblivious sketch:

Theorem 1. *The oblivious sketch is data-oblivious, assuming that the Path ORAM protocol is data-oblivious.*

Proof: In this proof, we construct a simulator that can simulate the oblivious sketch in the view of adversaries on the server. For an adaptively selected flow tuple (ID, v), the simulator uses the simulator of Path ORAM $\mathcal{S}_{ORAM}$ to select an entry in OBucket and scans all the buckets in that entry. Then, it updates the eviction count and obliviously swaps the input flow and minimal flow in the bucket. As a result, the insertion on OBucket is oblivious since the access pattern for each insertion is exactly the same: the adversary can only see a random entry is scanned, then the entry's eviction counter is updated. OBucket outputs a flow at the end of the insertion process, and this flow will be inserted into OCMSketch. We can invoke $\mathcal{S}_{ORAM}$ to simulate this process.

For the query process, the simulator computes the random oracle of flow ID and scans the corresponding entry retrieved by $\mathcal{S}_{ORAM}$. It will return a result no matter whether the flow ID is in the entry or not (the size is 0 if it does not exist). Then the simulator leverages the flow ID to query OCMSketch, which can be simulated by the simulator of Path ORAM. Finally, the query process gets two results (one from OBucket and another from OCMSketch) and accumulates them obliviously as the final output. Thus, the query process is also oblivious since the adversary can only see the random accesses on entries and paths in OBucket and OCMSketch, respectively.

We can conclude that the oblivious sketch is data-oblivious for its insertion and query process, i.e., each access has the same pattern independent of the given flow ID and size. Hence, an adversary cannot infer the sensitive network statistics via memory access side-channels on Intel SGX. ■

APPENDIX C
SECURITY OF MEASUREMENT TASKS

We analyse the security of each task by providing the simulator of each task as follows:

- **Flow size estimation**: The security of this task is guaranteed by Theorem 1 because it is based on the query process of the oblivious sketch.
- **Heavy-hitter detection**: To simulate the heavy-hitter detection, the simulator scans the whole OBucket and loads the content of each bucket into a map, then it invokes the simulator of bitonic sorting to sort the map and returns the top-k flows as results.
- **Heavy-change detection**: To simulate the heavy-change detection, the simulator scans the OBucket of current epoch and previous epoch and loads the content of each bucket into a map, then it leverages a nested-loop to compare the flows in the above two maps. All the flows that have more than T changes will be returned as results.
- **Cardinality estimation**: The simulator of the cardinality estimation first scans OBucket to count the number of flows in it. Then, it scans the all Path ORAM blocks of OCMSketch and adds the value of each block to a distribution array (c.f. OCMSKETCHSCAN in Algorithm 6). The distribution vector is then further scanned via the linear counting (c.f. Algorithm 5) to estimate the number of flows in OCMSketch. The simulator returns the sum of the above value as result.
- **Flow distribution estimation**: To simulate the flow distribution estimation, the simulator scans the all Path ORAM blocks of OCMSketch and adds the value of each block to a distribution array (cf. OCMSKETCHSCAN in Algorithm 6). Then, the simulator scans OBucket to update distribution array (cf. OBUCKETSCAN in Algorithm 6). In particular, OBUCKETSCAN leverages each flow ID in OBucket to query in OCMSketch and get the actual flow size of a heavy flow. Then, it updates the distribution array according to the new flow size.
- **Flow entropy estimation**: The simulator of the flow entropy estimation is based on the one for flow distribution estimation. After getting the final distribution array, the simulator can scan the whole array and compute the entropy.

As the above simulators are either been simulated by the simulator of oblivious primitives or based on linear-scan, they are data-oblivious in the presence of an adversary who can observe the memory space of a untrusted server.