electronics

MDPI

Article

FO-Sketch: A Fast Oblivious Sketch for Secure Network Measurement Service in the Cloud

Lingtong Liu [1], Yulong Shen [1,*], Shuiguang Zeng [1,2] and Zhiwei Zhang [1]

[1] Shaanxi Key Laboratory of Network and System Security, School of Computer Science and Technology, Xidian University, Xi'an 710071, China; xviviliu@gmail.com (L.L.); zengshuiguang@gmail.com (S.Z.); zwzhang@xidian.edu.cn (Z.Z.)
[2] Key Laboratory of Network and Cyber Security in Hebei Province, College of Computer and Cyber Security, Hebei Normal University, Shijiazhuang 050000, China
* Correspondence: ylshen@mail.xidian.edu.cn

Abstract: Network measurements are the foundation for network applications. The metrics generated by those measurements help applications improve their performance of the monitored network and harden their security. As severe network attacks using leaked information from a public cloud exist, it raises privacy and security concerns if directly deployed in network measurement services in a third-party public cloud infrastructure. Recent studies, most notably OblivSketch, demonstrated the feasibility of alleviating those concerns by using trusted hardware and Oblivious RAM (ORAM). As their performance is not good enough, and there are certain limitations, they are not suitable for broad deployment. In this paper, we propose FO-Sketch, a more efficient and general network measurement service that meets the most stringent security requirements, especially for a large-scale network with heavy traffic volume and burst traffic. Let a mergeable sketch update the local flow statistics in each local switch; FO-Sketch merges (in an Intel SGX-created enclave) these sketches obliviously to form a global "one big sketch" in the cloud. With the help of Oblivious Shuffle, Divide and Conquer, and SIMD speedup, we optimize all of the critical routines in our FO-Sketch to make it 17.3x faster than a trivial oblivious solution. While keeping the same level of accuracy and packet processing throughput as non-oblivious Elastic Sketch, our FO-Sketch needs only ∼4.5 MB enclave memory space in total to record metrics and for PORAM to store the global sketch in the cloud. Extensive experiments demonstrate that, for the recommended setting, it takes only ∼0.6 s in total to rebuild those data during each measurement interval.

Keywords: sketch; secure network measurement; network function virtualisation; software-defined network; intel SGX; path ORAM

Citation: Liu, L.; Shen, Y.; Zeng, S.; Zhang, Z. FO-Sketch: A Fast Oblivious Sketch for Secure Network Measurement Service in the Cloud. *Electronics* **2021**, *10*, 2020. https://doi.org/10.3390/electronics10162020

Academic Editor: Priyadarsi Nanda

Received: 22 July 2021
Accepted: 19 August 2021
Published: 20 August 2021

1. Introduction

Network measurements are the foundation for network management applications, such as analyzing flow-level statistics [1,2] for traffic engineering [3,4] and Quality of Service (QoS) [5,6], detecting heavy hitters for load balancing [7,8], tracking heavy changes [9,10] for traffic anomalies, estimating flow size distribution [11] for cache admission/eviction [12], and counting distinct flows [13] for identifying DoS attacks and port scans [14]. The metrics generated by those measurements help improve the performance of the monitored network and harden its security. By separating the data plane and the control plane, Software-Defined Networking (SDN) [1,15,16] makes network switches and routers in the data plane only forward packets and perform simple traffic statistics, leaving the network's control logic and measurements implemented in a logically centralized control plane, which simplifies network management and measurement. Furthermore, by decoupling those functions from hardware appliances on which they run, Network Function Virtualisation (NFV) [17,18] has the potential to boost agility and time-to-value while significantly reducing costs. With the rapid research progress and technological evolution of SDN and NFV, outsourcing of network measurement services to the cloud has become very flexible and efficient.

Despite the enormous potential of SDN and NFV, directly deploying a network measurement service in third-party public cloud infrastructures [19,20] incurs privacy and security issues [17,21–24]. To provide different application-level metrics of interest, the measurement service hosts accurate and timely statistics on the global flows. Considering an enterprise network, those metrics should be proprietary and confidential to the enterprise [25]. We must keep it a secret to prevent some curious or malicious cloud infrastructure or co-tenants from stealing it. To make matters worse, recent studies have shown that the abuse of flow statistics helps poison network visibility [26], carry out DDoS attacks [27], or manipulate network topology [28]. The feasibility of severe network attacks using leaked information makes guarding network measurement service in the cloud a must.

1.1. Available Solutions and Limitations

To provide network measurement service in a secure way in a third-party public cloud infrastructure, we can harness two completely different kinds of methods. One method is to design a set of composable cryptographically protected secure multi-party computation (SMC) [29,30] protocols that implement the measurement tasks we need. Note that those measurement tasks are executed over encrypted flow statistics. If the analyzed data are small or if we evaluate a simple metric by only scanning each item, SMC protocols can be designed using garbled circuits [25] and secret sharing [31,32]. When the data set becomes more extensive and the measurement becomes more complicated, data-dependent control transfer and data access both leak memory access patterns [33]. With prior knowledge, sophisticated inference attacks [33,34] can be carried out using those memory access side-channels. In the network measurement scenario, the whole underlying network is monitored persistently and the flow statistics are gathered periodically.

Imagine that a powerful attacker has hacked the cloud infrastructure and the memory access pattern is now exposed. Assume some flow A was queried in a former epoch by the attacker. As usually flow information is organized in a hash table [35–37], the attacker can associate the memory address accessed with flow A. In the next epoch, if some network application issues a same request, the same memory address is accessed and this event is detected immediately by this powerful attacker. The attacker then can conclude that this request is a flow size query for A. Using a memory access side-channel, more information can be obtained by the powerful attacker. Thanks to Path Oblivious RAM (PORAM) [38,39], SMC can protect computations on big data [25,40,41]. Without hardware assumption, SMC can provide a provable security guarantee. However, according to [25], extracting heavy hitters from only four thousand flows will take over 10 min. The periodically updated network measurement service's high-performance requirements make SMC unable to be used as a practical solution.

The other method is to confine the private flow statistics data and measurements that operate on the data in an integrity- and confidentiality-guaranteed trusted execution environment (TEE) created by a secure processor [42,43], leaving only data import and request interfaces in the potentially malicious public cloud infrastructure and co-tenants. As the data and computation are in plain form, the biggest advantage of using a secure processor is its rich functionality and high performance. Assisted by secure processors, a broad spectrum of secure network functions [44–47]—load balancer, firewall, and IDS, to name a few—have recently been implemented. It looks promising to deploy a network measurement service in a secure processor-enabled public cloud infrastructure [48–50]. However, the aforementioned access pattern leakage still exists in the hardware-assisted solution, threatening the deployment of secure network measurement service. ORAM storage for flow statistics and dedicated oblivious primitives for specific measurement logic together can make access patterns of measurement tasks oblivious [51]. Though OblivSketch [51] handles a typical link (40 Kpps, packets per second) with the same characteristic as a backbone link [52], a 10 Gbps line rate with maximum 14.88 Mpps [53] makes the measurement service hard to work with, not to mention monitoring a large-scale network with heavy traffic volume or burst traffic, where more than one switch is deployed.

1.2. Motivation and Challenges

Our motivation question is simple: *Can we develop a performance-secure measurement service in a public cloud that can monitor a large underlying network with more than one switch and that does not sacrifice accuracy and functionality?*

Two challenges are confronted in addressing this question.

Challenge I: How can we design a compact but still accurate mergeable flow statistic summary for network-wide high volume traffic? Using conventional flow summaries [54], to provide the same accuracy guarantee, we should approximately double the size of the statistic summary when the volume of packets monitored doubles. Though Elastic Sketch and SF-Sketch [37,55] can create extremely compact summaries with the same accuracy as conventional approaches [54], it is not easy to merge [35,56] network-wide flow summaries from different switches. On the other hand, to make data access oblivious, Path ORAM [39] blows up the memory footprint of the protected statistics Z ($Z \geqslant 4$ is required to make the failure probability negligible in the size of the protected data) times larger.

Challenge II: How can we design efficient oblivious operations for fast flow statistic sketch generation and network metric evaluation? In order to keep up with the measurement period, all of the oblivious merging and measurements should be finished within a time shorter than the interval, leaving room for queries from network management applications. Simply using SGX and Path ORAM technology cannot meet the above requirements. More sophisticated optimizations should be devised.

1.3. Contributions

In this paper, we propose FO-Sketch, a fast oblivious network measurement service using mergeable sketch and Intel SGX. FO-Sketch can monitor high volume traffics in a large-scale network with multiple switches and can protect private global flow statistics and measurements against active, software-based attacks.It provides fast oblivious service for network applications by efficiently rebuilding all the supported metrics (or metric candidates) within 0.6 s and as ORAM stores the flow statistic sketch once all information is received from local switches at the end of measurement epoch.

Figure 1 describes the components of our FO-Sketch. For a large-scale network spanning multiple regions, such as an enterprise network, we deploy one *merge server* for switches in the same region to merge encrypted flow statistic data structure sketches from those *local switches* periodically. As illustrated in Figure 1, FO-Sketch in the cloud infrastructure is composed of three components: *Statistics*, *Oblivious Measurement*, and *Metrics*. Note that FO-Sketch is designed to work in a secure container (called an enclave) with integrity and confidentiality guarantees. All of the operations or functions in the FO-Sketch must be implemented data-obliviously, leaking no access pattern. *Statistics* periodically collects merged sketches from merge servers and merges all of them obliviously into a global sketch, which summarizes all of the flow statistics in the monitored network. *Oblivious Measurement* is a library providing oblivious functions that analyze the global sketch to produce supported metrics. To provide a general network measurement service, our FO-Sketch supports five representative measurements, i.e., flow size estimation, heavy-hitter detection, heavy-change detection, cardinality estimation, and flow distribution estimation. To save on computation cost, in each measurement epoch, we only analyze each of the four metrics except flow size estimation once and store the calculated candidate result into *Metrics*. *Metrics* then responds to metric requests of the four types by returning the stored result or by merely filtering the required data from the candidate result.

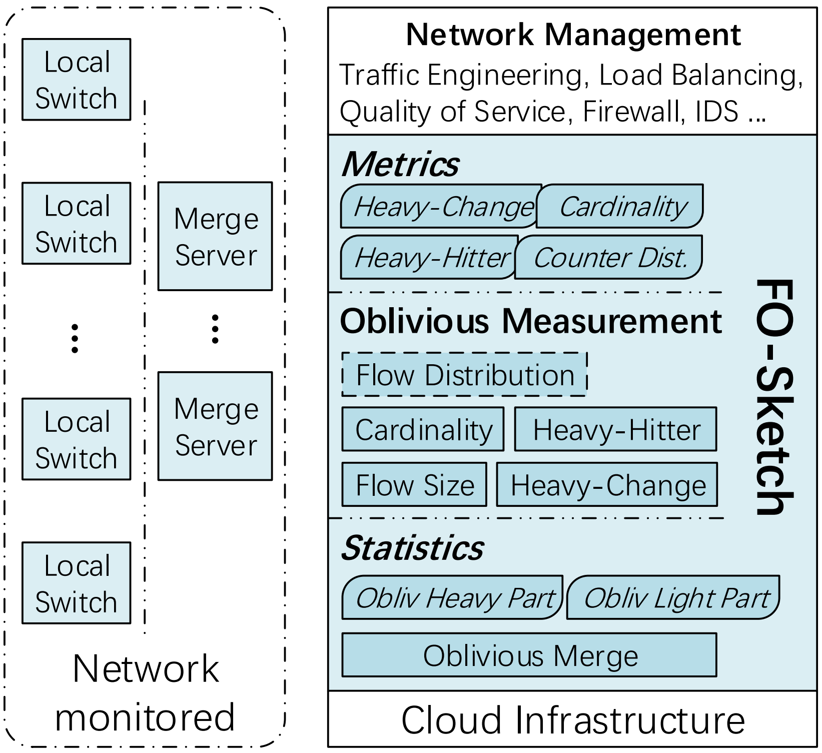

Figure 1. FO-Sketch works for a large-scale network and provides an oblivious network measurement service in a public cloud infrastructure via mergeable sketch and SGX-enabled oblivious functions. Trusted components (software and regions of memory) are shaded.

1.3.1. Design Choices

A small sample [57,58] of the packets can only estimate the average metric of a network, whereas a sketch can find it exactly [54]. As explained above, the smaller the memory footprint, the much faster the oblivious measurement. While compact sketches [37] can provide accurate metrics estimation for the five representative measurements, we chose Elastic Sketch as our underlying data structure for recording flow statistics. In a nutshell, Elastic Sketch comprises two parts: the heavy part used for recording candidate heavy flows in a sampling way and the light part for updating remaining flows in a sketch way. Although OblivSketch [51] used Elastic Sketch as the underlying sketch, it assumes 70k distinct flows in a 5 s period and transfers "(flowID, occurrence)" to the measurement server to update an PORAM-stored Elastic Sketch. It is not easy to calculate a flow's occurrence (size) in a fast packet arriving rate. At the same time, updating a PORAM-form Elastic Sketch in an oblivious way is still complex and time-consuming. Unlike OblivSketch, to support an even faster packet arrival rate, we chose not to update any sketch in the measurement server and to put the local flow statistic sketch's updating in the local switch. This choice can ease the work of the measurement server. Instead of sketch updates, only merging local sketches are required in the measurement server to obtain a global flow statistic sketch. The sketch for merging is a tinny variant of Elastic Sketch; details can be found in Section 5.1. As a secure processor-enabled public cloud is available and much faster than the SMC solution, we chose to store sketches using PORAM and to design an efficient oblivious sketch merger and measurement metrics estimation functions running inside a TEE known as an *enclave* created by Intel SGX [59].

1.3.2. Proposed Approaches

- In this paper, we identified five critical functions that can be optimized in implementing oblivious sketch merging and metrics estimation; they are the merging of the light parts from two different underlying sketches (*Merge LxL*, for short), the merging of

the light part with many heavy flows (*Merge LxF*), the calculation of heavy-change candidates (*Heavy-change*), the calculation of counter distribution from the light part (*Counter dist.*), and the calculation of flow distribution from the heavy part (*Flow dist.*). With the help of Oblivious Shuffle, Divide and Conquer, and SIMD instruction set AVX2 [60,61], we optimized all five critical functions obliviously.
- To provide an even faster flow size estimation, we optimized the process of PORAM access in an oblivious way.
- To prove the data-obliviousness of our proposed optimized functions, we proposed a simple and effective data-oblivious proof method to verify its correctness.
- We also provided a switchless request and response protocol.
- The extensive experiments show that our FO-Sketch can provide an oblivious network measurement service in an SGX-enabled public cloud for a large-scale network.

1.3.3. Paper Structure

We begin with related work in the next section. We then present the background in Section 3, the system overview that outlines how the service system works in Section 4. After providing detailed optimization for critical oblivious functions in Section 5, we realize the oblivious network measurement service in enclave in Section 6. After the evaluation (Section 7), we summarize FO-Sketch and discuss future directions in Section 8.

2. Related Work

2.1. Non-Oblivious Network Measurement Service

There have been considerable efforts [35–37,53,55,56,62,63] in making a reasonable trade-off among goals: achieving a higher line-rate (such as, 40 Gbps) for 64B packets (59.2 Mpps), providing better accuracy guarantees (than CM sketch), saving more memory space, or supporting more general measurement tasks. Thanks to their higher accuracy than sampling methods [57,58], most of them use sketches [54] as their primary data structure to record flow statistics. These systems work without considering security issues.

2.2. Hardware-Assisted Isolated Execution Environments for Securing Network Middleboxes

By creating isolated execution environments, secure processors [43,64] are widely adopted to build effective and efficient defensive tools for securing network middleboxes [44–47]. ARM TrustZone [65] creates two environments: a trust execution environment (TEE) hosting a secure container and a rich execution environment (REE) running an untrusted software stack by partitioning a system's resources. As TrustZone's TEE relies on a secure OS, the trusted code base (TCB) is more prone to having security vulnerabilities than a smaller one. Contrary to ARM TrustZone, Intel SGX [42,59] does not trust any privileged layer (firmware, hypervisor, OS) and its TCB only consists of the processor's microcode and a few privileged containers deployed by Intel. Due to its ease of deployment and small TCB, Intel SGX has recently been chosen to protect a broad spectrum of network middleboxes [44–47].

2.3. Access Pattern Attacks and Oblivious Solutions for SGX

Despite the fact that using SGX can resist ordinary attacks, more sophisticated forms of attacks still exist, including memory mapping attacks [33,66], cache timing attacks [67], and software side-channel attacks [68]. Mitigation measures [69,70] for SGX can only solve specific attacks, and new forms of attacks [71] continue to be discovered. One broad category of those attacks succeeds by exploring access patterns of target software. To avoid access pattern leakage, recent studies design oblivious systems [72–74] by storing critical data in an oblivious data structure and by devising data-oblivious algorithms. The state-of-art solution OblivSketch [51] for implementing an oblivious network measurement server uses an ORAM controller in enclave to obliviously update and access an Elastic Sketch. It receives $(flowid, size)$ sets from local switches and updates an ORAM-stored

Elastic Sketch obliviously from those pairs. The paper [51] also states that measurements in OblivSketch are oblivious.

3. Background

3.1. Sketches and Network Measurements

Like other network measurement systems [35,37,53,72], we mainly gather flow statistics and analyze typical metrics on them. Each flow is identified by a unique flow ID, being any combinations of the 5-tuple: a source IP address/port number, destination IP address/port number, and the protocol in use. The recommended sketch for the light part of Elastic Sketch is Count Min (CM) Sketch [54]. Though the widely adopted Count Min (CM) Sketch offers a flow statistic summary with theoretically guaranteed accuracy for estimating flow size, Conservative Update (CU) sketch [75] outperforms CM-Sketch with better accuracy using a more conservative increment update method. When applied for the network packet stream, the shortcoming of not supporting decrement updates can be ignored as packets' arrival only incurs increment update. As shown in Figure 2, CU-Sketch is composed of d arrays, each of w counters. When a packet arrives, its flow ID is extracted and then is added into the sketch. Upon updating, CU-Sketch exams the counters, each indexed by an independent hash function of the flow ID in each array, and increases the smallest ones to minimize overestimation. To answer a flow size query, CU-Sketch returns the minimum of the d counters.

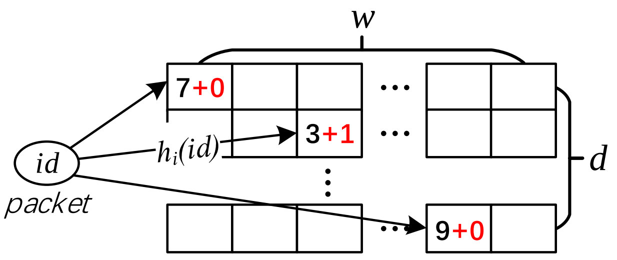

Figure 2. Example of a conservative update (CU) Sketch.

The following five kinds of well-studied flow-based network metrics are provided in our measurement server:

- Flow size: estimates the size of any flow by its flow ID;
- Heavy-hitters: detects flows whose sizes are larger than a user defined threshold;
- Heavy-changes: detects flows that have experienced a drastic change (judged by a user-defined threshold) in two adjacent monitoring windows;
- Cardinality: estimates the number of global flows;
- Flow distribution: estimates the distribution of global flow sizes.

3.2. Intel SGX

The Intel SGX-enabled processor [42,59] creates an *enclave* to host codes and data that require integrity and confidentiality protection. SGX isolates each enclave's environment from the untrusted software stack so that the privileged level cannot access the enclave memory or tamper with its content. As an enclave running in a non-privileged level, it also cannot access any privileged software. By providing local and remote software attestation, a client, local or remote, can be convinced that the specific code was running inside an enclave and can exchange secrets with the server's enclave part via the established secure channel.

3.3. Path ORAM and Oblivious Controller

Path ORAM (PORAM) [39], an ORAM protocol, stores user's data in a server in a specific way that the accessing of the data cannot leak access patterns. The PORAM protocol considers two parties: a server storing user's data remotely and a controller providing read/write interfaces to the user, where the user runs the controller locally. Specifically, in order to store a private data, it should be partitioned into fixed size data blocks $data_{id}$ with block id bid, (assume there has N data blocks). The server-side storage is organized as a perfect binary tree T of 2^L leaf nodes having the same depth of L, in which $L = \lfloor log_2 N \rfloor$. Each node in the tree is called a bucket that contains Z storage blocks. These blocks are either real blocks that hold user data or dummy blocks containing nothing. The controller consists of two data structures, a stash S for caching blocks (be newly generated blocks or old ones read from the server) that have not yet been written and a position map PM that randomly assigns a leaf position pos in the tree T to each user block $data_{id}$ with a block ID bid. For a block–leaf pair (bid, pos), PORAM ensures that a data block $data_{id}$ is stored in either some bucket on the path from the tree T's root to leaf pos or the stash S. Each storage block of type $(bid, pos, data_{id})$ contains the payload $data_{id}$ and stores to bid its block number and to pos an initialized random position in the leaf part of the T. The two values are used to check and verify the block's identity when accessing a block.

Below, we summarize how the protocol works:

- *(T; (S, PM))* $\leftarrow$ ***PORAM.Init****(; (N, Z, B))*. On input a block budget N, node size Z blocks, and block size B in the controller, *PORAM.Init* initializes the server S with $Z \times N$ encrypted blocks in a perfect tree form, in which each node contains Z blocks and output a initial state (S, PM) to the controller.
- *(T′; (data, pos, S′, PM′))* $\leftarrow$ ***PORAM.ReadBlock****(T; (bid, S, PM))*. Upon inputting a Tree T in the server and a block ID bid, a stash S and a position map PM in the controller, $PORAM.ReadBlock$ fetches all blocks on the path from the T's root to leaf $l = PM[bid]$, inserts them into S, and outputs the fetched block's $data$ to the controller. Furthermore, it assigns bid a new random leaf in PM'.
- *(T′; (S′, pos))* $\leftarrow$ ***PORAM.Evict****(T; (S, pos))*. Note that the controller can change the contents of blocks in S as needed. To write back blocks in S, *ORAM.Evict* push them as far down the path from root to pos as possible, while keeping with the main invariant that each bucket in the path contains at most Z blocks. The path is then evicted to the tree T.

As accessing enclave's secure memory still leaks access trace, ZeroTrace [73] implements an oblivious ORAM controller in an enclave leaking no access pattern. We mainly use the non-recursive version of the ORAM controller as the private data of global flow statistics is no more then 1 MB. Below we outline the critical protocols:

- *((T, E_{OC}); ϕ)* $\leftarrow$ ***ORAM.Init****(; (N, Z, B))*. First, the untrusted server loads OC into an enclave E_{OC}, where OC is the compiled enclave program for oblivious controller and E_{OC} is the process running in the enclave. Then, E_{OC} initiates $(T; (S, PM))$ with the server by executing *OPORAM.Init(; (N, Z, B))* (the Oblivious Controller). Third, with remote attestation, the user obtain a proof ϕ produced by E_{OC} from the server and verifies the proof ϕ with a trust third party to ensure that the controller program OC is honestly initialized by the server and exchanges a secret key K with E_{OC} secretly for further data transfer.
- *(T′, (S′, PM′), $data_1$, pos)* $\leftarrow$ ***ORAM.Access****(T; (S, PM); (op, bid, $data_0$))*. The serve relays the user's access request of read/write op the $data$ of block bid to E_{OC}. E_{OC} runs *OPORAM.ReadBlock(T; (bid, S, PM))* obliviously with the server to obtain *(T′; ($data_1$, pos, S′, PM′))* and runs *OPORAM.WriteToStash* to write block *(bid, pos, $data_0$* back into S' if $op \equiv write$. Then, E_{OC} responds to the access request by returning an encrypted $data_1$ through the server to the user. Finally, they execute *(T′; (S′, pos))* $\leftarrow$ *OPORAM.Evict(T; (S, pos))* obliviously.

4. System Overview

4.1. Architecture

We mainly consider a large-scale network outsourcing network measurement service to the public cloud. For a large underlying network spanning across multiple regions, as shown in Figure 3, we deploy one merge server for switches in the same region to merge encrypted sketches (flow statistics) from those switches. A local switch can host the merge service if a software switch [76,77] implements it. Thus, the local switches and merge servers are organized in a forest shape.

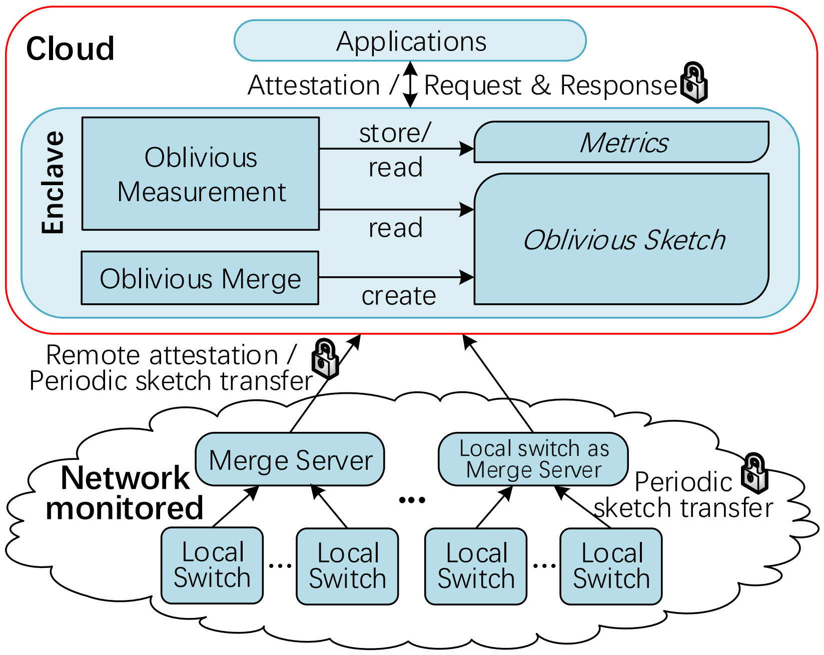

Figure 3. System overview of FO-Sketch. Trusted components are shaded.

Below, we outline how the service system works. Before providing flow statistics from its region, the merge server executes remote attestation with the measurement server to ensure that the measurement service is honestly initialized by the public cloud and exchanges secret keys to create a secure channel for transferring sketches. It also established a secure channel with related local switches. The local switch monitors local traffic and updates the local sketches with only selected packets (any ingress packets destined to the local area and only egress packets destined to the external network; details can be found in Section 6.4). At the end of each epoch, the sketch is transferred in ciphertext to an assigned merge server and then emptied to update the next epoch's packets in the local area. In the merge server, received sketches are merged and then transfered in ciphertext to the network measurement server. In the measurement server, the merge routine obliviously merges those sketches from different regions and then creates a global oblivious sketch for further analysis. Except for flow size, other network metrics can be calculated in advance. Thus, once the global sketch is created, the measurement module executes measurement tasks except for flow size estimation and then stores those metrics for following requests from applications. When a network application needs some network metrics, it first executes remote attestation with the measurement server to ensure its authenticity and exchanges secret keys for securing following requests and responses. After that, network applications can request desired network metrics from the measurement server. Upon receiving the encrypted request from an application, depending on the type of the request, the measurement module linearly scans the stored metrics or obliviously queries the global sketch by flow ID. Note that the global oblivious sketch and metrics are rebuilt periodically.

4.2. Threat Model

We describe security threats from the three different components.

4.2.1. Cloud Infrastructure

In our setting, network measurement logic (e.g., the five measurement tasks), merge routine, global flow statistics sketch, service, and monitor interfaces are implemented in a trusted hardware enclave to protect against a powerful attacker who can fully control the privileged software stack of a public cloud infrastructure. We assume that our attacker may control any component, all hardware except the processor chip, and all software except a few privileged enclaves deployed by Intel on the cloud. Thus, the attacker has unlimited access to untrusted memory, the memory bus, and any kind of communications (network level, process level, and bus level). Moreover, it can observe the access traces of data and code protected in the enclave memory at cache-line granularity. The network measurement logic is assumed to be implemented correctly.

4.2.2. Network Applications

In FO-Sketch, an application may misbehave: releasing its own requests or responses. However, it cannot affect the privacy of other clients.

4.2.3. Monitored Network

We assume that the (software) switches and merge servers are trusted since network operators can configure their switches and can merge servers with a strict policy [77]. If any merge server is deployed in an edge cloud, it must be treated as unsecure, and an oblivious merge service is needed (see Section 6.1).

4.2.4. Security Goals and No-Goals

FO-Sketch's highest supported security level is for the network measurements and merge routines running in an SGX enclave to be executed obliviously against any active, software-based attacks. FO-Sketch also supports the indistinguishability of request types and hides the size of responding data optionally (see Section 6.4). We do not defend against hardware attacks against the enclave. Specifically, speculative execution [78], power analysis [79], denial of service [80], etc. are out of our scope.

5. Optimize Functions and PORAM Controller Obliviously

We first describe the sketch data structure that is suitable for FO-Sketch. Then, we propose two packet selection strategies, analyze their feasibility, and give the merge algorithm without obliviousness. We also introduce several commonly used oblivious primitives, based on which we analyze and optimize the implementation of the five critical functions proposed above. Finally, we point out the deficiencies of the existing oblivious ORAM controller and propose an optimized implementation. The work in this section guarantees the realization of a fast and secure network measurement service system in the next section.

5.1. Sketch Data Structure for Flow Statistics

The sketch used in FO-Sketch exists in three locations: local switch, merge server and network measurement server; As shown in the Figure 4, the local switch has two forms of Sketch, namely *Update Sketch*, which is used to record the flow statistics related to local, and *Merge Sketch*, which is generated from Update Sketch and is transmitted to the merge server for mergers. The network measurement server is used to merge *Merge Sketches* to obtain the final *Global Sketch*. Local Update Sketch is used to update data packets in the local network in each measure epoch; before transmitting them to the merge server, it needs to be converted to a Merge Sketch supporting merge operation. Due to the scalability, high accuracy, and small footprint of Elastic Sketch, we chose Elastic Sketch to implement local Update Sketch, which consists of two parts: a heavy part and a light part. The heavy part is a hash table of four-tuple (flowID, size, flag, negative vote), and the light part can be CU Sketch or CM Sketch according to the update strategy. Merge Sketch's light part is a compressed version of the light part of Updates Sketch. Here, we can treat Update Sketch's light part and Merge Sketch's light part together be a SF Sketch [55]. This design choice takes advantage of Elastic Sketch and SF Sketch simultaneously, which reduces the network overhead and improves the accuracy of flow statistics at the same time. Merge Sketch's heavy part is an ordered list of (flowID, size) pairs. For each four-tuple (flowID, size, flag, and negative vote), if the "flag" is set to 1, it means that the flow size is composed of two portions: one is from the heavy part, and the other is from the light part; thus the flow size of Merge Sketch's heavy part should be added by the flow size estimation from Update Sketch's light part. By reducing the four-tuple to a pair, we have a more straightforward data structure of the light part than Elastic Sketch. This design choice further reduces the size of the light part and facilitates the subsequent optimization of the oblivious merge function.

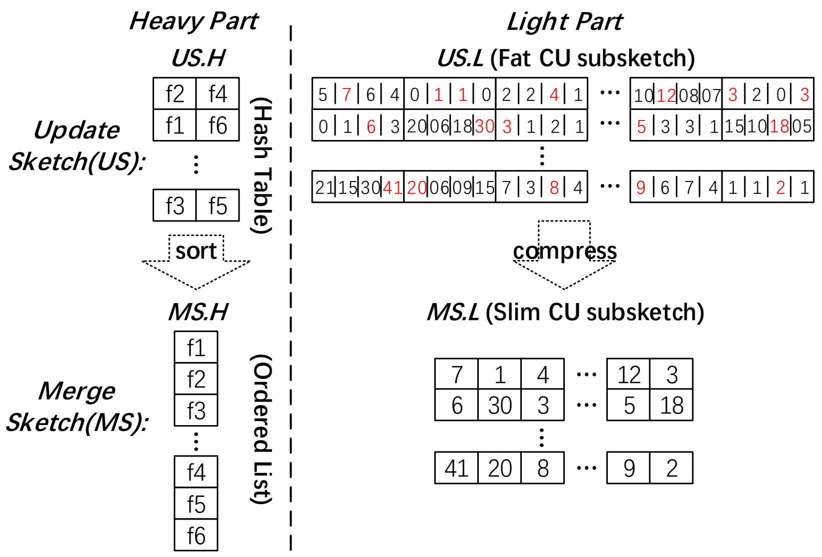

Figure 4. Data structures of Update Sketch and Merge Sketch. Note that each element of the heavy part is an ID–value pair to represent (flowID, size).

5.1.1. The update of the Update Sketch

For each selected data packet (see Section 5.2 for the selection strategy), Update Sketch extracts the flow identifier f_1 and flow size increment δ to update the heavy part first. If the flow size represents the number of data packets, the increment is 1; if the flow size represents the data volume of the flow, the increment is the size of the data packet. The heavy part is mainly used to filter out light flows and to track potentially heavy flow, leaving the light flows to be updated in the light part.

5.1.2. Updating the Heavy Part

Calculate index from hashing f_1, and obtain the four-tuple record (f, s, F, v) in the hash table of the heavy part; then, update the four-tuple accordingly:

- If the record is empty, replace with $(f_1, \delta, 0, 0)$. Here, $F = 0$ indicates f_1 only has the heavy portion in this record;
- If $f_1 = f$, then increase s by δ;
- If $f_1 \neq f$ and $\frac{s}{v+\delta} < \lambda$, then increase the negative vote by δ and update the light part by $(f1, \delta)$;
- If $f_1 \neq f$ and $\frac{s}{v+\delta} \geqslant \lambda$, then reset the record to $(f_1, \delta, 1, 1)$ and update the light part by (f, s). For now, f is deleted from the heavy part. As before $f1$ is chosen as a potentially heavy flow, with high probability, it may be already updated in the light part, we set $F = 1$ to indicate that this flow has a light portion from the light part.

As CU Sketch is more accurate than CM Sketch in practice, we use CU Sketch to represent the light part, and the updating of it can be found in Section 3.1.

5.2. Merge Algorithm and Mergeability

The mergeability of Merge Sketch is a prerequisite for the correctness of the FO-Sketch network measurement service system. We briefly introduce the merge algorithm, which includes merging the heavy part and the light part of Merge Sketches before analyzing the mergeability.

5.2.1. Merge Algorithm

When merging the heavy parts of two sketch, firstly, the ordered lists representing the heavy part are combined into an ordered list arranged in descending order of flow size. The high-order half of the ordered list is the heavy part of the merged sketch, and the low-order half is inserted into the light part of the merged sketch. Before inserting, the light parts of the two sketches need to be merged. Note that, for all of the sketches in the same measure epoch, be it Update Sketches or Merge Sketches, or Global Sketch, they share the same hash functions for the light part. The upper part of Figure 5 shows the max-merge scheme of two light parts. The maximum of the two counters from the same position in the light parts is assigned to the same position counter of the merged light part. The lower part of Figure 5 shows how to merge (flowID, size) pairs from the low-order half into the light part using max-merge. Though the above non-oblivious merge algorithm is easy to implement, it is hard to design an effective oblivious one (we postpone the description of oblivious merge later in Sections 5.4 and 6.1).

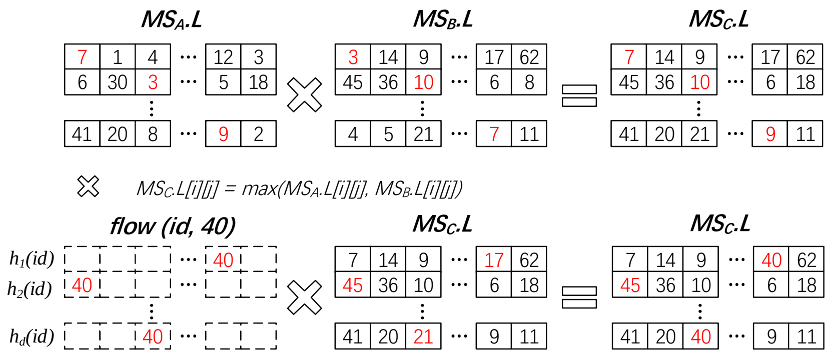

Figure 5. Critical part of the merge algorithm in the merge server.

5.2.2. Mergeability Analysis

Trivially selecting all of the packets to update a underlying sketch in a network monitoring scenario may make sketches not mergeable. Suitable selection criteria makes sketch support one of the two types of merge: sum-merge or max-merge. For sum-merge, we permit the two different flow sets monitored by the two sketches to have the same subset but restrict each packet to only being updated by one of the two sketches. The constraint above makes simply processing any packet passing through the switch not possible, as a packet from source to destination may be relayed by multiple switches. For max-merge, besides the above constraint, we should also restrict that the two switches monitor two disjoint flow sets. To support max-merge, we need to meet the above two constraints at the same time. Thus, suitable selection criteria for choosing specific packages information to record should be designed in each local switch to make the sketch max-mergeable. For a local switch, any packet received by any host of the served local subnetwork is called an ingress packet, while egress packets are those originated from any local host. A packet can be an ingress packet for some switch, while it can also be an egress packet for other switch. Thus recording both ingress packets and egress packets in a local switch may cause the same data packet to be recorded repeatedly. Therefore, we should selectively record some packets in each local switch. Note that, in most conditions, the monitored network may be connected to other networks. If we choose to record only ingress packets, it is apparently not enough, as all if the egress packets destined to the external network cannot be monitored by FO-Sketch. Therefore, in each local switch, the packet selection criteria of FO-Sketch is selecting all ingress packets and only egress packets destined to the external network. By doing so, each packet can be updated only once, and each flow can exist in only one local Update Sketch. The merge algorithm can be implemented by max-merge, which improves the accuracy of flow size estimation.

5.3. Oblivious Primitives

According to [81], "*evaluations that involve registers only are not recorded in the trace, hence, any register-to-register data manipulation is data-oblivious by default*". Similar to [81], we also chose oblivious assignments, comparisons, and array access as our basic primitives, including *OCMove* and *OCMoveEx*. As those primitives are implemented by in-line assembly in C/C++ using instructions from Intel AVX techniques [60] and it is too technical to illustrate, interested readers can find the implementation in "OPrimitive.h" from our source code [82]. If *cond* is true, $OCMove(a, b, cond)$ returns the value of a, else b. *OCMoveEx* is an extended version of *OCMove* to handle any type of variable. As Batcher's sorting network Bitonic sorters [83] achieve $O(n(log^2 n))$ time complexity with a small constant, similar to [81], we implemented our oblivious sort primitive *OSort* by using a Bitonic sorter with an oblivious compare-and-swap function *OCSwap*. Note that, the sorter in *OSort* and *OCSwap* can be replaced by the odd–even merging network [83] if the size of the input is a power of two as it achieves the same time complexity with a even smaller constant.

We designed a primitive *OMap* that can obliviously store a list of elements into a table indexed by a position map or a hash function. As *OMap* can be used to initiate a PORAM or a hash table obliviously, we specifically introduce it here. Algorithm 1 outlines how to implement *OMap*. It mainly uses the primitive *OSortByOrder* to make the map operation oblivious. *OSortByOrder* obliviously sorts a list of elements with a given order. We implemented *OSortByOrder* with a user-defined oblivious compare-and-swap primitive *OCSwap*. We implemented a shuffle primitive *OShuffle* to obliviously shuffle a list *List* as in Algorithm 1. We also used Bitonic sorter to merge (*OMerge*) two ordered lists of equal length. As a merging network is the second half of the Bitonic sorting network, it is very efficient in merging two ordered lists compared with a Bitonic sorter. *OMerge* takes only ∼6.25 ms in sorting two ordered heavy parts with recommended setting (details can be found in Section 7.1.4), while *OSortByOrder* needs ∼46 ms.

5.3.1. Security of Algorithm 1

We prove in the following that all of the algorithms in Algorithm 1 are data-oblivious.

Theorem 1. *Let f be an algorithm composed of single statements (including function calling statements) and loop blocks. The algorithm f is said to be* ***data-oblivious*** *if each statement (including statements in the loop blocks) is oblivious and if the number of iterations for each loop block is known in advance or not treated as a secret.*

Proof. The proof is obvious as the code access trace is predefined only by the number of iterations for each loops and not affected by any input parameters. □

According to Theorem 1, *OCSwap* is **data-oblivious**. Algorithms *OSortByOrder* and *OShuffle* are **data-oblivious** if *OSort* is data-oblivious and if the size of the input list is known in advance or not treated as a secret.

5.3.2. The Importance of Oblivious Shuffle

Oblivious shuffle (*OMap*, *OShuffle*) can be implemented by *OSort* in the list with a given secret array of orders. Given the maximum order T, the array of orders can be implicatively defined by a randomly chosen hash function or enumerated by instantiating each array element with a random number modulo T (Lines 2–4 in Algorithm *OShuffle*). After *OSortByOrder*, the list is shuffled according to the secret array of orders. The orders of each element of the shuffled list then form a new ordered array. We observe that, for the original list, any access pattern derived only by the ordered array of the shuffled list is data-oblivious. It is easy to understand as the oblivious shuffle breaks the tie between the two lists and the ordered array carries no information about the original list. This observation can help us in optimizing three (*Merge LxF, Heavy-change, Flow dist.*) of the five critical functions and Oblivious Sketch's initialization.

Algorithm 1: Oblivious map to a bucket table.

```
   // On input a $List$, position map $PM$ (or $Hash$), $T$ buckets each of
      size $Z$, return a $Table$ of $T$ buckets, if bucket's overflow
      happens, an $OverFlowList$ is also returned.
0  OMap($List$, $PM$ or $Hash$, $T$, $Z$):
1      new a $Table$ with $T$ dummy buckets, each $Z$ elements;
2      new an $OverFlowList$ to record overflowed elements;
3      new a $RandList$ with elements from $List$, order from $PM$ or
        Hash($List[0,\dots,|List|-1]$) $mod$ $T$;
4      OSortByOrder($RandList$);
5      for $i \leftarrow 0$ to $|RandList|-1$ do
6          Store $RandList[i].elmt$ in the bucket $RandList[i].order$, if overflow happens,
            store it into $OverFlowList$;
7      return ($Table$, $OverFlowList$);
0  OShuffle($List$):
1      new a $RandList$ with $|List|$ elements;
2      for $i \leftarrow 0$ to $(|List|-1)$ do
3          $RandList[i].elmt \leftarrow List[i]$;
4          $RandList[i].order \leftarrow$ rand() $mod$ $|List|$;
5      OSortByOrder($RandList$);
6      copy all the elements in $RandList$ to $List$;
0  OSortByOrder($List$):
       // implement OSort's CompAndSwap().
1      OSort.CompAndSwap() $\leftarrow$ Function (int i, int j) :
2          bool $greatL \leftarrow$ OGreater($List[i].order > List[j].order$);
3          OCSwap($List[i]$,$List[j]$,$greatL$);
4      OSort($List$);
   // Obliviously swap $ElementA$ and $ElementB$.
0  OCSwap($ElementA$, $ElementB$, $cond$):
1      $ElementT \leftarrow$ OCMoveEx($ElementA$, $ElementB$, $cond$);
2      $ElementA \leftarrow$ OCMoveEx($ElementB$, $ElementA$, $cond$);
3      $ElementB \leftarrow ElementT$;
```

Definition 1. *We denote by* ***oblivious shuffle block*** *any statement block that comprises two parts; the first part is an oblivious shuffle statement of an original list, and the second part is data access statements for any element of the shuffled list only controlled by the new ordered array (which has no confidential information and can be public).*

Lemma 1. ***Oblivious shuffle block*** *is* ***data-oblivious*** *if the size of the list to be obliviously shuffled is known in advance or is not treated as a secret.*

Proof. According to the explanation of *"The importance of oblivious shuffle"*, the oblivious shuffle breaks the tie between the two lists and the ordered array carries no information about the original list. As any data access statement in the oblivious shuffle block accesses elements of the shuffled list controlled by the new ordered array, which means that the data access pattern in this block can be derived only by the new ordered array and leaks no information about the original list, the oblivious shuffle block is data-oblivious. □

Theorem 2. *Let f be an algorithm composed of single statements, loop blocks of statements, and oblivious shuffle blocks. The algorithm f is said to be* ***data-oblivious*** *if each statement (including statements in the loop blocks) is oblivious and if the number of iterations for each loop block and the size of each lists to be obliviously shuffled are known in advance or not treated as a secret.*

As Lines 4–6 in Algorithm *OMap* is an oblivious shuffle block, according to Theorem 2, *OMap* is **data-oblivious** if T and the size of *List* are known in advance or not treated as a secret.

We also design critical primitives *ODelta* and *OCount* in Algorithm 2. **ODelta** is one of the critical parts in realizing oblivious heavy-change candidates calculation, as shown in Section 5.4.4. Specifically, for two lists L_1 and L_2 of ID–value pairs, *ODelta* (in Algorithm 2) obliviously calculates the delta of values if two IDs are the same. Save any pair if its delta or value is greater than a threshold γ; otherwise, dummy it. As *ODelta* only comprises oblivious statements, according to Theorem 1, it is **data-oblivious** if the size of L_1 and L_2 are known in advance or not treated as a secret. Though it can directly calculate heavy-change candidates, it is time consuming as its time complexity is $O(|L_1| \times |L_2|)$. However, there exists room for optimization; details can be found in Section 5.4.

Algorithm 2: Oblivious delta and count.

```
0  ODelta($L_1$,$L_2$,$\gamma$):
1      new an array $Match$ with $|L_2|$ booleans, all false;
2      new a dummy element $dmy \leftarrow (DummyID, 0)$;
3      for $i \leftarrow 0$ to $|L_2| - 1$ do
4          for $j \leftarrow 0$ to $|L_1| - 1$ do
5              $isMatch \leftarrow L_1[j].id \equiv L_2[i].id$;
6              $L_1[j].value \leftarrow$ OCMove($|L_1[j].value - L_2[i].value|, L_1[j].value, isMatch$);
7              $cond \leftarrow isMatch$ && $(L_1[j].value < \gamma)$;
8              $L_1[j] \leftarrow$ OCMoveEx($dmy, L_1[j], cond$);
9              $Match[i] \leftarrow$ OCMove($isMatch, Match[i], isMatch$);
10     for $i \leftarrow 0$ to $|L_2| - 1$ do
11         $L_2[i] \leftarrow$ OCMoveEx($dmy, L_2[i], Match[i]$);
0  OCount($OrdList$ in ascending order):
1      new a list $FiledDist$ with $|OrdList|$ $val$-$cnt$ pairs;
2      $curCnt \leftarrow 0$;   $cond \leftarrow 1$;   $size \leftarrow |OrdList|$;
3      for $i \leftarrow 0$ to $size - 2$ do
4          $curCnt \leftarrow$ OCMove($1, curCnt+1, cond$);
5          $FiledDist[i].cnt \leftarrow curCnt$;
6          $cond \leftarrow (OrdList[i+1].field > OrdList[i].field)$;
7          $FiledDist[i].val \leftarrow$ OCMove($OrdList[i].field, VALUE.MAX, cond$);
8      $FiledDist[size-1].val \leftarrow OrdList[size-1].field$;
9      $FiledDist[size-1].cnt \leftarrow$ OCMove($1, curCnt+1, cond$);
10     OSortByOrder($FiledDist$) in ascending order by $val$;
11     return $FiledDist$;
```

OCount is used to count the occurrence of the same value of a specific field from a list. The trivial oblivious solution to count the occurrences of all the distinct values is to resort to a double loop with the outer loop scanning the specific field of each element in the list and the inner loop scanning the same length *(value, count)* list *FiledDist* and only by adding the count by one with the same value as the field value obliviously. If the list needing counting is an ordered list, it becomes straightforward to calculate the field value distribution. As shown in Algorithm 2, upon inputting an ascending ordered list $OrdList$ with the specific field, *OCount* first scans the $OrdList$; for the same field value, count the number of elements and save a dummy $(VALUE.MAX, 0)$ pair to $FiledDist[i]$ if the next field value is not changed or save $(value, count)$ to $FiledDist[i]$ (Lines 5–7). It then obliviously sorts (*OSortByOrder*) $FiledDist$ in an ascending order by the values (Line 10). By doing so, dummy pairs are pushed to the end of $FiledDist$. According to Theorem 1, *OCount* is **data-oblivious** if the size of $OrdList$ is known in advance or not treated as a secret. *OCount* can be directly used to calculate fast oblivious flow distribution. Additionally, the idea in the design of *OCount* can help optimize the function of *Merge LxF*.

5.4. *Oblivious Functions*

In the following, we identify seven critical functions used in implementing oblivious sketch merging and measurement metrics estimation and illustrate how to optimize the five critical functions. All of the run times of the seven functions with the recommended setting are shown in Section 7.2.

5.4.1. *Merge LxL:* Merge of Two Light Parts

By using assembly AVX instructions, `vpmaxub` compares 32 pairs of unsigned 1-byte integers in two 256-bit YMM registers (such as registers `ymm13` and `ymm14`, there have 16 YMM registers; all are user-mode available) and stores the 32 maximum values in `ymm15`; `vpadd` sums 32 pairs of unsigned 1-byte integers in `ymm13` and `ymm14` and stores the 32 sum values in `ymm15`:

```
vpmaxub %%ymm13, %%ymm14, %%ymm15
vpaddd  %%ymm13, %%ymm14, %%ymm15
```

We can handle 32 byte-counters in one SIMD instruction in parallel. This greatly reduces the max-merge or sum-merge of light parts. As we linearly scan the two light parts, apparently the function *Merge LxL* is **data-oblivious** according to Theorem 1.

5.4.2. *Merge HxH:* Merge of Two Heavy Parts

It is trivial to merge two heavy parts by simply calling *OSortByOrd* or by calling *OMerge* if they are two ordered lists. The flowID-size pair's size implicitly defines the order. Apparently, it is **data-oblivious** according to Theorem 1. The result is a descending ordered list of double lengths of the heavy part. The upper part of the ordered list is the heavy part of the merged sketch and *the heavy-hitter candidates* if the merged sketch is the final "global sketch", while the lower part is used to merge with the intermediate light part of the merged sketch.

5.4.3. *Merge LxF:* Merge of Light Part and Many Heavy Flows

To obliviously obtain the final light part of the merged sketch, after *Merge LxL* and *Merge HxH*, we should obliviously merge the remaining lower part heavy flows from *Merge HxH* with the intermediate light part from *Merge LxL*. One trivial solution is to insert each remaining heavy flows into the PORAM-stored light part. According to our experiment (as shown in Section 7.3.1), it takes ∼922 ms to insert the remaining 16,384 heavy flows. Though it is not so slow, if the merged sketch is not the global sketch, we should not only store the light part in a PORAM but also extract all of the PORAM blocks sequentially to rebuild the light part. Apparently, it is not a good option. In contrast the other trivial non-PORAM solution is to resort to a double loop with the outer loop scanning the heavy flows and the inner loop scanning all counters. It consumes ∼2663 ms, which is intolerable.

Inspired by the idea of Divide and Conquer and the fast implementation of *OCount*, as shown in Figure 6, we can group the light part into a bucket table LB; each bucket contains a fixed number nc of counters and a bucket ID so that a YMM register can hold all of them. Here, in both Figures 6 and 7, 'f_i' indicates flow ID i, 'b_j' indicates block ID j, 'D' stands for dummy block, and 'X' stands for dummy data. By doing so, the size of the problem is reduced to ∼1/30 of the original one. For each heavy flow, we initialize the corresponding bucket, hash the flow ID by the hash function associated with the light part to obtain the position *pos* in the light part, then divide *pos* by *nc* to obtain the bucket ID, reset all of the counters in the bucket to 0, and set the counter in *pos* modulo *nc* to the size of the flow. Then, we obtain the initialized bucket table *HB* for table *HF* of heavy flows. *OSortByOrder HB* in ascending order by bucket ID and merge all of the buckets of the same bucket ID in a way similar to *OCount* to form a ascending ordered table *OB*; dummy all of the same buckets except the last one, and *OSortByOrder* them to put dummy buckets at the bottom of the table *OB*. The merger of two same ID buckets can be implemented by calling AVX instruction `vpmaxub` for max-merge or `vpaddd` for sum-merge. The *LB* and *OB* form a big bucket table, and we perform the same operations on the big table as those for *HB* to obtain the final merged bucket table *MB*. The light part of the merged sketch can now be obtained by extracting all of the counters in *MB* sequentially. As all of the operations are data-oblivious, the algorithm for *Merge LxF* is also **data-oblivious** according to Theorem 1.

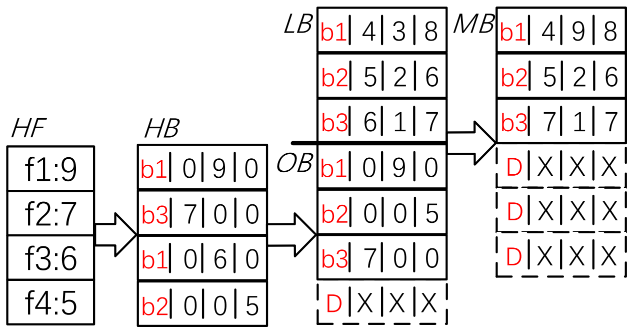

Figure 6. Obliv. max-merge of light part and heavy flows.

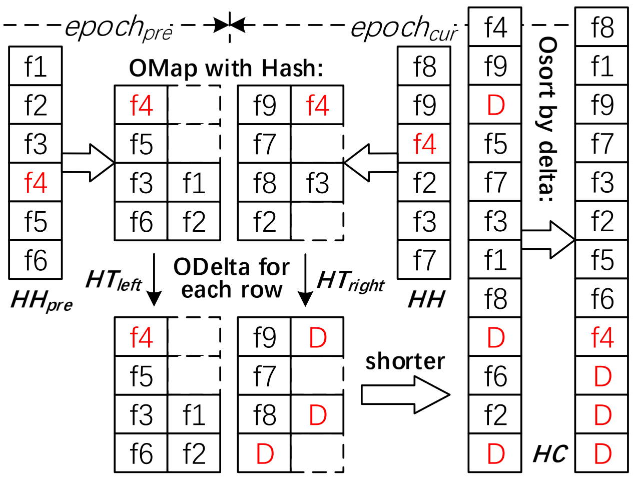

Figure 7. Oblivious heavy-change candidates calculation.

5.4.4. *Heavy-Change:* Heavy-Change Candidates Calculation

To calculate the heavy-change candidate list *HC* in the current measurement epoch, we need to calculate the change in the same flow from two adjacent heavy-hitter candidate lists. Though *HC* can be directly calculated by calling *ODelta*, the time cost (∼4.8 s, see Section 7.3.1) for *ODelta* with a recommended setting is intolerable. Inspired by the idea of Divide and Conquer, as shown in Figure 7, we first reduce the size of the input for *ODelta* by obliviously mapping (*OMap*) the ordered heavy-hitter candidate list HH_{pre} from the previous epoch and the current ordered list *HH* into two bucket hash tables HT_{left} and HT_{right} associated with a same randomly chosen hash function; each table has the same smaller size than $|HH|$, and each bucket has enough room to host heavy flows hashed to it. For buckets of the same row *i* of HT_{left} and HT_{right}, we call *ODelta*($HT_{left}[i]$,

HT_{right}*[i], γ_{min})* (see Algorithm 2) to obtain the change. Only the elements (in fact, they are flowID–size pairs as in list HF from Figure 6) in the solid squares in Figure 7 participate in the calculation. Note that the threshold used to filter candidate pairs is set to a predefined minimum meaningful value γ_{min}. Then, we put each solid element including dummy ones from the two tables into a longer list and feed the list into *OSortByOrder* to obliviously obtain a descending-ordered list HC of heavy-change candidates. We can know from the above description and Figure 7 that the code block including *OMap* and *ODelta* loop forms an oblivious shuffle block, as the length of each variable-length bucket of HT_{left} and HT_{right} is irrelevant to HH_{pre} and HH, respectively. As *OMap* and *ODelta* are data-oblivious and the operations above form an oblivious shuffle block, according to Theorem 2, the calculation of the ordered list HC of heavy-change candidates is also **data-oblivious**.

5.4.5. *Counter Dist.:* Counter Distribution Calculation

The counter distribution array $CountDist$ is an array of 256 integers; each $CountDist[i]$ is used to count the number of counters with the same value i from the light part of the global sketch. The trivial oblivious solution to calculate $CountDist$ is to resort to a double loop with the outer loop scanning each counter and the inner loop scanning the $CountDist$ and only adding the corresponding position obliviously. A more efficient solution is to replace the inner longer loop with a AVX instruction `vpadd`-supported shorter loop. As `vpadd` can handle either double-word integers, the number of iterations for the inner loop is shorted from 256 to 32. Apparently, *Counter dist.* is **data-oblivious**, according to Theorem 1.

5.4.6. *Flow Dist.:* Flow Distribution Calculation

The flow distribution list $FlowDist$ is a list of *(value, count)* pairs; each pair is used to record the number *count* of flows with the same flow size *value* from the heavy part of the global sketch. They can be calculated by simply calling $OCount$. Apparently, *Flow dist.* is **data-oblivious**.

5.4.7. Stash Table Initialization

To query flow size, we should not only query the light part but also query the heavy part of the global sketch. As linearly scanning the heavy part is time consuming, we should first initialize a hash table for fast lookup and store the table into PORAM for oblivious access. For a moderate size bucket, such as a bucket that hosts at most seven flowID–size pairs, it is challenging to initialize a bucket hash table with approximately double the size of the heavy part without bucket overflow. Thus, we should obliviously initialize a moderate-size-bucket table and use a stash to record the overflowed heavy flows. After randomly choosing a hash function, we *OMap* (see Algorithm 1) the heavy part by the hash. The returned *Table* is the bucket hash table, while the returned $OverFlowList$ is the stash. Apparently, *Flow dist.* is **data-oblivious**.

5.5. Oblivious PORAM Controller

5.5.1. OPORAM.Init

It is also easy to store a piece of *Data* into PORAM by using $OMap$. On input *Data* and PORAM parameters (Z blocks, each of B bytes), we first initialize a PORAM with no data by calling *OPORAM.Init(; sizeof(Data), Z, B)* (the Oblivious Controller). Then, we partition *Data* into a block list $BList$; each B-byte block has type *(bid, pos, data)* and contains *(B - (the total size of bid and pos))* bytes payload from *Data*, here *bid* is the block ID and *pos* a random number modulo $numLeaf$ (numLeaf is the number of leafs of the PORAM). The position map PM of PORAM is initialized by all of the *pos*. Third, we call *OMap* with PM to obliviously map $BList$ into the leaf bucket table of PORAM (for each level i of the PORAM tree, all of the level i buckets form a level i bucket table). The overflowed blocks are then inserted into the level *(leaf - 1)* bucket table with index ($Block.pos >> 1$). Following the obliviousness analysis of *Heavy-change*, the operations above form an oblivious shuffle block and the initialization of PORAM *OPORAM.init* is **data-oblivious** according to Theorem 2.

5.5.2. OPORAM.Evict

Note that the controller of the PORAM should works obliviously as it is also hosted in the enclave. We design a controller strictly in accordance with ZeroTrace's recommendation [73]. Table 1 presents the maximum real blocks that reside in the stash and the maximum real blocks after evicting the current path to the tree after 9000K random accesses. This gives us a clue to optimize the access speed of the oblivious controller. According to ZeroTrace, the controller consumes most of the time. This mainly because the stash size is set to 90 for the general case. In our network measurement scenario, the underlying sketch is no more than 1M, which is very small compared with default data size supported in ZeroTrace. As a position map changes randomly independent of the data access trace, the size of the stash after eviction leaks no information about the data access pattern. However, the block access trace in the stash does leak information about the data access pattern. Therefore, we propose to optimize the access speed by evicting all of the dummy blocks in the stash and by pushing the real blocks to the head of the stash after path eviction. In our recommended setting, we reset the stash size to be no less than five after stash compression. Section 7.3.2 illustrates the effectiveness of our stash compression.

Table 1. PORAM stash size statistics.

	1000K	**3000K**	**5000K**	**7000K**	**9000K**
Max	45	45	51	45	48
Max after Evict	10	9	8	10	12

6. Secure Network Measurement Service

6.1. Oblivious Merge of Sketches

To merge two underlying sketches obliviously, we only need to call *Merge LxL*, *Merge HxH* and *Merge LxF* sequentially.

6.2. Oblivious Access of the Global Sketch

Once all of the local sketches are merged into a global "one-big sketch", it should be stored into PORAM for oblivious query. Here, we store the light part and the heavy part into different PORAMs by calling *OPORAM.init*. To access a location of the PORAM stored light part, we should calculate the block ID and the relative index of the block to obliviously obtain the counter's value by calling *OPORAM.access*. Note that a PORAM block contains only one bucket of the stash table of the heavy part in our recommended setting. To access a location of the PORAM stored heavy part, we should also calculate the block ID and scan the block to check if any flowID–size pair in that block matches the queried flow ID.

6.3. Oblivious Measurement Tasks

6.3.1. Flow Size Estimation

It is easy to estimate the size of a flow by querying the PORAM stored global sketch with the $flowID$. We first access the PORAM stored heavy part with a given location calculated by associated hashing of $flowID$ and linearly scan the stash of the heavy part's stash table. Then, we access the PORAM stored light part with a given location calculated by associated hashing of $flowID$, finally returning the estimated size obliviously. As access to PORAM is data-oblivious, according to Theorem 1, the estimation of flow size is also **date-oblivious**.

6.3.2. Heavy-Hitter Detection

As we stored the ordered list HH of heavy-hitter candidates, to detect heavy-hitters by a user-defined threshold γ, we just scan HH and copy the flowID–size pairs into a list until it meets a pair in which the size *value* is smaller than γ. The list is then returned as a response for the bigger-than-γ heavy-hitter request.

6.3.3. Heavy-Change Detection

As we stored the ordered list HC of heavy-change candidates, for the bigger-than-γ heavy-change request, we perfirm the same procedure as the heavy-hitter request, except that the candidate list is replaced by HC.

6.3.4. Cardinality Estimation

The cardinality of the global sketch comprises the heavy part cardinality $Card_H$ and the light part one $Card_L$. $Card_H$ can be calculate by simply scanning the ordered heavy part and by increasing $Card_H$ by 1 until it meets a dummy flow. $Card_L$ can be calculated using the linear counting algorithm from [84]:

$$ratio \leftarrow (w - CountDist[0].value)/w$$
$$Card_L \leftarrow -w \times \ln ratio$$

For the cardinality request, we return the saved $(Card_H + Card_L)$.

6.3.5. Flow Distribution Estimation

Note that the flow distribution estimation is semi-supported, as we use the basic MRAC algorithm proposed by [11] as our non-oblivious form solution. We only obliviously support the online part of MRAC and respond to the flow distribution request by returning a counter distribution array. As the offline part is timing-consuming, we leave the calculation of it to the network applications that issue this type of request. According to [37], flow entropy can be calculated if flow distribution is known. Therefore, for network applications that want to request flow entropy, they can issue flow distribution requests instead.

We only respond to a flow distribution request using the stored counter distribution array $CountDist$ and the stored heavy flow distribution list $FlowDist$. Once a network application obtains the required $(CountDist, FlowDist)$ pair, it calculates the flow distribution by first estimate the distribution of the light part by calling the basic MRAC algorithm in [11] with $CountDist$ and then by merging it with $FlowDist$.

6.4. Implementation

6.4.1. Implementation Considerations

In the above Section 4.1, we outlined the working flow of our network measurement service. In the untrusted environment, the communication of a local switch with merge server, merge server with the measurement service in the cloud, and a network application with the measurement service is all encrypted with encryption keys exchanged secretly by remote/local attestation at the initial stage. To deploy the measurement service, there are several other considerations. Below, we illustrate them from two different aspects: security and performance.

6.4.2. Request Type Indistinguishability

In some situations, not only the flow statistics and metrics but also the request type and the size of the responded data should be protected. Whether to protect request type is network application specific. Furthermore, the size of the responding data may leak not only the request type but also the parameter associated with each request. Without padding, the flow size and cardinality consume the least memory footprint, while the number of the responded heavy-hitters or heavy-changes reveals information about the user-defined threshold in the related request. For the most powerful adversary in our threat model, an attacker can obtain the code and data access pattens in the cache line granularity. Thus, algorithms in the enclave can also be differentiated by the code access pattern.

In our implementation, we provide two options: one to support a full request type as being indistinguishable and the other to reveal only flow size request type. It is easy to achieve a full request type being indistinguishable if we call all five measurements one by one for each request and returning requests with padding at the maximum length of all responses. Though we stored the metrics in advance, it is time consuming to answer a flow size query (∼94 µs per query with recommended setting). The second option splits requests into two categories: one for flow size requests and the other for the other four requests. As the scale of the time cost for flow size query is higher than the other requests, this split is also helpful if request type leakage is not treated as a security threat. By picking flow size requests into a different worker thread in the enclave, the non-flow-size requests can be responded to quickly. The size of responded data can be hidden by partitioning the data into fixed-size segments and by padding the last one before the response. Below, we outline how to handle requests in an efficient way.

6.4.3. Switchless Request and Response

It is not free to securely transition from the untrusted mode to trusted mode (by `ECALL`) and the reverse (by `OCALL`) [42,85–87]. Specifically, running an I/O-intensive network measurement service in enclave leads to significant performance degradations. Considering the huge difference in time cost for different request types, we propose the switchless request and response channel to handle requests efficiently.

Figure 8 illustrates the architecture of the switchless channel. ① Requests are organized into a request queue, each of which is, in fact, a pointer to a memory block from the block pool, recording the request originator and the encrypted request type and associated parameter. ② Once a request is detected by the enclave, it is read into the enclave, decrypted, and relayed to one of the two worker threads depending on its request type. ③ If it is a flow size request, it is handled by the enclave's worker thread 1. ④ As the response data to flow size request is just an integer, it is encrypted and written into the same block pointed by the corresponding request, and the pointer is then added into the response queue. ③′ If it is a non-flow-size request, it is handled by the enclave's worker thread 2 immediately. ④′ If the response data size exceeds that of a block, worker thread 2 will `OCALL` the block allocation function for one or several other blocks from the block pool. To make the block allocation function as a switchless `OCALL`, the EDL attribute `transition_using_threads` should be postfixed to it (details of using switchless calls can be found in [87]). By using a switchless `OCALL` mechanism, the `OCALL`'s are transformed to `OCALL` tasks relayed to worker thread 3. Worker thread 3 handles the block request task by calling the block allocation function and relays the return value to the `OCALL`. The block allocation function manages the block pool; once the required number of empty blocks are found, those blocks are organized into a linked list and returned by the pointer to this linked list to the `OCALL` in the enclave. Upon receiving all the required blocks, worker thread 2 encrypts the response data and writes it into those blocks. The pointer to the block list is then added to the response queue. ⑤ Once a response is detected by the untrusted part of the measurement server, it is transferred to the corresponding request originator.

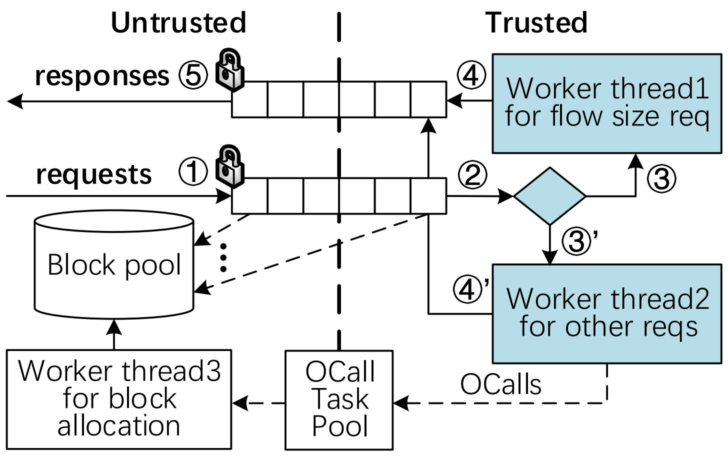
Untrusted
Trusted
responses ⑤
④
Worker thread1 for flow size req
requests ①
②
③
③'
Block pool
④'
Worker thread2 for other reqs
Worker thread3 for block allocation
OCall Task Pool
OCalls

Figure 8. The architecture of a switchless request and response.

As there has no transition between the untrusted mode and trusted mode in handling requests, the requests can be responded to efficiently.

7. Evaluation

As the local switch uses Elastic Sketch [37] as the underlying sketch to record and update flow statistics, we omit the accuracy analysis and focus mainly on the performance gain of our optimized functions compared with trivial oblivious ones. We also test the flow-size estimation performance by varying the settings of sketch and PORAM.

7.1. Setup

7.1.1. Platform

We implement the prototype service and made the source code publicly available on Github [82]. The prototype service is deployed for test in an SGX-enabled laptop equipped with Intel Core i7-6500U 2.50GHz CPU (4 cores) and 8 GB RAM.

7.1.2. Traces

We used the same data set as OblivSketch: two one-hour public traffic traces CAIDA1 and CAIDA2 collected by Equinix-nyc monitor and Equinix-chicago monitor, respectively; they are published on the CAIDA official website [52]. CAIDA1 contains 1.56 billion packets and 4 million flows (use source IP as flow ID), while CAIDA2 contains 1.79 billion packets and 8.4 million flows. We divided the trajectory into different measurement time intervals. The flow cardinality of each interval is given in Table 2. We followed the non-oblivious Elastic Sketch [37] and used 5 s as the default measurement interval.

Table 2. The statistics of CAIDA traces (K: thousand, M: million).

Intervals	**1 s**	**5 s**	**10 s**	**30 s**	**60 s**	**120 s**	**240 s**
CAIDA1	30K	68K	110K	200K	284K	430K	690K
CAIDA2	43K	115K	178K	372K	600K	916K	1.47M

7.1.3. Baseline

We implemented the trivial oblivious solution based on multiple nested loops for each critical function and implemented a conventional non-oblivious solution.

7.1.4. Recommended Setting

For the recommended setting of the underlying sketch, we followed the choice of Elastic Sketch with a bit of variation for better performance without degrading the accuracy: 450 KB for the light part with one array of 8-bit counters and 128 KB (not 150 KB) for the heavy part with a list of 16384 flowID-size pairs, each 8 bytes. For the recommended setting of PORAM, we chose to set 5 blocks of size 64 bytes in each bucket and to build a PORAM stash with 105 blocks to ensure a negligible false positive rate (2^{-128}). The 64-byte block contains 2 bytes block ID, 2 bytes leaf position, and 60 bytes payload. As the memory read/write cache-line of our platform is 64 bytes and the enclave's read/write transactions *"are at the granularity of 512-bit blocks"* [88], we chose 64 bytes as the size of PORAM block. Note that no more than 55 blocks exist in the PORAM stash at any time in all our experiments with the recommended setting. For recommended setting of the stash table for the heavy part, we allocated 224 KB for the bucket hash table which is 1.75 times the heavy part, set 7 flowID-size pairs in each bucket (so that a 64 bytes PORAM block can host exactly one table bucket), and built a table stash that can host 1000 pairs. Note that the hash collision of our setting makes no more than 400 pairs stored in the table stash.

7.2. Critical Functions Performance

Table 3 illustrates a detailed time division for all eight functions to rebuild the metrics and PORAM-stored global sketch with recommended setting. Two solutions are used to compare our FO-Sketch. Besides the trivial solution, we also tested non-oblivious algorithms of all the critical functions; it is 18 times faster than our FO-Sketch. As our *Merge LxF* algorithm needs to obliviously sort four times to implement the functionality, it is the most time-consuming algorithm. Our *Heavy-change*, *Flow dist.*, *Stash table Init*, and *two PORAM Inits* also need to obliviously sort 3, 1, 1, and 2 times. As our *Merge HxH* needs only one oblivious merger of ordered lists, it is the one that consumes the least time, except *Merge LxL*. Note that *two PORAM Inits* initialize two PORAMs for the heavy part and the light part of the global sketch. Though *Flow dist.* and *Stash table Init* both have to sort the same number of items once, the size of the items in *Stash table Init* is double the size of the items in *Flow dist.* and *Stash table Init* is more complex. As we use SIMD instructions, our *Merge LxL* outperforms the non-oblivious one 80 times, though they both consume trivial time.

Table 3. Time consumption [a] (ms) for critical functions with the recommended setting.

Solution	*Merge LxL*	*Merge HxH*	*Merge LxF*	*Heavy-Change*	*Counter Dist.*	*Flow Dist.*	*Stash Table*	Total [b]
Trivial	2.859	6.254	2663.00	4780.00	626.10	2234.00	58.30	10414
FO-Sketch	0.045	6.254	240.20	165.20	66.59	20.16	58.30	600
Non-Obliv	3.370	2.271	1.44	16.34	1.19	0.93	5.77	32

[a] The decryption of two sketches consumes ∼0.31 ms. Only ∼247 ms is needed for merging one more sketch and $\sim 247 log(n)$ ms for n sketches if hierarchically organized. [b] The initialization of two PORAMs for oblivious solutions takes 43.47 ms.

Scalability of *OSort*-Supported Critical Functions

We picked four of our *OSort*-supported functions to test their scalability and the result can be found in Figure 9. The units (i.e., the scale 1) on the legend of Figure 9 for different functions are the recommended size. *Merge HxH* can be fed by two ordered lists (*OMerge*-supported) or two unordered lists (*OSort*-supported). It is very clear that *OMerge* (∼6 ms for scale 1) is extremely faster than *OSort* (∼46 ms) for the same input size. We chose *OSort*-supported *Merge HxH* as the base line to illustrate how *OSort* affects the performance. As we can see, the number of times *OSort* executed and the input size of the function linearly affect the run time. A clear drop can be seen near scales 4 and 8 for *Heavy-change*. It is because, in our implementation, if the input size is of power 2, we use an odd–even merge sorter instead of the famous Bitonic sorter (as explained in Section 5.3).

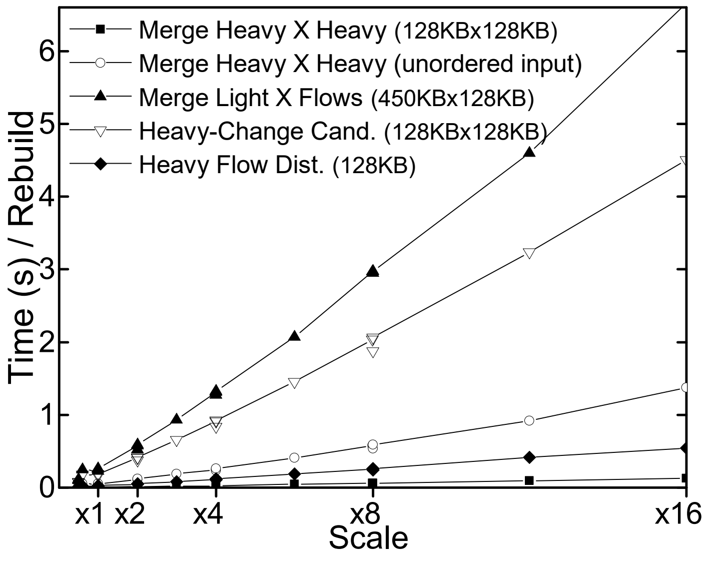

Figure 9. Input size vs. computation time.

7.3. PORAM Performance

7.3.1. PORAM Block Size Selection

In addition to the theoretical explanation in Section 7.1 of why we chose 64 bytes as the PORAM block size, we illustrated the benefit of a 64-byte block by experimentation. As shown in Figure 10, to store 450 KB data in PORAM, we varied the size of the PORAM block. PORAM with a block size of no bigger than 64 bytes has almost the same least access time, while PORAM with a block size of no smaller than 64 bytes has nearly the same least memory footprint.

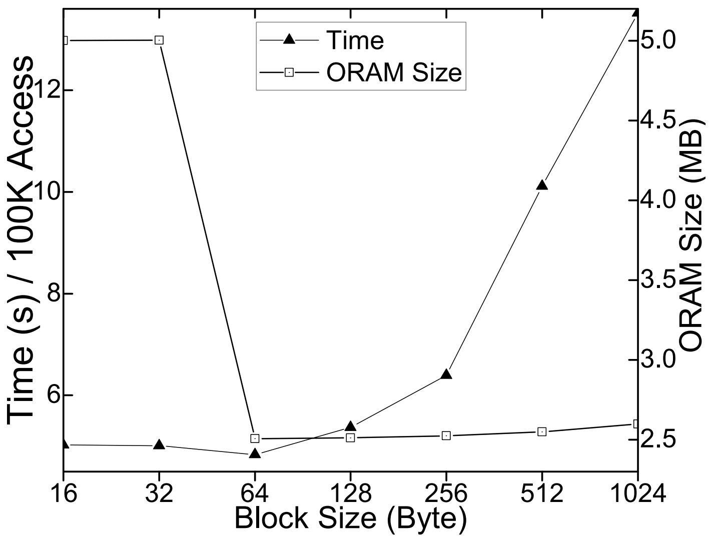

Figure 10. Block size selection for 450 KB data.

7.3.2. Scalability of PORAM Access

In Section 5.5, we proposed using *stash compression* to optimize the oblivious controller's access to PORAM. Here, we prove its effectiveness through the experiment with a 64-byte PORAM block. As shown in Figure 11, we compared the time cost after executing 100 K PORAM random accesses when using stash compression with that when using no optimization by varying the number of blocks. When setting the initial stash size to 50 and using no optimization, the time cost is intolerable. When setting the initial stash size to 25 and using no optimization, the time cost is 1.5 times longer than when using stash compressing with a minimum stash size of 5. The time cost of non-optimization methods is random on a small scale and tends to become longer as the data becomes larger. This phenomenon for a small scale is reasonable as the stash size is not set to 105. There is a probability during the accesses that the maximum stash size can be reached at larger than 50, and with no optimization, the stash size cannot be reduced. As the time when the stash size becomes larger than 50 is random, the time cost is also random on a small scale. We omit to draw line for a fixed stash size of 105, as it takes more than 27 s to finish 100K oblivious accesses for 2^{10} blocks. A clear jump can be observed at a block number of the power of 2 in the access time trace when using stash compressing and in an ORAM memory consumption trace. It is a typical phenomenon for PORAM as the height of the tree is increased only when the number of blocks reaches a power of 2.

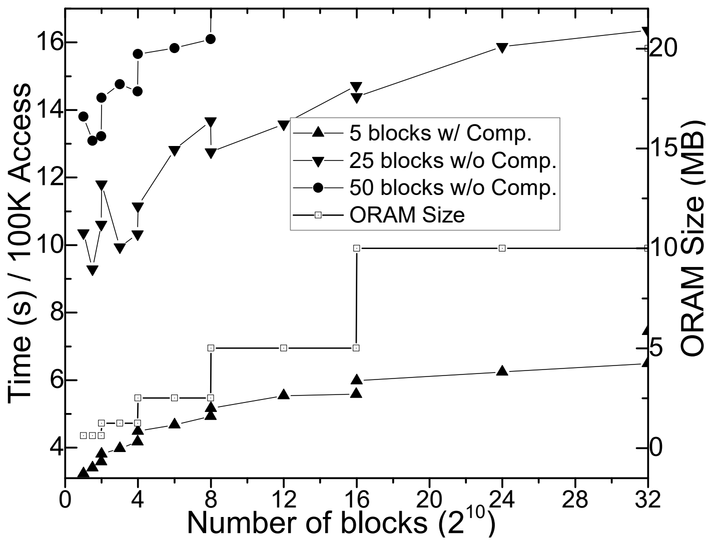

Figure 11. Data size scale with compression.

7.4. Comparison with OblivSketch.

Though FO-Sketch is inspired by OblivSketch [51], we did not directly compare the two in terms of performance above. We argue that our FO-Sketch is superior to OblivSketch in the following three aspects:

7.4.1. Obliviousness

OblivSketch is not as secure as FO-Sketch in terms of preventing access pattern leakage in the enclave. We give some examples. *"OCMSketchScan"* of Algorithm 6 in OblivSketch [51] is not oblivious as Line 4 "$n_j + = oselector(0, 1, T[i].bid = -1)$" of the algorithm may leak access pattern to a powerful adversary described in Section 4.2. Note that j in Line 3 is assigned the value of the ith counter, as accessing the jth element in the "distribution array $\{n\}$" can tell the adversary the value of the ith counter. We also checked the source code [89] they provided, which implements the OblivSketch and found that some functions in it were not oblivious too. Specifically, we found that the PORAM's controller was not implemented obliviously, though it is claimed in OblivSketch [51] that the action of it should be oblivious. Line 119 "$uint32\_tx = position[bid];$" in the head file "PathORAM.h" [90] of the OblivSketch source code may leak access patterns and is not oblivious. Unlike OblivSketch, we carefully checked each critical function in our implementation [82] of the FO-Sketch and proved their obliviousness.

7.4.2. The Rationality of PORAM Parameter Setting

According to Path ORAM [39] and ZeroTrace [73], the PORAM parameter setting of OblivSketch for storing Elastic Sketch is not reasonable. It is claimed that the size of the storage block's payload of "OCMSketch" in OblivSketch [51] is set to 1 byte. As the storage block of PORAM has another two fields—block ID and leaf position—for a 450 KB CM Sketch, the block ID field consumes $\lceil log_2(450 \times 2^{10}) \rceil = 19$ bits to represent 450 K blocks and the leaf position field also needs $\lfloor log_2(450 \times 2^{10}) \rfloor = 18$ bits to represent 2^{18} leaves. Therefore, storing it in PORAM requires $\lceil((19 + 18)/8 + 1)\rceil \times 5 \times (2^{19} - 1) \approx 15$ MB of secure memory space, which is very inefficient. Compared with OblivSKetch, our FO-Sketch only needs 2.5 MB to store a 450 KB CM Sketch on PORAM, significantly saving secure memory space.

7.4.3. Generality

As shown in Table 2, when the measurement period is greater than 10 s, the number of flows transferred in a typical link may be greater than 200 thousand; thus, OblivSketch needs more than 2.5 s to generate a PORAM-stored Elastic Sketch from those flows. To make matters worse, for a local network with a large amount of traffic and burst traffic, it is challenging to accurately update the flow size information due to the switch's limited computing and storage capabilities. In OblivSketch, flow cardinality is a crucial factor affecting the time cost of generating Elastic Sketch. In an extreme case, OblivSketch needs about 101 s to update 6.6 million (flowID, size) pairs to obtain the final Elastic Sketch, which is intolerable. In contrast, FO-Sketch circumvents the impractical solution of tracking each flow in a switch. It takes only 600 ms to merge two Merge Sketches and takes $247log_2(n) + 353$ ms to merge n Merge Sketches from n switches into a Global Sketch. In a nutshell, the time cost of Global Sketch's generation in FO-Sketch has nothing to do with the flow cardinality in each measurement period and is only affected by the number of switches.

8. Conclusions and Future Directions

This paper proposed FO-Sketch, a fast, oblivious network measurement service running in an Intel SGX-created secure container—enclave—in the public cloud. FO-Sketch can support large-scale network monitoring with heavy traffic volume and burst traffic by optimizing critical oblivious functions.

Several interesting future directions are suggested by this work. The most immediate is whether an efficient oblivious flow distribution estimation algorithm without offline calculations exists. We also expect that more measurement metrics can be calculated by analyzing our Global Sketch. As oblivious shuffle or sort play important roles in optimizing oblivious functions, we also expect a more efficient oblivious shuffle or sort algorithm, especially in a distributed fashion.

Author Contributions: Conceptualization, L.L., Y.S. and Z.Z.; formal analysis, Y.S.; funding acquisition, Y.S.; investigation, S.Z.; methodology, L.L.; project administration, Z.Z.; resources, Y.S.; software, L.L. and S.Z.; supervision, Y.S.; validation, Z.Z.; visualization, S.Z.; writing—original draft preparation, L.L. and Z.Z.; writing—review and editing, L.L., Y.S. and Z.Z. All authors have read and agreed to the published version of the manuscript.

Funding: This work was supported in part by the National Key Research and Development Program of China under Grant 2018YFE0207600; in part by the Key Research and Development Program of Shaanxi Province under Grant 2019ZDLGY13-06 and 2021KWZ-04; in part by the Youth Foundation of Hebei Educational Committee under Grant QN2020251.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. McKeown, N.; Anderson, T.; Balakrishnan, H.; Parulkar, G.; Peterson, L.; Rexford, J.; Shenker, S.; Turner, J. OpenFlow: Enabling innovation in campus networks. *ACM SIGCOMM Comput. Commun. Rev.* **2008**, *38*, 69–74. [CrossRef]
2. cisco.com. Cisco IOS NetFlow. Available online: https://www.cisco.com/c/en/us/products/ios-nx-os-software/ios-netflow/index.html (accessed on 20 December 2020).
3. Benson, T.; Anand, A.; Akella, A.; Zhang, M. MicroTE: Fine grained traffic engineering for data centers. In Proceedings of the Seventh Conference on emerging Networking EXperiments and Technologies, Tokyo, Japan, 6–9 December 2011; pp. 1–12.
4. Feldmann, A.; Greenberg, A.; Lund, C.; Reingold, N.; Rexford, J.; True, F. Deriving traffic demands for operational IP networks: Methodology and experience. *IEEE/ACM Trans. Netw.* **2001**, *9*, 265–279. [CrossRef]
5. Curtis, A.R.; Mogul, J.C.; Tourrilhes, J.; Yalagandula, P.; Sharma, P.; Banerjee, S. DevoFlow: Scaling flow management for high-performance networks. In Proceedings of the ACM SIGCOMM 2011 Conference, Toronto, ON, Canada, 15–19 August 2011; pp. 254–265.
6. Narayana, S.; Sivaraman, A.; Nathan, V.; Goyal, P.; Arun, V.; Alizadeh, M.; Jeyakumar, V.; Kim, C. Language-directed hardware design for network performance monitoring. In Proceedings of the Conference of the ACM Special Interest Group on Data Communication, Los Angeles, CA, USA, 21–25 August 2017; pp. 85–98.
7. Alizadeh, M.; Edsall, T.; Dharmapurikar, S.; Vaidyanathan, R.; Chu, K.; Fingerhut, A.; Lam, V.T.; Matus, F.; Pan, R.; Yadav, N.; et al. CONGA: Distributed congestion-aware load balancing for datacenters. In Proceedings of the 2014 ACM Conference on SIGCOMM, Chicago, IL, USA, 17–22 August 2014; pp. 503–514.
8. Liu, Z.; Bai, Z.; Liu, Z.; Li, X.; Kim, C.; Braverman, V.; Jin, X.; Stoica, I. Distcache: Provable load balancing for large-scale storage systems with distributed caching. In Proceedings of the 17th USENIX Conference on File and Storage Technologies, Boston, MA, USA, 25–28 February 2019; pp. 143–157.

9. Krishnamurthy, B.; Sen, S.; Zhang, Y.; Chen, Y. Sketch-based change detection: Methods, evaluation, and applications. In Proceedings of the 3rd ACM SIGCOMM Conference on Internet Measurement, Miami Beach, FL, USA, 27–29 October 2003; pp. 234–247.
10. Schweller, R.; Gupta, A.; Parsons, E.; Chen, Y. Reversible sketches for efficient and accurate change detection over network data streams. In Proceedings of the 4th ACM SIGCOMM Conference on Internet Measurement, Taormina, Italy, 25–27 October 2004; pp. 207–212.
11. Kumar, A.; Sung, M.; Xu, J.; Wang, J. Data streaming algorithms for efficient and accurate estimation of flow size distribution. *ACM Sigmetrics Perform. Eval. Rev.* **2004**, *32*, 177–188. [CrossRef]
12. Huang, Q.; Birman, K.; Van Renesse, R.; Lloyd, W.; Kumar, S.; Li, H.C. An analysis of Facebook photo caching. In Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles, Farminton, PA, USA, 3–6 November 2013; pp. 167–181.
13. Bar-Yossef, Z.; Jayram, T.; Kumar, R.; Sivakumar, D.; Trevisan, L. Counting distinct elements in a data stream. In Proceedings of the International Workshop on Randomization and Approximation Techniques in Computer Science, Cambridge, MA, USA, 13–15 September 2002; pp. 1–10.
14. Estan, C.; Varghese, G.; Fisk, M. Bitmap algorithms for counting active flows on high speed links. In Proceedings of the 3rd ACM SIGCOMM Conference on Internet Measurement, Miami Beach, FL, USA, 27–29 October 2003; pp. 153–166.
15. Van Adrichem, N.L.; Doerr, C.; Kuipers, F.A. Opennetmon: Network monitoring in openflow software-defined networks. In Proceedings of the 2014 IEEE Network Operations and Management Symposium (NOMS), Krakow, Poland, 5–9 May 2014; pp. 1–8.
16. Yassine, A.; Rahimi, H.; Shirmohammadi, S. Software defined network traffic measurement: Current trends and challenges. *IEEE Instrum. Meas. Mag.* **2015**, *18*, 42–50. [CrossRef]
17. Mijumbi, R.; Serrat, J.; Gorricho, J.L.; Bouten, N.; De Turck, F.; Boutaba, R. Network function virtualization: State-of-the-art and research challenges. *IEEE Commun. Surv. Tutorials* **2015**, *18*, 236–262. [CrossRef]
18. Herrera, J.G.; Botero, J.F. Resource allocation in NFV: A comprehensive survey. *IEEE Trans. Netw. Serv. Manag.* **2016**, *13*, 518–532. [CrossRef]
19. Benson, T.; Akella, A.; Shaikh, A.; Sahu, S. Cloudnaas: A cloud networking platform for enterprise applications. In Proceedings of the 2nd ACM Symposium on Cloud Computing, Cascais, Portugal, 26–28 October 2011; pp. 1–13.
20. Costa, P.; Migliavacca, M.; Pietzuch, P.; Wolf, A.L. NaaS: Network-as-a-Service in the Cloud. In Proceedings of the 2nd USENIX Workshop on Hot Topics in Management of Internet, Cloud, and Enterprise Networks and Services, San Jose, CA, USA, 24 April 2012.
21. Pearson, S. Privacy, security and trust in cloud computing. In *Privacy and Security for Cloud Computing*; Springer: Berlin/Heidelberg, Germany, 2013; pp. 3–42. ISBN 978-1-4471-4189-1.
22. He, J.; Ota, K.; Dong, M.; Yang, L.T.; Fan, M.; Wang, G.; Yau, S.S. Customized network security for cloud service. *IEEE Trans. Serv. Comput.* **2017**, *13*, 801–814. [CrossRef]
23. Wang, N.; Fu, J.; Li, J.; Bhargava, B.K. Source-location privacy protection based on anonymity cloud in wireless sensor networks. *IEEE Trans. Inf. Forensics Secur.* **2019**, *15*, 100–114. [CrossRef]
24. Chaudhry, S.A.; Irshad, A.; Yahya, K.; Kumar, N.; Alazab, M.; Zikria, Y.B. Rotating behind Privacy: An Improved Lightweight Authentication Scheme for Cloud-based IoT Environment. *ACM Trans. Internet Technol. (TOIT)* **2021**, *21*, 1–19. [CrossRef]
25. Jagadeesan, N.A.; Pal, R.; Nadikuditi, K.; Huang, Y.; Shi, E.; Yu, M. A secure computation framework for SDNs. In Proceedings of the Third Workshop on Hot Topics in Software Defined Networking, Chicago, IL, USA, 22 August 2014; pp. 209–210.
26. Hong, S.; Xu, L.; Wang, H.; Gu, G. Poisoning network visibility in software-defined networks: New attacks and countermeasures. In Proceedings of the Ndss 2015, San Diego, CA, USA, 8–11 February 2015; Volume 15, pp. 8–11.
27. Xu, L.; Huang, J.; Hong, S.; Zhang, J.; Gu, G. Attacking the brain: Races in the {SDN} control plane. In Proceedings of the 26th USENIX Security Symposium, Vancouver, BC, Canada, 16–18 August 2017; pp. 451–468.
28. Marin, E.; Bucciol, N.; Conti, M. An in-depth look into sdn topology discovery mechanisms: Novel attacks and practical countermeasures. In Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, London, UK, 11–15 November 2019; pp. 1101–1114.
29. Goldreich, O. *Foundations of Cryptography: Volume 2, Basic Applications*; Cambridge University Press: Cambridge, UK, 2009.
30. Bogdanov, D.; Kamm, L.; Laur, S.; Sokk, V. Rmind: A tool for cryptographically secure statistical analysis. *IEEE Trans. Dependable Secur. Comput.* **2016**, *15*, 481–495. [CrossRef]
31. Burkhart, M.; Strasser, M.; Many, D.; Dimitropoulos, X.A. SEPIA: Privacy-Preserving Aggregation of Multi-Domain Network Events and Statistics. In Proceedings of the 19th USENIX Security Symposium, Washington, DC, USA, 11–13 August 2010; pp. 223–240.
32. Corrigan-Gibbs, H.; Boneh, D. Prio: Private, robust, and scalable computation of aggregate statistics. In Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation, Boston, MA, USA, 27–29 March 2017; pp. 259–282.
33. Xu, Y.; Cui, W.; Peinado, M. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In Proceedings of the 2015 IEEE Symposium on Security and Privacy, San Jose, CA, USA, 17–21 May 2015; pp. 640–656.
34. Fuller, B.; Varia, M.; Yerukhimovich, A.; Shen, E.; Hamlin, A.; Gadepally, V.; Shay, R.; Mitchell, J.D.; Cunningham, R.K. Sok: Cryptographically protected database search. In Proceedings of the 2017 IEEE Symposium on Security and Privacy (SP), San Jose, CA, USA, 22–26 May 2017; pp. 172–191.

35. Liu, Z.; Manousis, A.; Vorsanger, G.; Sekar, V.; Braverman, V. One sketch to rule them all: Rethinking network flow monitoring with univmon. In Proceedings of the 2016 ACM SIGCOMM Conference, Florianopolis, Brazil, 22–26 August 2016; pp. 101–114.
36. Huang, Q.; Jin, X.; Lee, P.P.C.; Li, R.; Tang, L.; Chen, Y.C.; Zhang, G. SketchVisor: Robust Network Measurement for Software Packet Processing. In Proceedings of the Conference of the ACM Special Interest Group on Data Communication, Los Angeles, CA, USA, 21–25 August 2017; pp. 113–126.
37. Yang, T.; Jiang, J.; Liu, P.; Huang, Q.; Gong, J.; Zhou, Y.; Miao, R.; Li, X.; Uhlig, S. Elastic sketch: Adaptive and fast network-wide measurements. In Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication, Budapest, Hungary, 20–25 August 2018; pp. 561–575.
38. Goldreich, O.; Ostrovsky, R. Software protection and simulation on oblivious RAMs. *J. ACM (JACM)* **1996**, *43*, 431–473. [CrossRef]
39. Stefanov, E.; Van Dijk, M.; Shi, E.; Fletcher, C.; Ren, L.; Yu, X.; Devadas, S. Path ORAM: An extremely simple oblivious RAM protocol. In Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security, Berlin, Germany, 4–8 November 2013; pp. 299–310.
40. Liu, C.; Huang, Y.; Shi, E.; Katz, J.; Hicks, M. Automating efficient RAM-model secure computation. In Proceedings of the 2014 IEEE Symposium on Security and Privacy, Berkeley, CA, USA, 18–21 May 2014; pp. 623–638.
41. Wang, X.S.; Huang, Y.; Chan, T.H.; Shelat, A.; Shi, E. SCORAM: Oblivious RAM for secure computation. In Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, Scottsdale, AZ, USA, 3–7 November 2014; pp. 191–202.
42. Costan, V.; Devadas, S. Intel SGX Explained. *IACR Cryptol. EPrint Arch.* **2016**, *2016*, 1–118.
43. Costan, V.; Lebedev, I.A.; Devadas, S. Secure Processors Part I: Background, Taxonomy for Secure Enclaves and Intel SGX Architecture. *Found. Trends Electron. Des. Autom.* **2017**, *11*, 1–248. [CrossRef]
44. Goltzsche, D.; Rüsch, S.; Nieke, M.; Vaucher, S.; Weichbrodt, N.; Schiavoni, V.; Aublin, P.L.; Cosa, P.; Fetzer, C.; Felber, P.; et al. Endbox: Scalable middlebox functions using client-side trusted execution. In Proceedings of the 2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), Luxembourg, 25–28 June 2018; pp. 386–397.
45. Duan, H.; Wang, C.; Yuan, X.; Zhou, Y.; Wang, Q.; Ren, K. LightBox: Full-stack protected stateful middlebox at lightning speed. In Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, London, UK, 11–15 November 2019; pp. 2351–2367.
46. Poddar, R.; Lan, C.; Popa, R.A.; Ratnasamy, S. Safebricks: Shielding network functions in the cloud. In Proceedings of the 15th USENIX Symposium on Networked Systems Design and Implementation, Renton, WA, USA, 9–11 April 2018; pp. 201–216.
47. Han, J.; Kim, S.; Ha, J.; Han, D. Sgx-box: Enabling visibility on encrypted traffic using a secure middlebox module. In Proceedings of the First Asia-Pacific Workshop on Networking, Hong Kong, China, 3–4 August 2017; pp. 99–105.
48. cloud.google.com. Shielded VMs. Available online: https://cloud.google.com/shielded-vm (accessed on 10 January 2021).
49. ibm.com. IBM Cloud Data Shield. Available online: https://www.ibm.com/cloud/data-shield (accessed on 10 January 2021).
50. azure.microsoft.com. Azure Confidential Computing. Available online: https://azure.microsoft.com/en-us/solutions/confidential-compute/ (accessed on 10 January 2021).
51. Lai, S.; Yuan, X.; Liu, J.K.; Yi, X.; Li, Q.; Liu, D.; Nepal, S. OblivSketch: Oblivious Network Measurement as a Cloud Service. In Proceedings of the 28th Annual Network and Distributed System Security Symposium, NDSS 2021, virtually, 21–25 February 2021; The Internet Society: Reston, VA, USA, 2021.
52. caida.org. CAIDA Data Monitors—Active and Passive Data Monitors. Available online: https://www.caida.org/catalog/datasets/monitors/ (accessed on 17 October 2020).
53. Liu, Z.; Ben-Basat, R.; Einziger, G.; Kassner, Y.; Braverman, V.; Friedman, R.; Sekar, V. Nitrosketch: Robust and general sketch-based monitoring in software switches. In Proceedings of the ACM Special Interest Group on Data Communication, Beijing, China, 19–23 August 2019; pp. 334–350.
54. Cormode, G.; Garofalakis, M.N.; Haas, P.J.; Jermaine, C. Synopses for Massive Data: Samples, Histograms, Wavelets, Sketches. *Found. Trends Databases* **2012**, *4*, 1–294. [CrossRef]
55. Liu, L.; Shen, Y.; Yan, Y.; Yang, T.; Shahzad, M.; Cui, B.; Xie, G. SF-Sketch: A Two-Stage Sketch for Data Streams. *IEEE Trans. Parallel Distrib. Syst.* **2020**, *31*, 2263–2276. [CrossRef]
56. Basat, R.B.; Einziger, G.; Feibish, S.L.; Moraney, J.; Raz, D. Network-wide routing-oblivious heavy hitters. In Proceedings of the 2018 Symposium on Architectures for Networking and Communications Systems, Ithaca, NY, USA, 23–24 July 2018; pp. 66–73.
57. Gibbons, P.B.; Matias, Y. New sampling-based summary statistics for improving approximate query answers. In Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data, Seattle, WA, USA, 2–4 June 1998; Volume 27, pp. 331–342.
58. Cormode, G.; Muthukrishnan, S.; Yi, K.; Zhang, Q. Optimal sampling from distributed streams. In Proceedings of the Twenty-Ninth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, Indianapolis, IN, USA, 6–11 June 2010; pp. 77–86.
59. McKeen, F.; Alexandrovich, I.; Berenzon, A.; Rozas, C.V.; Shafi, H.; Shanbhogue, V.; Savagaonkar, U.R. Innovative instructions and software model for isolated execution. In Proceedings of the HASP 2013, The Second Workshop on Hardware and Architectural Support for Security and Privacy, Tel-Aviv, Israel, 23–24 June 2013; Lee, R.B., Shi, W., Eds.; ACM: New York, NY, USA, 2013; p. 10.
60. Intel. Introduction to Intel Advanced Vector Extensions. Available online: https://software.intel.com/content/www/us/en/develop/articles/introduction-to-intel-advanced-vector-extensions.html (accessed on 20 December 2020).

61. Intel. IntrinsicsGuide. Available online: https://software.intel.com/sites/landingpage/IntrinsicsGuide/ (accessed on 20 December 2020).
62. Yu, M.; Jose, L.; Miao, R. Software Defined Traffic Measurement with OpenSketch. In Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation, Lombard, IL, USA, 2–5 April 2013; pp. 29–42.
63. Li, Y.; Miao, R.; Kim, C.; Yu, M. Flowradar: A better netflow for data centers. In Proceedings of the 13th USENIX Symposium on Networked Systems Design and Implementation, Santa Clara, CA, USA, 16–18 March 2016; pp. 311–324.
64. Zhang, F.; Zhang, H. SoK: A study of using hardware-assisted isolated execution environments for security. In Proceedings of the Hardware and Architectural Support for Security and Privacy 2016, Seoul, Korea, 18 June 2016; pp. 1–8.
65. ARM. ARM Security Technology Building a Secure System using TrustZone Technology. Available online: https://developer.arm.com/documentation/genc009492/c (accessed on 20 December 2020).
66. Van Bulck, J.; Weichbrodt, N.; Kapitza, R.; Piessens, F.; Strackx, R. Telling your secrets without page faults: Stealthy page table-based attacks on enclaved execution. In Proceedings of the 26th USENIX Security Symposium, Vancouver, BC, Canada, 16–18 August 2017; pp. 1041–1056.
67. Brasser, F.; Müller, U.; Dmitrienko, A.; Kostiainen, K.; Capkun, S.; Sadeghi, A.R. Software grand exposure: SGX cache attacks are practical. In Proceedings of the 11th USENIX Workshop on Offensive Technologies, Vancouver, BC, Canada, 14–15 August 2017.
68. Lee, S.; Shih, M.W.; Gera, P.; Kim, T.; Kim, H.; Peinado, M. Inferring fine-grained control flow inside SGX enclaves with branch shadowing. In Proceedings of the 26th USENIX Security Symposium, Vancouver, BC, Canada, 16–18 August 2017; pp. 557–574.
69. Seo, J.; Lee, B.; Kim, S.M.; Shih, M.W.; Shin, I.; Han, D.; Kim, T. SGX-Shield: Enabling Address Space Layout Randomization for SGX Programs. In Proceedings of the NDSS 2017, San Diego, CA, USA, 26 February–1 March 2017.
70. Shih, M.W.; Lee, S.; Kim, T.; Peinado, M. T-SGX: Eradicating Controlled-Channel Attacks Against Enclave Programs. In Proceedings of the NDSS 2017, San Diego, CA, USA, 26 February–1 March 2017.
71. Lee, D.; Jung, D.; Fang, I.T.; Tsai, C.C.; Popa, R.A. An off-chip attack on hardware enclaves via the memory bus. In Proceedings of the 29th USENIX Security Symposium, San Diego, CA, USA, 12–14 August 2020.
72. Ahmad, A.; Kim, K.; Sarfaraz, M.I.; Lee, B. OBLIVIATE: A Data Oblivious Filesystem for Intel SGX. In Proceedings of the NDSS 2018, San Diego, CA, USA, 18–21 February 2018.
73. Sasy, S.; Gorbunov, S.; Fletcher, C.W. ZeroTrace: Oblivious Memory Primitives from Intel SGX. In Proceedings of the 2018 Network and Distributed System Security Symposium, San Diego, CA, USA, 18–21 February 2018.
74. Mishra, P.; Poddar, R.; Chen, J.; Chiesa, A.; Popa, R.A. Oblix: An efficient oblivious search index. In Proceedings of the 2018 IEEE Symposium on Security and Privacy (SP), San Francisco, CA, USA, 20–24 May 2018; pp. 279–296.
75. Estan, C.; Varghese, G. New directions in traffic measurement and accounting. In Proceedings of the 2002 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications, Pittsburgh, PA, USA, 19–23 August 2002; pp. 323–336.
76. Pfaff, B.; Pettit, J.; Koponen, T.; Jackson, E.; Zhou, A.; Rajahalme, J.; Gross, J.; Wang, A.; Stringer, J.; Shelar, P.; et al. The design and implementation of open vswitch. In Proceedings of the 12th USENIX Symposium on Networked Systems Design and Implementation, Oakland, CA, USA, 4–6 May 2015; pp. 117–130.
77. Siron, E. What Is the Hyper-V Virtual Switch and How Does It Work? Available online: https://www.altaro.com/hyper-v/the-hyper-v-virtual-switch-explained-part-1/ (accessed on 18 December 2020).
78. Van Bulck, J.; Minkin, M.; Weisse, O.; Genkin, D.; Kasikci, B.; Piessens, F.; Silberstein, M.; Wenisch, T.F.; Yarom, Y.; Strackx, R. Foreshadow: Extracting the keys to the intel SGX kingdom with transient out-of-order execution. In Proceedings of the 27th USENIX Security Symposium, Baltimore, MD, USA, 15–17 August 2018; pp. 991–1008.
79. Murdock, K.; Oswald, D.; Garcia, F.D.; Van Bulck, J.; Gruss, D.; Piessens, F. Plundervolt: Software-based fault injection attacks against Intel SGX. In Proceedings of the 2020 IEEE Symposium on Security and Privacy (SP), San Francisco, CA, USA, 18–21 May 2020; pp. 1466–1482.
80. Gruss, D.; Lipp, M.; Schwarz, M.; Genkin, D.; Juffinger, J.; O'Connell, S.; Schoechl, W.; Yarom, Y. Another flip in the wall of rowhammer defenses. In Proceedings of the 2018 IEEE Symposium on Security and Privacy (SP), San Francisco, CA, USA, 21–23 May 2018; pp. 245–261.
81. Ohrimenko, O.; Schuster, F.; Fournet, C.; Mehta, A.; Nowozin, S.; Vaswani, K.; Costa, M. Oblivious multi-party machine learning on trusted processors. In Proceedings of the SEC'16 Proceedings of the 25th USENIX Conference on Security Symposium, Austin, TX, USA, 10–12 August 2016; pp. 619–636.
82. Liu, L.; Shen, Y. The Source Codes of Our Prototype Service. Available online: https://github.com/paper2021anonymous/fosketch (accessed on 1 May 2020).
83. Batcher, K.E. Sorting networks and their applications. American Federation of Information Processing Societies. In Proceedings of the 1968 Spring Joint Computer Conference, AFIPS Conference, Atlantic City, NJ, USA, 30 April–2 May 1968; Thomson Book Company: Washington, DC, USA, 1968; Volume 32, pp. 307–314.
84. Whang, K.Y.; Vander-Zanden, B.T.; Taylor, H.M. A linear-time probabilistic counting algorithm for database applications. *ACM Trans. Database Syst.* **1990**, *15*, 208–229. [CrossRef]
85. Arnautov, S.; Trach, B.; Gregor, F.; Knauth, T.; Martin, A.; Priebe, C.; Lind, J.; Muthukumaran, D.; O'Keeffe, D.; Stillwell, M.L.; et al. SCONE: Secure Linux containers with Intel SGX. In Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation, Savannah, GA, USA, 2–4 November 2016; pp. 689–703.

86. Orenbach, M.; Lifshits, P.; Minkin, M.; Silberstein, M. Eleos: ExitLess OS Services for SGX Enclaves. In Proceedings of the Twelfth European Conference on Computer Systems, Belgrade, Serbia, 23–26 April 2017; pp. 238–253.
87. 01.org. Intel Software Guard Extensions (Intel SGX) SDK for Linux OS. Available online: https://download.01.org/intel-sgx/sgx-linux/2.9.1/docs/Intel_SGX_Developer_Reference_Linux_2.9.1_Open_Source.pdf (accessed on 1 December 2020).
88. Gueron, S. A Memory Encryption Engine Suitable for General Purpose Processors. *IACR Cryptol. EPrint Arch.* **2016**, *2016*, 204.
89. Lai, S. The Source Codes of OblivSketch. Available online: https://github.com/MonashCybersecurityLab/measurement (accessed on 10 May 2021).
90. Lai, S. The Implementation of PathORAM in OblivSketch Source Code. Available online: https://github.com/MonashCybersecurityLab/measurement/Oblivious/Enclave/ORAM/PathORAM.h (accessed on 10 May 2021).