S^{3}ORAM: A Computation-Efficient and Constant Client Bandwidth Blowup ORAM with Shamir Secret Sharing

Thang Hoang
EECS, Oregon State University
Corvallis, Oregon
hoangmin@oregonstate.edu

Ceyhun D. Ozkaptan
EECS, Oregon State University
Corvallis, Oregon
ozkaptac@oregonstate.edu

Attila A. Yavuz
EECS, Oregon State University
Corvallis, Oregon
attila.yavuz@oregonstate.edu

Jorge Guajardo
Robert Bosch RTC
Pittsburgh, PA
Jorge.GuajardoMerchan@us.bosch.com

Tam Nguyen
EECS, Oregon Sate University
Corvallis, Oregon
nguyeta4@oregonstate.edu

ABSTRACT

Oblivious Random Access Machine (ORAM) enables a client to access her data without leaking her access patterns. Existing client-efficient ORAMs either achieve $O(\log N)$ client-server communication blowup without heavy computation, or $O(1)$ blowup but with expensive homomorphic encryptions. It has been shown that $O(\log N)$ bandwidth blowup might not be practical for certain applications, while schemes with $O(1)$ communication blowup incur even more delay due to costly homomorphic operations.

In this paper, we propose a new distributed ORAM scheme referred to as *Shamir Secret Sharing ORAM (S^{3}ORAM)*, which achieves $O(1)$ client-server bandwidth blowup and $O(1)$ blocks of client storage *without* relying on costly partial homomorphic encryptions. S^{3}ORAM harnesses Shamir Secret Sharing, tree-based ORAM structure and a secure multi-party multiplication protocol to eliminate costly homomorphic operations and, therefore, achieves $O(1)$ client-server bandwidth blowup with a high computational efficiency. We conducted comprehensive experiments to assess the performance of S^{3}ORAM and its counterparts on actual cloud environments, and showed that S^{3}ORAM achieves *three orders of magnitude lower end-to-end delay* compared to alternatives with $O(1)$ client communication blowup (Onion-ORAM), while it is *one order of magnitude faster* than Path-ORAM for a network with a moderate bandwidth quality. We have released the implementation of S^{3}ORAM for further improvement and adaptation.

1 INTRODUCTION

Oblivious Random Access Memory (ORAM) [21] allows Alice to access her data outsourced to a cloud without leaking to the server which data blocks have been accessed. Despite recent progress, it has been shown that existing ORAM designs are costly due to their high communication and/or computation overhead [1, 6, 22, 29, 30].

CCS '17, October 30-November 3, 2017, Dallas, TX, USA

ACM ISBN 978-1-4503-4946-8/17/10…$15.00
https://doi.org/10.1145/3133956.3134090

The objective of this paper is to create an efficient ORAM scheme that simultaneously achieves (i) a low client communication overhead (i.e., $O(1)$ client bandwidth blowup), (2) low computational overhead by avoiding costly partial homomorphic encryptions, and (iii) low client storage (i.e., $O(1)$ block storage).

1.1 Research Gap and Problem Statement

ORAM with $O(\log N)$ bandwidth blowup. Stefanov et al. in [40] proposed Path-ORAM scheme that achieves the optimal lower bound of $\Omega(\log N)$ communication blowup under $O(1)$ blocks of client storage [7, 42]. However, Path-ORAM has been shown to be costly for certain applications [6, 30, 36] due to the transmission cost of $O(\log N)$ blocks per access request. The client communication blowup can be reduced by introducing computation at the server side [3, 12]. Ring-ORAM [33] improved the communication efficiency of Path-ORAM by approximately 2.5 times by allowing the server to perform XOR operations. However, it still requires $O(\log N)$ communication blowup. Other ORAM schemes (e.g., [11, 26]) used single-server PIR (e.g., [41]) to reduce the communication overhead. However, they still require $O(\log N)$ bandwidth blowup, and also incur significant computation cost due to single-server PIR techniques (e.g., [41]).

ORAM with $O(1)$ bandwidth blowup. Recent ORAM schemes (e.g., Onion-ORAM [12], Bucket-ORAM [14], and [3]) rely on fully or partial Homomorphic Encryption (HE) (e.g., [31]) to achieve $O(1)$ bandwidth blowup with $O(1)$ blocks of client storage. However, these ORAMs introduce an extremely high end-to-end delay due to heavy computations incurred by HE operations. For instance, it has been shown in [1, 28] that HE operations take much longer execution time than the use of ORAMs with $O(\log N)$ communication blowup (e.g., Path-ORAM [40]). Therefore, distributed ORAM schemes have been proposed to achieve a better computation performance trade-off.

Distributed ORAM without Costly HE Operations. Stefanov et al. proposed an ORAM scheme [37] that leverages two non-colluding computation-capable servers to achieve $O(1)$ client-server bandwidth blowup with $O(\log N)$ server-server communication blowup. However, it requires $O(\sqrt{N})$ blocks of client storage due to its underlying ORAM primitive (i.e., Partition-ORAM [39]), which is extremely costly for memory-limited clients.

Table 1: Asymptotic and Experimental Performance Comparison of S^{3}ORAM and its Counterparts.

Scheme	Bandwidth Blowup†		Server Computation	Client Block Storage‡	End-to-end Delay (s)	# of servers
	Client-server	Server-server				
Path-ORAM [40]	$O(\log N)$	-	-	$O(\lambda)$	20.3	1
Ring-ORAM [33]	$O(\log N)$	-	XOR	$O(\lambda)$	13.2	1
Onion-ORAM [12]	$O(1)$	-	Additively HE [10]	$O(1)$	10^4	1
S^{3}ORAM	$O(1)$	$O(\log N)$	Secure addition and multiplication of SSS values	$O(1)$	**2.5**	3

This table presents the performance result of selected ORAM schemes with 40GB database containing 128-KB blocks and the network setting that offers a download and upload throughput of approximately 27 and 6 Mbps, respectively. We refer the reader to Section 5 for the details of our experiments. λ denotes the security parameter.
†*Bandwidth blowup* is defined as the ratio between the communication introduced by ORAM and the base case where the access pattern is not hidden.
‡ *Client block storage* is defined as the number of data blocks that need to be temporarily stored at the client. This is equivalent to the stash component used in [33, 40].

Some ORAM schemes (e.g., [1, 27]) attempted to use multi-server PIR (e.g., [9]) to decrease the communication overhead under $O(1)$ blocks of client storage without using costly homomorphic encryption. Abraham et al. in [1] showed an asymptotically tight sub-logarithmic communication blowup bound of $\Omega(\log_{cD} N)$ for composing ORAM with PIR, where c, D are the numbers of blocks stored by the client and performed by PIR operations, respectively. Therefore, although the CHf-ORAM scheme in [27] claimed to achieve $O(1)$ bandwidth blowup with $O(1)$ blocks of client storage, it has been shown to violate this bound with two concrete attacks in [1]. To the best of our knowledge, there is no secure distributed ORAM scheme that can achieve an $O(1)$ client-server bandwidth blowup with $O(1)$ blocks of client storage overhead.

1.2 Our Contribution

We developed a new distributed ORAM that we refer to as *Shamir Secret Shared ORAM (S^{3}ORAM)*. Below, we first present our main idea followed by desirable properties and contributions of our scheme.

Main idea. Although Onion-ORAM [12] is considered a theoretical construction due to its costly partial HE operations, it offers an elegant eviction strategy, which is useful to achieve $O(1)$ client bandwidth blowup with $O(1)$ blocks of client storage. Our main idea is to harness the "homomorphic" properties of Shamir Secret Sharing (SSS) along with a secure multi-party multiplication protocol to perform eviction operations in the line of Onion-ORAM, but in a significantly more efficient and practical manner. By doing this, S^{3}ORAM inherits all desirable properties of Onion-ORAM *without* the costly homomorphic operations and, thus, requiring only a lightweight computation and suitability for larger block sizes. Table 1 outlines a high-level comparison of S^{3}ORAM and its counterparts.

Desirable properties and contributions. We summarize the desirable properties of S^{3}ORAM and our contributions as follows:

- *Low client-server communication:* S^{3}ORAM achieves $O(1)$ client bandwidth blowup, compared with $O(\log N)$ of Path-ORAM [40] and Ring-ORAM [33] (with a fixed number of servers). Moreover, S^{3}ORAM features smaller block sizes (i.e., $\Omega(\log N)$) than those of other ORAM schemes with $O(1)$ communication blowup, which require fully or partial HE operations (e.g., $\Omega(\log^5 N)$ in Onion-ORAM [12], $\Omega(\log^6 N)$ in Bucket-ORAM [14]).
- *Low server computation:* In S^{3}ORAM, servers only perform lightweight modular additions and multiplications, which are much more efficient than partial HE (e.g., [10]) operations. In particular, we show in Section 5 that, the server computation of S^{3}ORAM is three orders of magnitude faster than that of Onion-ORAM.
- *Low client computation:* In S^{3}ORAM, the client only performs lightweight computations for retrieval and eviction operations. Thus, it is more efficient than Onion-ORAM which requires a number of partial HE operations. For example, S^{3}ORAM requires only a few milliseconds compared to minutes of Onion-ORAM to generate encrypted select queries (see Section 5). Moreover, since blocks in S^{3}ORAM are single encrypted, the decryption is less costly and, therefore, faster than other ORAMs (e.g., [12, 37]) whose blocks are onion-encrypted.
- *Low end-to-end delay:* S^{3}ORAM is approximately three orders of magnitude faster than Onion-ORAM, while it is one order of magnitude faster than Path-ORAM in networks with *moderate* bandwidth (e.g., < 240 Mbps).
- *Compact client storage:* S^{3}ORAM features $O(1)$ blocks of client storage, compared to $O(\lambda)$ in Path-ORAM and Ring-ORAM, respectively, and $O(\sqrt{N})$ of Stefanov et al. in [39].
- *High security*: S^{3}ORAM relies on Shamir Secret Sharing and a secure multi-party multiplication protocol, and therefore, it offers information-theoretic security.
- *Full-fledged implementation and experiments:* We implemented S^{3}ORAM and evaluated its performance in an actual cloud environment (i.e., Amazon EC2). The detailed experiments in Section 5 showed that S^{3}ORAM is efficient in practice, and it can even be deployed on mobile devices with limited computation capacity and network connection. We also release the source code of S^{3}ORAM for public use and wide adaptation[1].

S^{3}ORAM does not rely on the direct composition of PIR and ORAM, and it requires servers to communicate with each other to execute a secure multi-party multiplication protocol with the communication blowup of $O(\log N)$. Therefore, S^{3}ORAM does not violate the asymptotic communication bound of Abraham et al. [1]. Note that a high bandwidth is available for inter-cloud communications via dedicated connections [23]. Hence, the inter-cloud communication

[1] Available at https://github.com/thanghoang/S3ORAM

of S^{3}ORAM has a minimal impact on the end-to-end client-server delay as detailed in Section 5.

On the other hand, in many practical scenarios, it may not be possible to guarantee a reliable high-bandwidth communication link between the client and the servers. This is particularly true in the case of home networks and mobile devices with wireless network connectivity (e.g., Wi-Fi, LTE). Therefore, following recent work, one of the main goals of this work is to minimize the client communication overhead while at the same time requiring a low computational overhead at the client and the server sides. In Section 5, we demonstrate the advantages of S^{3}ORAM over its counterparts with $O(\log N)$ and $O(1)$ communication blowup for such moderate bandwidth network settings. It turns out, that the advantages of S^{3}ORAM over its counterparts with $O(1)$ bandwidth blowup such as Onion-ORAM are significant due to its computational efficiency. This efficiency is obtained, however, at the cost of requiring multiple servers (at least three) in the distributed setting.

2 PRELIMINARIES AND BUILDING BLOCKS

Notation. $x \xleftarrow{\$} \mathcal{S}$ denotes that x is randomly and uniformly selected from set $\mathcal{S}$. $|\mathcal{S}|$ denotes the cardinality of set $\mathcal{S}$. $|x|$ denotes the size of variable x. For any integer l, $(x_1, \ldots, x_l) \xleftarrow{\$} \mathcal{S}$ denotes $(x_1 \xleftarrow{\$} \mathcal{S}, \ldots, x_l \xleftarrow{\$} \mathcal{S})$. We denote a finite field as $\mathbb{F}_p$, where p is a prime. Given $\mathbf{u}$ and $\mathbf{v}$ as vectors with the same length, $\mathbf{u} \cdot \mathbf{v}$ denotes the inner product of $\mathbf{u}$ and $\mathbf{v}$. Given an n-dimensional vector $\mathbf{u}$ and a matrix $\mathbf{I}$ of size $n \times m$, $\mathbf{v} = \mathbf{u} \cdot \mathbf{I}$ denotes the matrix product of $\mathbf{u}$ and $\mathbf{I}$ resulting in an m-dimensional vector $\mathbf{v}$. $\mathbf{I}[i, *]$ denotes accessing data of row i of matrix $\mathbf{I}$.

2.1 Model of Computation

Following the literature in distributed secure computation (e.g., [5, 19]), we assume a synchronous network which consists of a client and $\ell \geq 2t + 1$ semi-honest servers $\mathcal{S} = \{\mathcal{S}_1, \ldots, \mathcal{S}_\ell\}$. It is also assumed that the channels between all the players are pairwise-secure, i.e., no player can tamper, read, or modify the contents of the communication channel of other players. We assume that all parties behave in an "honest-but-curious" manner in which parties always send messages as expected but try to learn as much as possible from the shared information received or observed. Notice that in this paper, we *do not* allow parties to provide malicious inputs, i.e., parties are not allowed to behave in a Byzantine manner.

A protocol is t-private [5] (see [19] for similar definitions in the context of distributed PIR) if any set of at most t parties cannot compute after an execution of a protocol more than they could compute individually from their set of private inputs and outputs. Alternatively, the parties have not "learned" anything. Our protocols in general, offer information-theoretic guarantees, unless something is said explicitly to the contrary. This implies that our solutions are secure against computationally unbounded adversaries. As it is standard, we require that all computations by the servers and client be polynomial time and efficient. Finally notice that in this paper, not only is the interaction between the servers and client performed in such a way that information-theoretic security is guaranteed but also the database being accessed is shared among the servers in a way that no coalition of up to t servers can find anything about the database contents (also in an information-theoretic manner).

2.2 Shamir Secret Sharing

We recall (t, ℓ)-threshold Shamir's Secret Sharing scheme [34] which comprises two algorithms SSS.Create and SSS.Recover as presented in Algorithm 1. To share a secret $\alpha \in \mathbb{F}_p$ among ℓ parties, a dealer generates a random polynomial f where $f(0) = \alpha$ and evaluates $f(x_i)$ for party $\mathcal{P}_i$ for $1 \leq i \leq \ell$, where $x_i \in \mathbb{F}_p \setminus \{0\}$ is a deterministic non-zero element of $\mathbb{F}_p$ that uniquely identifies party $\mathcal{P}_i$ and it is considered public information (SSS.Create algorithm). $f(x_i)$ is referred to as the share of party $\mathcal{P}_i$, and it is denoted by $[\![\alpha]\!]_i$. To reconstruct the secret α, the shares of at least $t + 1$ parties have to be combined via Lagrange interpolation (SSS.Recover algorithm).

Algorithm 1 Shamir Secret Sharing (SSS) scheme [34]

$([\![\alpha]\!]_1, \ldots, [\![\alpha]\!]_\ell) \leftarrow$ SSS.Create(α, t): Create t-private shares of α

1: $(a_1, \ldots, a_t) \xleftarrow{\$} \mathbb{F}_p$
2: **for** $i = 1, \ldots, \ell$ **do**
3: $\quad [\![\alpha]\!]_i \leftarrow \alpha + \sum_{j=1}^{t} a_j \cdot x_i^j$
4: **return** $([\![\alpha]\!]_1, \ldots, [\![\alpha]\!]_\ell)$

$\alpha \leftarrow$ SSS.Recover$(\mathcal{A}, t)$: Recover the value α from $\geq t + 1$ shares

1: Randomly pick $t + 1 \leq \ell$ shares $\{[\![\alpha]\!]_{x_1}, \ldots, [\![\alpha]\!]_{x_{t+1}}\}$ in $\mathcal{A}$
2: $g(x) \leftarrow \text{LagrangeInterpolation}\left(\{(x_i, [\![\alpha]\!]_{x_i})\}_{i=1}^{t+1}\right)$
3: $\alpha \leftarrow g(0)$
4: **return** α

We extend the notion of secret share for a value into the share for a vector in the natural way as follows: Given a vector $\mathbf{v} = (v_1, \ldots, v_n)$, $[\![\mathbf{v}]\!]_i = ([\![v_1]\!]_i, \ldots, [\![v_n]\!]_i)$ indicates the share of $\mathbf{v}$ for party $\mathcal{P}_i$, which is a vector whose elements are the shares of the elements in $\mathbf{v}$. Similarly, given a matrix $\mathbf{I}$, $[\![\mathbf{I}]\!]$ denotes the share of $\mathbf{I}$, which is also a matrix with each cell $[\![\mathbf{I}[i, j]]\!]$ being the share of the cell $\mathbf{I}[i, j]$. In some cases, to ease readability, we drop the subscript i, when the party is understood from the context.

Shamir [34] showed that SSS is information-theoretic secure and t-private in the sense that no set of t or less shares reveals any information about the secret. More precisely, $\forall m, m' \in \mathbb{F}_p$, $\forall \mathcal{I} \subseteq \{1, \ldots, \ell\}$ s.t. $|\mathcal{I}| \leq t$ and for any set $\mathcal{A} = \{a_1, \ldots, a_{|\mathcal{I}|}\}$ where $a_i \in \mathbb{F}_p$, the probability distributions of $\left\{s_{i\in\mathcal{I}} : (s_1, \ldots, s_\ell) \leftarrow \text{SSS.Create}(m, t)\right\}$ and $\left\{s'_{i\in\mathcal{I}} : (s'_1, \ldots, s'_\ell) \leftarrow \text{SSS.Create}(m', t)\right\}$ are identical and uniform:

$$\Pr(\{s_{i\in\mathcal{I}}\} = \mathcal{A}) = \Pr(\{s'_{i\in\mathcal{I}}\} = \mathcal{A}).$$

Ben-Or et al. [5] showed that SSS can be used to obtain t-private protocols. Lemma 2.1 summarizes the homomorphic properties of SSS and it was first described in [5].

Lemma 2.1 (SSS homomorphic properties [5]). *Let $[\![\alpha]\!]_i^{(t)}$ be the Shamir share of value $\alpha \in \mathbb{F}_p$ with privacy level t for $\mathcal{P}_i$. SSS offers additively and multiplicatively homomorphic properties:*

- *Addition of two shares*

$$[\![\alpha_1]\!]_i^{(t)} + [\![\alpha_2]\!]_i^{(t)} = [\![\alpha_1 + \alpha_2]\!]_i^{(t)}. \quad (1)$$

- *Multiplication w.r.t a scalar $c \in \mathbb{F}_p$*

$$c \cdot [\![\alpha]\!]_i^{(t)} = [\![c \cdot \alpha]\!]_i^{(t)}. \tag{2}$$

- *Partial share multiplication*

$$[\![\alpha_1]\!]_i^{(t)} \cdot [\![\alpha_2]\!]_i^{(t)} = [\![\alpha_1 \cdot \alpha_2]\!]_i^{(2t)}. \tag{3}$$

The two-share partial multiplication (3) in Lemma 2.1 results in a share of $\alpha_1 \cdot \alpha_2$ which is t-private and represented by a $2t$-degree polynomial. It was first observed in [5] that the resulting polynomial is not uniformly distributed. In order to achieve the uniform distribution and computation consistency over $[\![\alpha_1 \cdot \alpha_2]\!]$, it is required to reduce the degree of the polynomial representation of $[\![\alpha_1 \cdot \alpha_2]\!]$ from $2t$ to t and re-share the polynomial. This multiplication operation with degree reduction can be achieved via a secure multiplication protocol shown in the following section[2].

2.3 Secure Multi-party Multiplication

Gennaro et al. [17] presents a Secure Multiplication Protocol (SMP) for two Shamir secret-shared values among multiple parties. Given $\alpha_1, \alpha_2 \in \mathbb{F}_p$ shared by (t, ℓ)-threshold SSS as $[\![\alpha_1]\!]_i^{(t)}$ and $[\![\alpha_2]\!]_i^{(t)}$ for $1 \le i \le \ell$ respectively, $2t+1$ parties $\mathcal{P}_i$ among ℓ parties would like to compute the multiplication of α_1, α_2 without revealing the value of α_1 and α_2. The protocol requires a Vandermonde matrix $\mathbf{V}_{\{x_i\}}$ of size $(2t+1) \times (2t+1)$ having the structure as follows:

$$\mathbf{V}_{\{x_1,\ldots,x_{2t+1}\}} = \begin{bmatrix} x_1^0 & x_1^1 & \ldots & x_1^{2t} \\ x_2^0 & x_2^1 & \ldots & x_2^{2t} \\ \vdots & \vdots & \ddots & \vdots \\ x_{2t+1}^0 & x_{2t+1}^1 & \ldots & x_{2t+1}^{2t} \end{bmatrix}, \tag{4}$$

where $x_i \in \mathbb{F}_p$ are unique identifiers of participating $\mathcal{P}_i$. We refer to $\mathbf{V}^{-1}$ as the inverse of Vandermonde matrix. Each $\mathcal{P}_i$ locally multiplies $[\![\alpha_1]\!]_i^{(t)}$ and $[\![\alpha_2]\!]_i^{(t)}$ resulting in $[\![\alpha_1 \cdot \alpha_2]\!]_i^{(2t)}$, and creates shares of $[\![\alpha_1 \cdot \alpha_2]\!]_i^{(2t)}$ by a new random polynomial of degree t for $2t+1$ parties and distributes them to other $2t$ parties. Finally, each party locally performs the inner product between the received shares and $\mathbf{V}^{-1}_{\{x_i\}}[1, *]$ to obtain a new share of $\alpha_1 \cdot \alpha_2$ which is now represented by a polynomial of degree t as $[\![\alpha_1 \cdot \alpha_2]\!]_i^{(t)}$. Protocol 1 presents this multiplication protocol.

Protocol 1 SMP Protocol [17]

Input: $\mathcal{P}_i$ owns $[\![\alpha_1]\!]_i^{(t)}$, $[\![\alpha_2]\!]_i^{(t)}$ and wants to compute $[\![\alpha_1 \cdot \alpha_2]\!]_i^{(t)}$

Output: Each $\mathcal{P}_i$ obtains $[\![\beta]\!]_i^{(t)}$, where $\beta = \alpha_1 \cdot \alpha_2$

1: **for** each $\mathcal{P}_i \in \{\mathcal{P}_1, \ldots, \mathcal{P}_{2t+1}\}$ **do**
2: $\quad [\![\beta]\!]_i^{(2t)} \leftarrow [\![\alpha_1]\!]_i^{(t)} \cdot [\![\alpha_2]\!]_i^{(t)}$
3: $\quad ([\![\beta]\!]_j^{(t)})_{j=1}^{\ell} \leftarrow \mathsf{SSS.Create}([\![\beta]\!]_i^{(2t)}, t)$
4: $\quad$ Distribute $[\![\beta]\!]_j^{(t)}$ to all $\mathcal{P}_j \in \{\mathcal{P}_1, \ldots, \mathcal{P}_{2t+1}\} \setminus \mathcal{P}_i$
5: **for** each $\mathcal{P}_i \in \{\mathcal{P}_1, \ldots, \mathcal{P}_{2t+1}\}$ **do**
6: $\quad [\![\beta]\!]_i^{(t)} \leftarrow \sum_{j=1}^{2t+1} \mathbf{V}^{-1}[1, j] \cdot [\![\beta]\!]_j^{(t)}$

[2]Benor et al. [5] proposed a secure multiplication protocol, however the protocol of Gennaro et al. [17] is more efficient and thus, is the subject of Section 2.3.

Lemma 2.2 (SMP protocol Privacy [17]). *The SMP protocol in [17] (denoted as $\star$ operator) offers homomorphic property for full multiplication between two SSS-shares whose result is t-private as:*

$$[\![\alpha_1 \cdot \alpha_2]\!]_i^{(t)} = [\![\alpha_1]\!]_i^{(t)} \star [\![\alpha_2]\!]_i^{(t)} \tag{5}$$

2.4 Multi-server Private Information Retrieval

Private Information Retrieval (PIR) enables retrieval of a data item from an (unencrypted) public database without revealing which item being fetched. We follow the presentation of [4, 19] as follows.

Definition 2.3 (multi-server PIR [4, 9, 19]). Let $\mathbf{b} = (b_1, \ldots, b_n)$ be a database consisting of n items being stored in ℓ servers. A multi-server PIR protocol consists of three algorithms: PIR.CreateQuery, PIR.Retrieve and PIR.Reconstruct. Given an item b_i in $\mathbf{b}$ to be retrieved, the client creates queries $(e_1, \ldots, e_\ell) \leftarrow \mathsf{PIR.CreateQuery}(i)$ and distributes e_j to server $\mathcal{S}_j$. Each server responds with an answer $a_j \leftarrow \mathsf{PIR.Retrieve}(e_j, \mathbf{b})$. Upon receiving ℓ answers, the client computes the value of item b_i by invoking the reconstruction algorithm $b \leftarrow \mathsf{PIR.Reconstruct}(a_1, \ldots, a_\ell)$.

Security of the protocol is defined in terms of *correctness* and *privacy*. A multi-server PIR protocol is *correct* if the client computes the correct value of b from any ℓ answers via PIR.Reconstruct algorithm with probability 1. The concept of t-privacy for protocols is applied naturally to the PIR setting and follows directly from the t-privacy of SSS and the fact that among the servers they only have access to t shares of the query vector [19].

2.5 Multi-server ORAM Security

We now define the security of multi-server ORAM in the semi-honest setting proposed in [1] as a straightforward extension of the definition in [1] to the multi-server setting.

Definition 2.4 (Multi-server ORAM with server computation). Let $\mathbf{x} = \big((\mathsf{op}_1, \mathsf{id}_1, \mathsf{data}_1), \ldots, (\mathsf{op}_q, \mathsf{id}_q, \mathsf{data}_q)\big)$ be a data request sequence of length q, where $\mathsf{op}_j \in \{\mathsf{Read}, \mathsf{Write}\}$, id_j is the identifier to be read/written and data_j is the data identified by id_j to be read/written. Let $ORAM_j(\mathbf{x})$ represent the ORAM client's sequence of interactions with the server $\mathcal{S}_i$ given a data request sequence $\mathbf{x}$.
Correctness. A multi-server ORAM is correct if for any access sequence $\mathbf{x}$, $\big\{ORAM_1(\mathbf{x}), \ldots, ORAM_\ell(\mathbf{x})\big\}$ returns data consistent with $\mathbf{x}$ except with a negligible probability.
t-security. A multi-server ORAM is t-secure if $\forall \mathcal{I} \subseteq \{1, \ldots, \ell\}$ such that $|\mathcal{I}| \le t$, for any two data access sequences $\mathbf{x}, \mathbf{y}$ with $|\mathbf{x}| = |\mathbf{y}|$, their corresponding transcripts $\big\{ORAM_{i \in \mathcal{I}}(\mathbf{x})\big\}$ and $\big\{ORAM_{i \in \mathcal{I}}(\mathbf{y})\big\}$ observed by a coalition of up to t servers $\{\mathcal{S}_{i \in \mathcal{I}}\}$ are statistically/computationally indistinguishable.

3 THE PROPOSED S^{3}ORAM SCHEME

S^{3}ORAM follows the typical procedure of tree-based ORAMs [35]. Specifically, given a block to be accessed, the client first retrieves it from the outsourced ORAM structure via a secure retrieval operation. The retrieved block is then assigned to a random path, and written back to the root bucket. Finally, an eviction operation is performed in order to percolate data blocks to lower levels in the ORAM structure. *The intuition behind S^{3}ORAM access protocol is as follows:* (1) to integrate SSS with a multi-server PIR protocol

Table 2: Notations.

Symbol	Description
$\mathsf{T}, \mathsf{T}[i]$	S^{3}ORAM tree structure and the bucket indexing i.
B, b, c	Block size, block and block chunk, resp.
N, m	Number of blocks and number of chunks in a block.
H	Height of the S^{3}ORAM tree T.
A	Eviction frequency.
Z	Bucket size.
pm	Position map.
$(\mathsf{pID}, \mathsf{pIdx}) \leftarrow \mathsf{pm}[\mathsf{id}]$	Precise location (path ID & path index) of a block id.
$\mathcal{I} \leftarrow \mathcal{P}(\mathsf{pID})$	(Ordered) indexes of buckets residing in path pID.
$i \leftarrow \mathcal{P}(\mathsf{pID}, h)$	Index of the bucket on path pID at level-h.
n_e, n_r	Current number of eviction and retrieval operations, resp.

to perform a private retrieval operation with some homomorphic properties; and (2) to leverage these homomorphic properties of SSS and a SMP protocol to perform block permutation and to preserve t-privacy level of ORAM structure in the eviction phase, without relying on costly partial HE operations. In Table 2, we outline the notation used in the S^{3}ORAM scheme and throughout the rest of the paper.

3.1 S^{3}ORAM Data Structure

The S^{3}ORAM structure is a balanced binary tree denoted as T with a height of H. T can store up to $N \leq A \cdot 2^{H-1}$ data blocks b_i, where constant A is the eviction frequency. A node in T is a *bucket* with Z slots, which can store up to Z real blocks. We index buckets in T according to the top-to-bottom, left-to-right order. Hence, leaf buckets are indexed in $[2^H, 2^{H+1})$ as exemplified in Figure 1. $\mathsf{T}[i]$ and $\mathsf{T}[i, j]$ denote an access operation to the bucket with index i, and to the slot j ($1 \leq j \leq Z$) of the i-th bucket, respectively. S^{3}ORAM has a position map $\mathsf{pm} := (\mathsf{id}, \langle \mathsf{pID}, \mathsf{pIdx} \rangle)$ to store the position of real blocks in T, where $2^H \leq \mathsf{pID} < 2^{H+1}$ denotes the path assigned for the block id, and $1 \leq \mathsf{pIdx} \leq Z \cdot (H+1)$ denotes the block's index in the path pID. We present the construction of S^{3}ORAM data structure in Algorithm 3. Given a database DB organized into N B-bit blocks and an ORAM tree T as the input, the

Algorithm 3 $(\llbracket \mathsf{T} \rrbracket_1, \ldots, \llbracket \mathsf{T} \rrbracket_\ell) \leftarrow$ S^{3}ORAM.Setup(DB, T)

1: Organize DB into blocks $(b_1, \ldots, b_N)$ with IDs $(\mathsf{id}_1, \ldots, \mathsf{id}_N)$
2: $\mathsf{T}[i, j] \leftarrow \{0\}^B$ for $1 \leq i < 2^{H+1}, 1 \leq j \leq Z$
3: $n_e \leftarrow 0, n_r \leftarrow 0$
4: **for** $i = 1 \ldots, N$ **do**
5: $\quad z_i \xleftarrow{\$} [2^H, \ldots, 2^{H+1})$
6: $\quad$ Select (x_i, y_i) s.t. $x_i \in \mathcal{P}(z_i)$ and $\mathsf{T}[x_i, y_i]$ is empty
7: $\quad \mathsf{T}[x_i, y_i] \leftarrow b_i$
8: $\quad \mathsf{pm}[\mathsf{id}_i] \leftarrow (z_i, \lfloor \log_2 x_i \rfloor \cdot Z + y_i)$
9: **for** $i = 1, \ldots, 2^{H+1} - 1$ **do**
10: $\quad$ **for** $j = 1, \ldots, Z$ **do**
11: $\quad\quad (c_{i,j}^{(1)}, \ldots, c_{i,j}^{(m)}) \leftarrow \mathsf{T}[i, j]$, where $c_{i,j}^{(k)} \in \mathbb{F}_p$
12: $\quad\quad$ **for** $k = 1, \ldots, m$ **do**
13: $\quad\quad\quad (\llbracket c_{i,j}^{(k)} \rrbracket_1, \ldots, \llbracket c_{i,j}^{(k)} \rrbracket_\ell) \leftarrow$ SSS.Create$(c_{i,j}^{(k)}, t)$
14: $\quad\quad \llbracket \mathsf{T}[i, j] \rrbracket_l \leftarrow (\llbracket c_{i,j}^{(1)} \rrbracket_l, \ldots, \llbracket c_{i,j}^{(m)} \rrbracket_l)$, for all $1 \leq l \leq \ell$
15: **return** $(\llbracket \mathsf{T} \rrbracket_1, \ldots, \llbracket \mathsf{T} \rrbracket_\ell)$ ▷ Send $\llbracket \mathsf{T} \rrbracket_i$ to $\mathcal{S}_i$, for $1 \leq i \leq \ell$

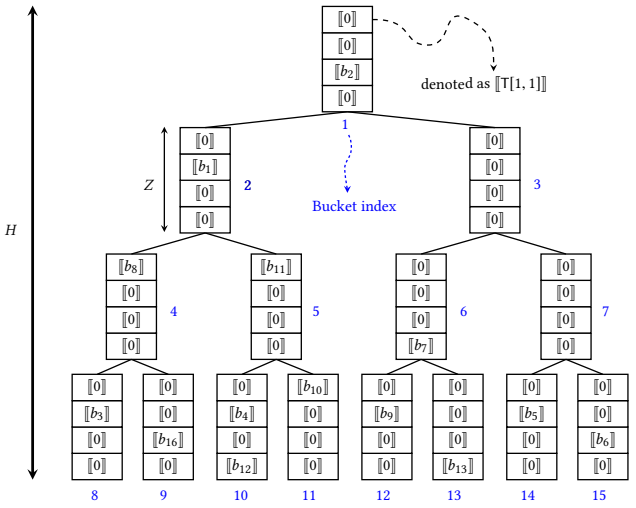


Figure 1: S^{3}ORAM tree structure.

S^{3}ORAM.Setup creates the shares of T as the output for ℓ servers as follows. First, the client initializes every slot in each bucket of T with a 0's string of length B (step 2). The client organizes all data blocks into T, wherein each b_i is independently assigned to a random leaf bucket of T. Notice that B can be larger than $\lceil \log_2 p \rceil$ and therefore, it might not be suitable for arithmetic computation over $\mathbb{F}_p$. To address this, the client splits the data in each slot of T into equal-sized chunks $c_j \in \mathbb{F}_p$ (step 11)[3]. Finally, the client creates shares of T via SSS.Create for each chunk in each slot in T (step 13). The distributed S^{3}ORAM structure consists of ℓ shares of T as $\{\llbracket \mathsf{T} \rrbracket_1, \ldots, \llbracket \mathsf{T} \rrbracket_\ell\}$. Figure 1 outlines the structure of S^{3}ORAM.

3.2 S^{3}ORAM Access Protocol

Protocol 2 $b \leftarrow$ S^{3}ORAM.Access(op, id, b^*)

1: $b \leftarrow$ S^{3}ORAM.Retrieve(id)
2: $\mathsf{pm}[\mathsf{id}].\mathsf{pID} \xleftarrow{\$} [2^H, \ldots, 2^{H+1})$
3: **if** op = write **then**
4: $\quad b \leftarrow b^*$
5: $\mathsf{pm}[\mathsf{id}].\mathsf{pIdx} \leftarrow n_r + 1$
6: $(c_1, \ldots, c_m) \leftarrow b$
7: **for** $j = 1 \ldots, m$ **do**
8: $\quad (\llbracket c_j \rrbracket_1, \ldots, \llbracket c_j \rrbracket_\ell) \leftarrow$ SSS.Create(c_j, t)
9: **for** $i = 1 \ldots, \ell$ **do**
10: $\quad$ Write $(\llbracket c_1 \rrbracket_i, \ldots, \llbracket c_m \rrbracket_i)$ to slot $\llbracket \mathsf{T}[1, n_r + 1] \rrbracket_i$ in server $\mathcal{S}_i$
11: $n_r \leftarrow n_r + 1 \mod A$
12: **if** $n_r = 0$ **then**
13: $\quad$ Execute S^{3}ORAM.SSS-SMP-TripletEviction protocol
14: $\quad n_e \leftarrow n_e + 1 \mod 2^H$
15: **return** b

The S^{3}ORAM access protocol consists of two subroutines including S^{3}ORAM.Retrieve and S^{3}ORAM.SSS-SMP-TripletEviction

[3]We assume implicitly that we choose an appropriate prime p such that every string c_j when interpreted as an element of $\mathbb{F}_p$ is less than p.

as shown in Protocol 2. We first describe a SSS-based select scheme that is used in the S^{3}ORAM.Retrieve subroutine. We then describe our new eviction strategy based on Triplet Eviction [12]. S^{3}ORAM eviction is performed after every A successive accesses as in [12, 33].

• *SSS-based Select Scheme:* Our objective in this select scheme is to privately retrieve a block of interest residing in the queried path from the S^{3}ORAM structure. Recall that in single-server HE-based ORAM schemes (e.g., [3, 12]), the select query is encrypted with costly additive/fully HE. In our case, we "encrypt" S^{3}ORAM structure with SSS that offers highly efficient additive and multiplicative homomorphic properties. We observe that multi-server PIR scheme in [4, 19] relies on SSS to create select queries and, therefore, it can serve as a suitable private retrieval tool to be used for S^{3}ORAM structure. We describe SSS-based select scheme in Algorithm 5, and further outline it as follows:

Algorithm 5 SSS-based Select Scheme

$(\llbracket \mathbf{e} \rrbracket_1, \ldots, \llbracket \mathbf{e} \rrbracket_\ell) \leftarrow$ PIR.CreateQuery(j): Create select queries

1: Let $\mathbf{e} := (e_1, \ldots, e_n)$, where $e_j \leftarrow 1$, $e_i \leftarrow 0$ for $1 \le i \ne j \le n$
2: **for** $i = 1, \ldots, n$ **do**
3: $\quad (\llbracket e_i \rrbracket_1^{(t)}, \ldots, \llbracket e_i \rrbracket_\ell^{(t)}) \leftarrow$ SSS.Create(e_i, t)
4: $\llbracket \mathbf{e} \rrbracket_i^{(t)} := (\llbracket e_1 \rrbracket_i^{(t)}, \ldots, \llbracket e_n \rrbracket_i^{(t)})$, for $1 \le i \le \ell$
5: **return** $(\llbracket \mathbf{e} \rrbracket_1^{(t)}, \ldots, \llbracket \mathbf{e} \rrbracket_\ell^{(t)})$

$\llbracket b \rrbracket_i^{(2t)} \leftarrow$ PIR.Retrieve$(\llbracket \mathbf{e} \rrbracket_i^{(t)}, \llbracket \mathbf{b} \rrbracket_i^{(t)})$: Retrieve the queried block

1: $\llbracket b \rrbracket_i^{(2t)} \leftarrow \llbracket \mathbf{e} \rrbracket_i^{(t)} \cdot \llbracket \mathbf{b} \rrbracket_i^{(t)}$
2: **return** $\llbracket b \rrbracket_i^{(2t)}$

$b \leftarrow$ PIR.Reconstruct$(\llbracket b \rrbracket_1^{(2t)}, \ldots, \llbracket b \rrbracket_\ell^{(2t)})$: Reconstruct the block

1: $b \leftarrow$ SSS.Recover$(\llbracket b \rrbracket_1^{(2t)}, \ldots, \llbracket b \rrbracket_\ell^{(2t)}, 2t)$
2: **return** b

Assume that each server $\mathcal{S}_i$ stores a share of the database $\mathbf{b}$ containing n items denoted as $\llbracket \mathbf{b} \rrbracket_i$, which can be interpreted as a vector with each i-th component being the share of the i-th item in $\mathbf{b}$. Let j be the index of the item in $\mathbf{b}$ to be privately retrieved. The client creates an n-dimensional select vector with all zero coordinates except the j-th coordinate being set to 1 and then, secret-shares it with SSS (PIR.CreateQuery algorithm). The client then distributes these shares to their corresponding servers, each answering with the result of the inner product between the received share vector and its share of $\mathbf{b}$ (PIR.Retrieve algorithm). Finally, the client invokes SSS.Recover function over ℓ answers to recover the desired item (PIR.Reconstruct algorithm). Note that $\mathbf{b}$ in this context is SSS-secret shared, instead of being unencrypted as in [4, 19]. Therefore, our PIR.Reconstruct algorithm requires *at least* $2t+1$ *shares instead of* $t+1$ to recover the item correctly.

S^{3}ORAM Retrieval: We present S^{3}ORAM retrieval protocol in Subroutine 1, which employs three functions of the aforementioned SSS-based select scheme. Given a block to be read, the client first determines its location in the S^{3}ORAM structure via the position map pm and then, retrieves it using SSS-based select protocol. In

Subroutine 1 $b \leftarrow$ S^{3}ORAM.Retrieve(id)

Client:

1: $(s, j) \leftarrow$ pm[id]
2: $(\llbracket \mathbf{e} \rrbracket_1^{(t)}, \ldots, \llbracket \mathbf{e} \rrbracket_\ell^{(t)}) \leftarrow$ PIR.CreateQuery(j)
3: Send $(s, \llbracket \mathbf{e} \rrbracket_i^{(t)})$ to server $\mathcal{S}_i$, for $1 \le i \le \ell$

Server: each $\mathcal{S}_i \in \{\mathcal{S}_1, \ldots, \mathcal{S}_\ell\}$ receiving $(s, \llbracket \mathbf{e} \rrbracket_i^{(t)})$ **do**

4: $\mathcal{I} \leftarrow \mathcal{P}(s)$
5: **for** $j = 1, \ldots, m$ **do**
6: $\quad$ Let $\llbracket \mathbf{c}_j \rrbracket_i^{(t)}$ contain j-th chunk of Z slots in $\llbracket \mathsf{T}[i'] \rrbracket_i^{(t)}, \forall i' \in \mathcal{I}$
7: $\quad \llbracket c_j \rrbracket_i^{(2t)} \leftarrow$ PIR.Retrieve$(\llbracket \mathbf{e} \rrbracket_i^{(t)}, \llbracket \mathbf{c}_j \rrbracket_i^{(t)})$
8: Send $(\llbracket c_1 \rrbracket_i^{(2t)}, \ldots, \llbracket c_m \rrbracket_i^{(2t)})$ to client

Client: On receive $(\{\llbracket c_1 \rrbracket_i^{(2t)}\}_{i=1}^{\ell}, \ldots, \{\llbracket c_m \rrbracket_i^{(2t)}\}_{i=1}^{\ell})$

9: **for** $j = 1, \ldots, m$ **do**
10: $\quad c_j \leftarrow$ PIR.Reconstruct$(\llbracket c_j \rrbracket_1^{(2t)}, \ldots, \llbracket c_j \rrbracket_\ell^{(2t)})$
11: $b \leftarrow (c_1, \ldots, c_m)$
12: **return** b

this case, we interpret all slots in the retrieval path as the database input $\mathbf{b}$ in PIR.Retrieve algorithm. Hence, the size of $\mathbf{b}$ and the length of the query vector is $n = Z \cdot (H+1)$. Note that there are m separate chunks in each slot, the servers need to invoke PIR.Retrieve algorithm m times with the same select query but over different $\mathbf{b}_j$, where each $\mathbf{b}_j$ contains the j-th chunk of all slots in the retrieval path. Finally, the client obtains the desired block by recovering all chunks upon receiving their shares with PIR.Reconstruct algorithm (steps 9–11).

After the block is retrieved, the client creates its new SSS-shares (steps 7–8, Protocol 2), and then writes the share to an empty slot in the root bucket of the corresponding server (step 10). After A successive retrievals, the background eviction is performed as described below.

SSS-SMP-based Triplet Eviction: For each eviction operation, the client selects a deterministic eviction path according to the reverse lexicographical order as proposed in [18]. Given a binary ORAM tree of height H, where edges in each level are indexed by either 0 (left) or 1 (right), the collection of edges of the eviction path at the n_e-th eviction operation is determined as:

$$v = \mathsf{DigitReverse}_2(n_e \mod 2^H), \tag{6}$$

where $\mathsf{DigitReverse}_2(a)$ denotes the order-reversal of base-2 digits of the integer a.

The S^{3}ORAM eviction strategy is inspired on the "Triplet Eviction" strategy of Onion-ORAM [12]. Specifically, it percolates real data blocks in the eviction path to lower levels as much as possible, in which each real block b from each bucket T_i can be moved to one of T_i's children such that b still resides in its own path. This strategy incurs significant cost in Onion-ORAM due to the following reasons: (i) Onion-ORAM relies on additively HE so that after each pushdown operation, a layer of encryption is added to blocks, which increases the cost of select query creation and block decryption in the retrieval phase. (ii) It requires multiple rounds of client-server communication to (ii-1) get the precise location of

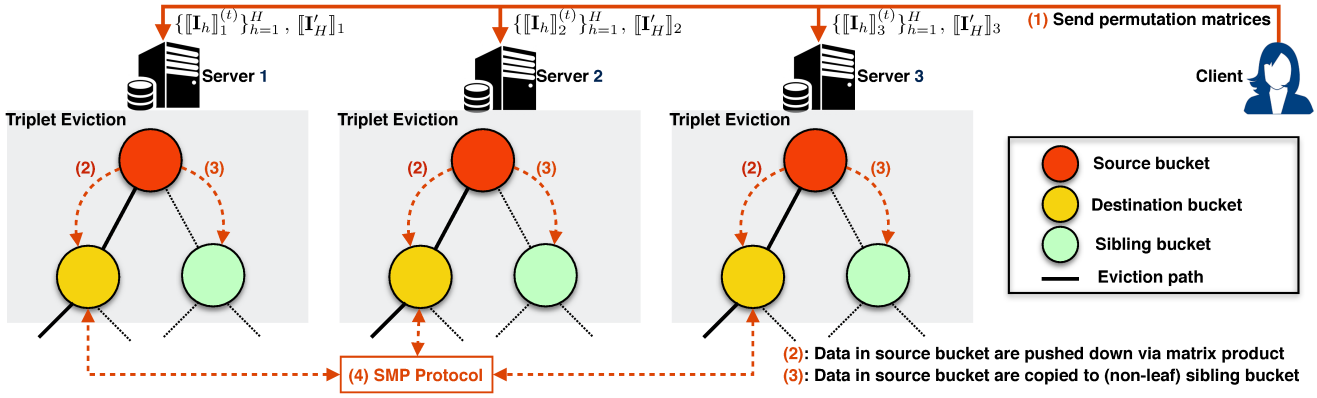


Figure 2: The SSS-SMP-based Triplet Eviction.

Subroutine 3 S^3ORAM.SSS-SMP-TripletEviction

Client:

$v \leftarrow \text{DigitReverse}_2(n_e \mod 2^H)$
Let $\vec{u} = (u_1, \ldots, u_H)$ be the ordered indexes of source buckets on the path indicated by v, starting from the root
for $h = 1, \ldots, H-1$ **do**
  Let $\mathbf{I}_h$ be a $2Z \times Z$ matrix, set $\mathbf{I}_h[*,*] \leftarrow 0$. Let $\hat{u}_h$ be the index of destination bucket of $\mathsf{T}[u_h]$
  for each real block with id in $\mathsf{T}[\hat{u}_h]$ **do**
    $\mathbf{I}_h[j - Z \cdot (h-1), j - Z \cdot h] \leftarrow 1$ where $j \leftarrow \mathsf{pm}[\text{id}].\mathsf{pIdx}$
  for each real block with id′ in $\mathsf{T}[u_h]$ **do**
    $(s', j') \leftarrow \mathsf{pm}[\text{id}']$
    if $\mathcal{P}(s', h) = \hat{u}_h$ **then**
      $\mathbf{I}_h[j' - Z \cdot (h-1), j''] \leftarrow 1$, $\mathsf{pm}[\text{id}'].\mathsf{pIdx} \leftarrow Z \cdot h + j''$, where j'' is the index of an empty slot selected in $\mathsf{T}[\hat{u}_h]$
    else
      $\mathsf{pm}[\text{id}'].\mathsf{pIdx} \leftarrow j' + Z$
Execute steps (3)–(10) with $h = H$ producing $\mathbf{I}_H$ for source-to-destination permutation at leaf level
Execute steps (3)–(10) with $h = H$, $\hat{u}_h$ = index of sibling bucket of $\mathsf{T}[u_h]$ producing $\mathbf{I}'_H$ for source-to-sibling permutation at leaf level
$[\![\mathbf{I}_h[i,j]]\!]_1^{(t)}, \ldots, [\![\mathbf{I}_h[i,j]]\!]_\ell^{(t)} \leftarrow \text{SSS.Create}(\mathbf{I}_h[i,j], t)$ for $1 \le h \le H$, $1 \le i \le 2Z$, $1 \le j \le Z$
$[\![\mathbf{I}'_H[i,j]]\!]_1^{(t)}, \ldots, [\![\mathbf{I}'_H[i,j]]\!]_\ell^{(t)} \leftarrow \text{SSS.Create}(\mathbf{I}'_H[i,j], t)$ for $1 \le i \le 2Z$, $1 \le j \le Z$
Send $\left(v, \{[\![\mathbf{I}_h]\!]_i^{(t)}\}_{h=1}^H, [\![\mathbf{I}'_H]\!]_i^{(t)}\right)$ to $\mathcal{S}_i$, for $1 \le i \le \ell$

Server: each $\mathcal{S}_i \in \{\mathcal{S}_1, \ldots, \mathcal{S}_\ell\}$ receiving $\left(v, \{[\![\mathbf{I}_h]\!]_i^{(t)}\}_{h=1}^H, [\![\mathbf{I}'_H]\!]_i^{(t)}\right)$ **do**

Let $\vec{u} = (u_1, \ldots, u_H)$ be the ordered indexes of source buckets on the path indicated by v, starting from the root
$[\![\mathsf{T}[\hat{u}_h]]\!]_i \leftarrow [\![\mathsf{T}[u_h]]\!]_i$ for $1 \le h < H$ ▷ Replace sibling bucket with source bucket at non-leaf levels
for $h = 1, \ldots, H$ **do**
  Let $\hat{u}_h$ be the index of destination bucket of $\mathsf{T}[u_h]$
  for $j = 1, \ldots, m$ **do**
    $[\![\mathbf{c}'_{h,j}]\!]_i^{(2t)} \leftarrow [\![\mathbf{c}_{h,j}]\!]_i^{(t)} \cdot [\![\mathbf{I}_h]\!]_i^{(t)}$, where $[\![\mathbf{c}_{h,j}]\!]_i^{(t)}$ contains j-th chunks from Z slots of $[\![\mathsf{T}[u_h]]\!]_i^{(t)}$ and $[\![\mathsf{T}[\hat{u}_h]]\!]_i^{(t)}$
    Execute SMP protocol to reduce $[\![\mathbf{c}'_{h,j}]\!]_i^{(2t)}$ to $[\![\mathbf{c}'_{h,j}]\!]_i^{(t)}$, and update j-th chunks in $[\![\mathsf{T}[u_h]]\!]_i^{(t)}$ and $[\![\mathsf{T}[\hat{u}_h]]\!]_i^{(t)}$ with $[\![\mathbf{c}'_{h,j}]\!]_i^{(t)}$
Execute steps (20)–(24) with $h = H$, $[\![\mathbf{I}_h]\!]_i^{(t)} = [\![\mathbf{I}'_H]\!]_i^{(t)}$, $\hat{u}_h$ = index of sibling bucket of $\mathsf{T}[u_h]$

the real blocks for select query creation (since this info is stored in the bucket's metadata) and, (ii-2) bound the number of encryption layers at the leaf buckets. Notice that the latter also requires the client to perform a number of costly homomorphic encryptions and decryptions.

• *Our New Triplet Eviction Strategy:* In this paper, we propose a Triplet Eviction strategy that only requires *single-round client-server communication and lightweight client computation and avoids accumulating multi-layer of encryption to the ORAM structure.* The main idea is to leverage SSS and SMP protocol to perform block permutation and maintain the consistency of privacy level. We present this strategy as follows:

Remark that since the precise location of real blocks is locally stored in the position map, the client is not required to interact with the server(s) to read the metadata. For each level in the eviction path, we obliviously move blocks from source bucket T_i to its children T_{2i}, T_{2i+1}. We follow the terminology used in [12] to denote the buckets involved in each Triplet Eviction operation: if one child of the source bucket resides in the eviction path, it is called *destination* bucket and the other is called *sibling* bucket (see Figure 2 for clarification). Eviction is performed according to the following rules:

• *Source to destination:* Let $[\![\mathbf{u}]\!]$ be a $2Z$-dimensional share vector formed by concatenating all data in the source bucket and the destination bucket. The client creates a permutation matrix $\mathbf{I} \in \{0,1\}^{2Z\times Z}$ such that the matrix product of $[\![\mathbf{u}]\!]$ and $\mathbf{I}$ will result in a Z-dimensional vector $[\![\mathbf{v}]\!]$, in which data at position i in $[\![\mathbf{u}]\!]$ is moved to position j in $[\![\mathbf{v}]\!]$. That is, $\mathbf{I}$ is a matrix, where $\mathbf{I}[i,j] \leftarrow 1$ if the block at position i in $[\![\mathbf{u}]\!]$ is expected to move to position j in $[\![\mathbf{v}]\!]$. As a result, $\mathbf{I}[i+Z,i] \leftarrow 1$ if the block currently at position i in $[\![\mathbf{v}]\!]$ stays at the same location. To hide the location information of real blocks after permutation, the client "encrypts" every single element of $\mathbf{I}$ with SSS resulting in a share matrix $[\![\mathbf{I}]\!] \in \mathbb{F}_p^{2Z\times Z}$. Note that the matrix product between these two shares results in a share vector with each element being represented by a degree-$2t$ polynomial. To maintain the consistency of the S^{3}ORAM structure, servers will together execute the SMP Protocol presented in Section 2 to reduce the degree of polynomial of each component in $[\![\mathbf{v}]\!]$ from $2t$ to t.

• *Source to sibling:* We can apply the same trick as in the source-to-destination operation above to push real blocks down to sibling buckets. However, since non-leaf sibling buckets are guaranteed to be empty by previous evictions featuring a negligible bucket overflow probability (see Lemma 4.1), this process can be further optimized as discussed in [12] as follows. For evictions not involved with leaf buckets, the client simply requests servers to copy all data from the source buckets to sibling buckets and then, update the path index of real blocks in the position map accordingly. For leaf level, it is required to use the matrix permutation as described above since leaf buckets are not empty. This optimization can halve the bandwidth cost of client-server and server-server communication.

Generally, we can see that our eviction approach requires only one client-server communication and guarantees that all data after eviction are consistently "encrypted" by degree-t polynomials. Figure 2 illustrates this new SSS-based Triplet Eviction strategy. We present the algorithmic description of this strategy in Subroutine 3.

3.3 Asymptotic Cost Analysis

In this section, we study the cost of S^{3}ORAM pertaining to block size and number of blocks, where the security level and other system parameters (e.g., prime field, bucket size) are fixed.

• **Communication**: The size of each select query being sent to ℓ servers in S^{3}ORAM retrieval is $(Z\cdot(H+1)\cdot\lceil\log_2 p\rceil)$ bits. The client sends $H+1$ permutation matrices in the S^{3}ORAM eviction to ℓ servers, each being of size $2Z^2\cdot\lceil\log_2 p\rceil$ bits. Each S^{3}ORAM access incurs one block of size B to be transferred between client and server. Since $H \geq \log N/A + 1$ and ℓ, A, Z, p are constants, the overall client-server communication complexity is $O(B+\log N)$. In the eviction, each server distributes the shares of H buckets with size of $Z\cdot B$ bits to each other in H communication rounds. Hence, the inter-server communication overhead is $O(B\cdot\log N)$.

Achieving $O(1)$ client-server bandwidth blowup: The client bandwidth blowup is defined as the *ratio* between the number of client-server communication introduced by ORAM and the base case without ORAM being used. The communication complexity of S^{3}ORAM shows that the size of select vector and permutation matrix is independent of block size. Note that in Onion-ORAM, the select vector size is also independent of the block size. Therefore, $O(1)$ client bandwidth blowup can be achieved in S^{3}ORAM and Onion-ORAM by selecting a suitable block size. That is, by selecting the block size B to be $\Omega(\log N)$, S^{3}ORAM achieves $O(1)$ client bandwidth blowup. In Onion-ORAM, it requires selecting a (larger) block size of $O(\log^5 N)$ to absorb the size of select queries.

• **Computation**: Each server computes the inner product between the $Z\cdot(H+1)$-dimensional select vector and the block vector containing $Z\cdot(H+1)$ blocks of size B. For eviction, each server computes H times the matrix product between a vector containing $2\cdot Z$ blocks of size B and a permutation matrix of size $2Z\times Z$. After that, each server computes the share and performs degree reduction in the SMP protocol on $Z\cdot H$ blocks of size B. In total, the server computation complexity is $O(B\cdot\log N)$.

The client invokes SSS.Create algorithm $Z\cdot(H+1)$ times and $2Z^2\cdot H$ times to create a select query and H permutation matrices, respectively. The client invokes SSS.Recover and SSS.Create algorithms to reconstruct and re-share a block of size B, respectively. Thus, the overall client computation complexity is $O(B+\log N)$.

• **Storage**: S^{3}ORAM tree structure is of height H which has $2^H\cdot Z$ slots and can store up to $N \leq A\cdot 2^{H-1}$ real blocks. Since Z and A are constants, the server storage cost is $O(B\cdot N)$. Notice that the share of the value has the same size as the value (i.e., no ciphertext expansion as in Onion-ORAM), the server storage of S^{3}ORAM is constant and does not increase after a sequence of access operations.

Similar to Onion-ORAM, the block storage in S^{3}ORAM is $O(1)$ since the client immediately writes retrieved block back to the root bucket. The client locally stores the position map whose cost is $O(N(\log N + \log\log N))$.

Achieving O(1) client storage via Recursion: For theoretical interest, S^{3}ORAM can achieve (in total) $O(1)$ client storage by storing the position map in smaller ORAMs using the recursion technique described in [39] and bucket metadata structure in [12]. Specifically, for each S^{3}ORAM bucket, we create a metadata that stores the current index (pIdx) and the assigned path (pID) of blocks residing in it. For each S^{3}ORAM access, the metadata of buckets along the retrieval/eviction path will be read first to get the precise location and the assigned path of blocks of interest. This information will be used to create the select query and permutation matrices. Next, we construct a series of S^{3}ORAM structures S^{3}ORAM$_0,\ldots,$S^{3}ORAM$_X$, where $X = O(\log N)$, S^{3}ORAM$_0$ stores data blocks and S^{3}ORAM$_{i+1}$ stores the position map of S^{3}ORAM$_i$. Note that in this recursion, the position map only stores the blocks' assigned path since their precise location is already maintained in bucket metadata. We refer the reader to [12, 39] for the detailed descriptions.

In S^{3}ORAM$_0$, the bucket metadata is of size $O(\log N)$. Each S^{3}ORAM$_0$ retrieval/eviction accesses the metadata from $O(\log N)$

buckets resulting in $O(\log^2 N)$ client-server bandwidth overhead. Therefore, to achieve $O(1)$ client bandwidth blowup, the block size of S^{3}ORAM$_0$ needs to be $\Omega^2(\log N)$. Since the block size of S^{3}ORAM$_{i+1}$ is smaller than that of S^{3}ORAM$_i$, applying recursion technique to S^{3}ORAM does not increase the bandwidth, computation or server storage overhead in the asymptotic point of view. However, it incurs $O(\log N)$ communication *rounds* and requires $O(\log N)$ factor larger block size, which might significantly increase the end-to-end delay in reality. Thus, it is recommended to maintain the position map locally, given that its size is small enough, to gain performance advantages in practice.

4 SECURITY

The S^{3}ORAM eviction follows the Triplet Eviction proposed in Onion-ORAM [12]. Therefore, they have the same failure probability. We refer the reader to [12] for the details of the proof.

Lemma 4.1 (Bucket Overflow Probability). *If $Z \geq A$ and $N \leq A \cdot 2^H - 1$, the probability that a bucket overflows after an eviction operation is bounded by $e^{-\frac{(2Z-A)^2}{6A}}$, where $Z = A = \Theta(\lambda)$.*

Proof. We refer the reader to [12]. □

The correctness of S^{3}ORAM is shown in Theorem 4.2.

Theorem 4.2 (S^{3}ORAM Correctness). *S^{3}ORAM is correct according to Definition 2.4.*

Proof. See Appendix. □

The security of S^{3}ORAM is given in Theorem 4.3.

Theorem 4.3 (S^{3}ORAM Security). *S^{3}ORAM is unconditionally t-secure according to Definition 2.4.*

Proof. See Appendix. □

Malicious setting: We do not consider malicious servers in this paper. However, since the S^{3}ORAM relies on SSS as its main building block, it is possible to extend the scheme to tolerate malicious inputs. This requires more servers and additional rounds of interaction during the eviction phase due to the more involved security requirements (e.g., verifiable secret sharing [16]) in order to distribute correct shares and detect malicious behaviors. We will investigate the cost for malicious setting in the full version.

5 EXPERIMENTAL EVALUATION

We first describe our implementation details and configuration. We then give our evaluation metrics and methodology, followed by a detailed comparison of S^{3}ORAM and its counterparts on an actual cloud environment with various network and system settings. Note that in this evaluation, we evaluate ORAM schemes under their non-recursive form, where the position map is stored at the client since its size is practically small to be stored locally.

5.1 Implementation Details and Configuration

Software setting. We implemented S^{3}ORAM in C++ with two external libraries: (i) Shoup's NTL library v9.10.0[4] for pseudo-random number generation and modular computations since it offers low-level (e.g., assembly) optimizations for modular multiplication and inner product; (ii) ZeroMQ library[5] for socket programming. Our implementation supports parallelization to take full advantage of multi-core CPUs at the server. We used `libtomcrypt`[6] with AES-CTR to implement IND-CPA encryption for S^{3}ORAM counterparts.

Hardware setting. We conducted our experiments on two types of client devices: (i) A 2015 Macbook Pro laptop as the client, which was equipped with Intel Core i5-5287U CPU @ 2.90GHz and 16 GB RAM. (ii) A Google Nexus 6P smartphone, which ran Android 7.0 Nougat and was equipped with Qualcomm Snapdragon 810 CPU @ 2 GHz and 3 GB RAM. At the server side, we used Amazon EC2 with `c4.4xlarge` type to deploy three server instances. Each server was running Ubuntu 16.04 and equipped with 16 vCPUs Intel Xeon E5-2666 v3 @2.9 GHz, 30 GB RAM and 512 GB SSD.

Network setting. We located three servers to be geographically close to the clients as well as to each other, which results in the network latency between them to be approximately 15 ms. Servers were connected to each other via dedicated networks whose throughput for both download and upload is approximately 250 Mbps.
The laptop client was connected to a home Internet service via Wi-Fi, which offers download and upload throughputs of 29 Mbps and 5 Mbps to the servers, respectively. For mobile client, we used LTE network to communicate with server(s), which has a network latency of 25 ms, and the download and upload throughputs of approximately 27 Mbps and 9 Mbps, respectively.

Database size. We evaluated the performance of all compared schemes with a randomly generated database and block size ranging from 0.5 GB to 40 GB, and from 64 KB to 768 KB, respectively.

5.2 Evaluation Metrics and Methodology

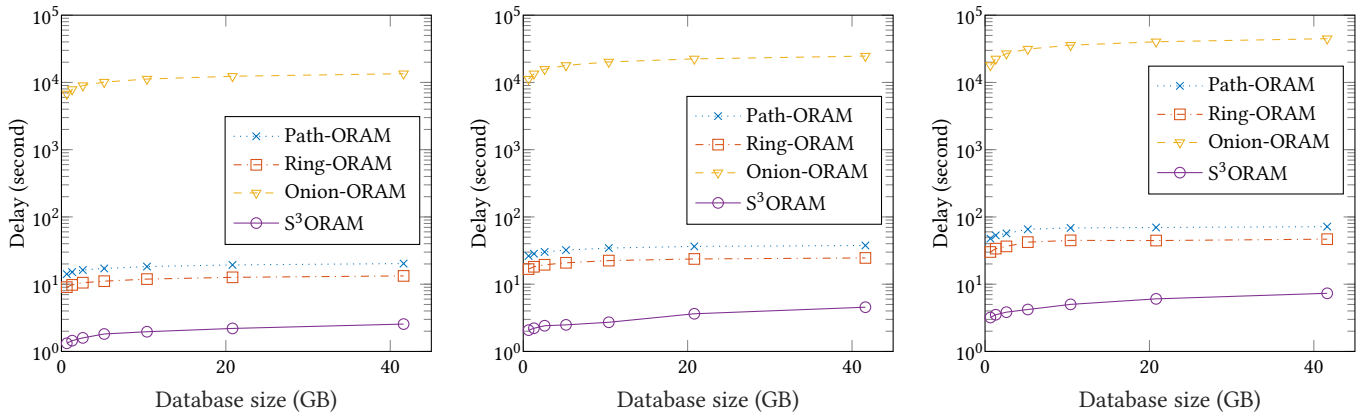
Evaluation Metrics. We compared S^{3}ORAM with its counterparts based on: (1) Building time for ORAM structure (executed once at the beginning); (2) End-to-end delay for different database and block sizes; The cost breakdown of end-to-end delay to assess the impacts of (3) client computation overhead, (4) server computation overhead, (5) client-server communication, (6) server-server communication, (7) disk access time, (8) network bandwidth quality, (9) client and server storage overhead.

We selected Path-ORAM [40] and Onion-ORAM [12] as the main counterparts of S^{3}ORAM , since the former achieves the optimal lower bound of $\Omega(\log N)$ communication blowup without server computation, while the latter achieves $O(1)$ communication blowup with $O(1)$ client storage with server computation. We also chose Ring-ORAM [33] as it is an efficient ORAM scheme with server computation. We did not consider alternatives that (i) failed to achieve $O(1)$ client communication blowup but incurred more delay (e.g., [11, 26]), (ii) were shown to be insecure (e.g., [27, 29]), or (iii) incur significantly more cost than ORAMs considered in our experiments (e.g., [3]) (see Section 6 for related work). We also did not explicitly compare the performance of S^{3}ORAM against the distributed ORAM by Stefanov et al. [37] because of the major difference in terms of client block storage between the two schemes

[4] Available at http://www.shoup.net/ntl/download.html

[5] Available at http://zeromq.org

[6] Available at https://github.com/libtom/libtomcrypt



(a) Block size = 128 KB (b) Block size = 256 KB (c) Block size = 512 KB

Figure 3: End-to-end delay of S^{3}ORAM and its counterparts on a laptop with home network.

($O(1)$ vs. $O(\sqrt{N})$). Given a very large outsourced database, the storage requirement by [37] might not be suitable for resource-limited devices such as mobile phone. Moreover, if $O(\sqrt{N})$ block storage is acceptable, then the lower bound in [1] might imply a better ORAM strategy than S^{3}ORAM that leverages PIR-only technique to achieve $O(1)$ client bandwidth blowup.

Evaluation Methodology. Our methodology is as follows.

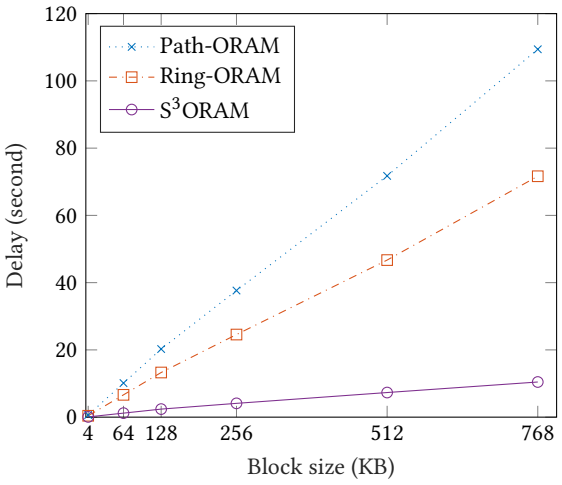
- *S^{3}ORAM*: We fully implemented S^{3}ORAM and measured the delay of each operation (see *cost breakdown* part below). We selected the bucket size and the eviction frequency as $Z = A = 333$ to achieve a negligible bucket overflow probability of 2^{-80}. The cost for each S^{3}ORAM access was measured as the retrieval delay with the write-to-root delay (step 10, Protocol 2) plus the amortized cost of eviction.
- *Path-ORAM*: We measured the delay of Path-ORAM as the time to (1) download/upload $O(\log N)$ blocks, and (2) IND-CPA encryption/decryption at the client. We selected the bucket size as 5 to guarantee a negligible stash overflow probability of 2^{-80}.
- *Ring-ORAM*: We measured the delay of Ring-ORAM as the time to (1) retrieve $O(1)$ block, (2) perform XOR and IND-CPA encryption/decryption at the client, (3) perform XOR operations at the server side. The amortized cost of each Ring-ORAM access is calculated by including the amortized cost of eviction and early shuffles based on the equation $(H + 1)(2Z + S)/A \cdot (1 + \text{PoissCDF}(S, A))$ given in Section 5 of [33]. We selected Ring-ORAM parameters $Z = 16$, $S = 25$ and $A = 20$ as stated in [33] for a negligible stash overflow probability of 2^{-80}.
- *Onion-ORAM*: We measured the delay of Onion-ORAM as the time to (1) perform homomorphic computations at the client and server, (2) transfer $O(1)$ blocks and select queries. We selected the size of RSA modulus to be 1024 bits for AHE according to [2]. Similar to S^{3}ORAM, we selected the bucket size and the eviction frequency of Onion-ORAM as $Z = A = 333$. The cost for each Onion-ORAM access was also included with the amortized cost of eviction operation. Since Onion-ORAM is extremely computationally costly, measuring its delay even on a medium database takes insurmountable amount of time. Therefore, we had to measure its delay on a very small database (i.e., 1 MB) first, and then estimate the delay for larger database sizes.

5.3 Experimental Results

• *Building Time of ORAM Structure (executed only once during offline phase)*: We first provide the total building time for constructing the distributed S^{3}ORAM tree structures. Since S^{3}ORAM relies on highly efficient SSS operations (e.g., basic arithmetics with modular addition/multiplication), it only took around 1 hour to create shares of a large database (i.e., 40 GB) for three servers with the laptop as the client device. This cost is comparable with IND-CPA encryption being used in traditional ORAM schemes. For instance, it took approximately 50-60 minutes to encrypt counterpart ORAM schemes with AES-CTR encryption.

• *End-to-end Delay:* Figure 3 presents the end-to-end delay of S^{3}ORAM and its counterparts. S^{3}ORAM outperformed its counterparts for increasing database and block sizes. As shown in Figure 3b, S^{3}ORAM took around 4.5 seconds to access a 256-KB block, which is approximately 8.3× and 5.4× faster than Path-ORAM and Ring-ORAM, respectively, while being three orders of magnitude faster than Onion-ORAM (40 GB DB).

Figure 3 shows that choosing larger block sizes had a minimal impact on the delay of S^{3}ORAM compared to its counterpart. For instance, S^{3}ORAM took around 2.4 and 7.3 seconds to access a 128-KB block and a 512-KB block, respectively, which corresponds to a linear growth but with a small slope (40 GB DB). Figure 4 further illustrates the influence of block size on the delay of S^{3}ORAM and its counterparts. Although the cost of each ORAM scheme grows linearly with respect to the block size, the slope of S^{3}ORAM is significantly smaller than that of its counterparts. Given any block size in the range from 4 KB to 768 KB, S^{3}ORAM is always approximately 5× and 8× faster than Ring-ORAM and Path-ORAM, respectively. This gives an advantage to S^{3}ORAM over its counterparts for applications with a large block size such as image or video storage services.



*We excluded Onion-ORAM since its plot is far beyond the limit of y-axis.

Figure 4: End-to-end for varying block sizes for a 40GB DB.

Detailed cost breakdown of S^{3}ORAM. We now dissect the end-to-end delay of S^{3}ORAM to investigate which factors contributed the most to the total delay. Figure 5 shows the detailed delay analysis of S^{3}ORAM with three different block sizes on a laptop as a client device with three servers.

• *Server Computation*: The server computation only occupied a small amount (5–8%) of the total delay.

• *Client-server Communication and Disk I/O Access*: The client-server communication and server disk I/O access contributed more than 90% of the total delay. Notice that most of the client-server communication time was spent for retrieving/writing blocks over three servers. The transmission of select vector and permutation matrix only cost approximately 6–7% of the total client-server communication overhead with 512 KB block size. Due to the cache miss issue and the infrastructure of selected Amazon EC2 server instances (i.e., `c4.4xlarge` type), disk I/O access caused a significant delay, especially for large database sizes. Specifically, the maximum RAM of each server was limited to 30GB while each S^{3}ORAM retrieval incurs a random disk I/O access of around 1-2 GB of data due to the large bucket size (i.e., $Z = 333$). As a result, after some random retrieval operations, the cache miss may appear more frequently. Moreover, S^{3}ORAM data structure was stored in a networked storage unit called "Elastic Block Storage" (EBS), which was connected to Amazon EC2 computing unit with a maximum throughput of 160 MB per second. The disk I/O access was also limited by this throughput, and therefore, it is much slower than a local storage setting, in which the read/write throughput of 400–500 MB/s can be achieved. Hence, we expect that the latency of S^{3}ORAM can be further optimized (at least twice) by minimizing the disk I/O access via special server instances offering either local storage (e.g., internal dedicated SSD) or higher throughput (e.g., st1 volume type).

• *Server-server Communication*: The overhead of server-server communication is minimal, since servers are connected with a high-bandwidth network (i.e., 250 Mbps). Moreover, the eviction was only performed after $A = 333$ successive retrievals and, therefore, the overhead of server-server communication was amortized.

• *Client Computation*: The client computation (e.g., block recovery, the creation of shares for blocks, the select vectors and permutation matrices) is negligible and, therefore, is difficult to observe in Figure 5.

Detailed cost breakdown of counterpart schemes.

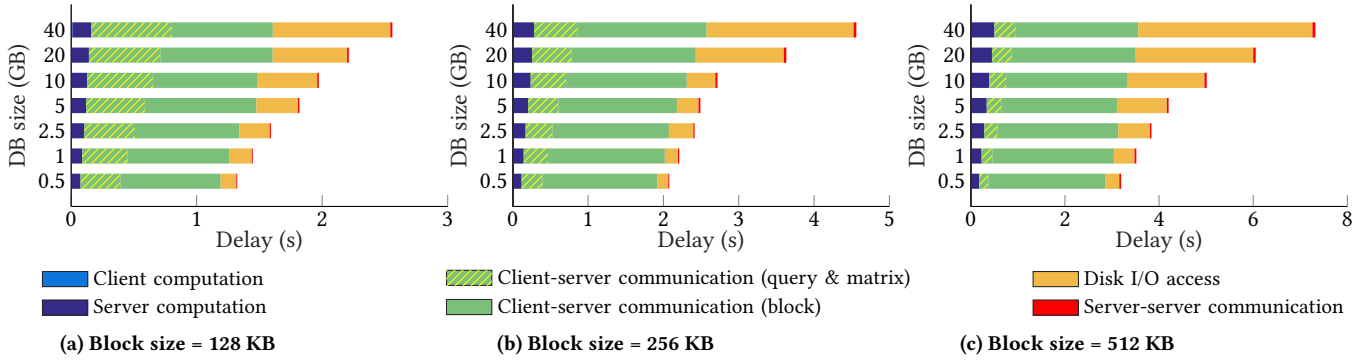
• *Path-ORAM*: Most of the delays in Path-ORAM was due to $O(\log N)$ client-server communication blowup, which accounted for 97% of overall delay. The client computation was negligible (cost less than 2%) due to IND-CPA encryption/decryption. Since the Path-ORAM bucket size is also much smaller than S^{3}ORAM and Onion-ORAM (i.e., 5 vs. 333) which incurs less data to be read and to be written. Therefore, the disk I/O access time of Path-ORAM only took 1% of the overall delay.

• *Ring-ORAM*: Similar to Path-ORAM, the most significant delay of Ring-ORAM was due to the amortized communication cost of early reshuffle and eviction operations, which accounted for 96% of total delay presented in Figure 3. Client and server computations were negligible due to IND-CPA encryption and XOR operations accelerated by multi-threading, respectively. The disk I/O access was also negligible since the server only read 1 block per bucket. Thus, all operations except the communication only contributed less than 4% to the total delay.

• *Onion-ORAM*: Due to AHE, the computation cost dominated all other costs in Onion-ORAM scheme. Specifically, we estimated that given a database of size 0.5 GB containing 256-KB blocks, the server computation with multi-core processing might take approximately 3 hours, which accounted for 99% of the overall delay. Meanwhile, the client took around 38 seconds to generate a select query. Although the disk I/O access time of Onion-ORAM is similar to that of S^{3}ORAM (due to the same bucket size setting), it contributed the least. Transmission also took a small amount of time since Onion-ORAM offers $O(1)$ bandwidth blowup.

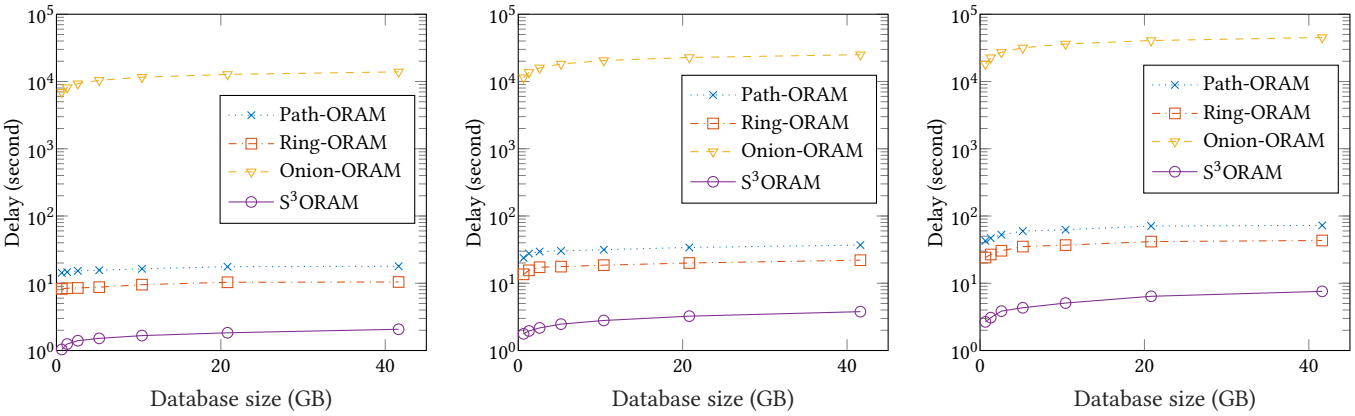
Experiments with a Mobile Client Device. Figure 6 presents end-to-end delays of S^{3}ORAM and its counterparts on a mobile client device with LTE network. The server computation of Onion-ORAM still dominated all others as in previous settings. In addition, Onion-ORAM also requires the client to perform costly computations to generate encrypted select queries, which took a few minutes with a mobile client device. Note that the client computations just took around 50 milliseconds in S^{3}ORAM as shown in Figure 7. The performance of Path-ORAM and Ring-ORAM in the LTE network was relatively better than that of the home network since the LTE network offered a slightly higher upload throughput. S^{3}ORAM was also slightly better but not affected much by the limited client computation. That is, although the client computation on the mobile device contributed a slightly larger delay than that on the laptop, this portion still occupied less than 5% of the total cost.

Comparison of S^{3}ORAM with Path-ORAM and Ring-ORAM for varying network bandwidths. S^{3}ORAM outperformed its counterparts in both home and mobile network settings. However, assume that the user has high bandwidth network connection. In this case, S^{3}ORAM might not be the best choice in terms of end-to-end delay. Hence, we ran another experiment to show that, $O(\log N)$-communication ORAMs are better than computational ORAMs after a certain threshold of network bandwidth. Specifically,



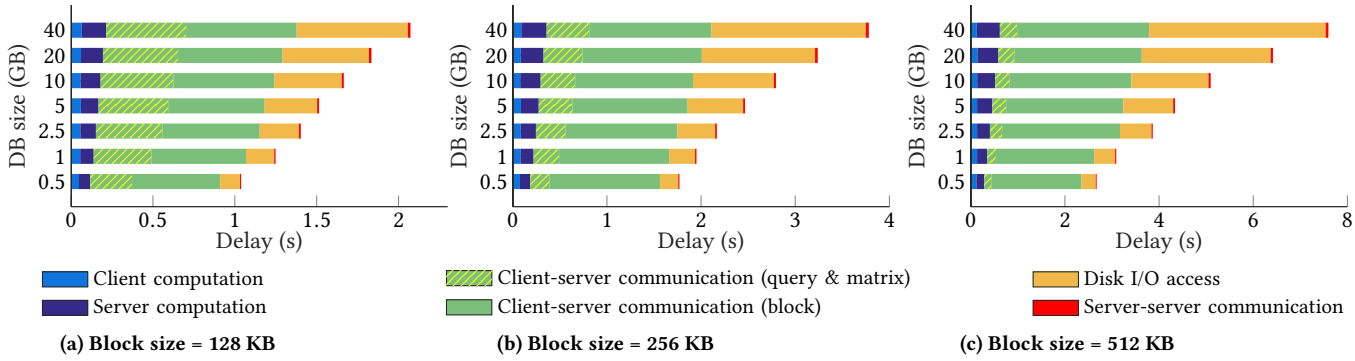
(a) Block size = 128 KB **(b) Block size = 256 KB** **(c) Block size = 512 KB**

Figure 5: Detailed cost breakdown of S^{3}ORAM on a laptop with home network.



(a) Block size = 128 KB **(b) Block size = 256 KB** **(c) Block size = 512 KB**

Figure 6: End-to-end delay of S^{3}ORAM and its counterparts on a mobile client device with LTE network.

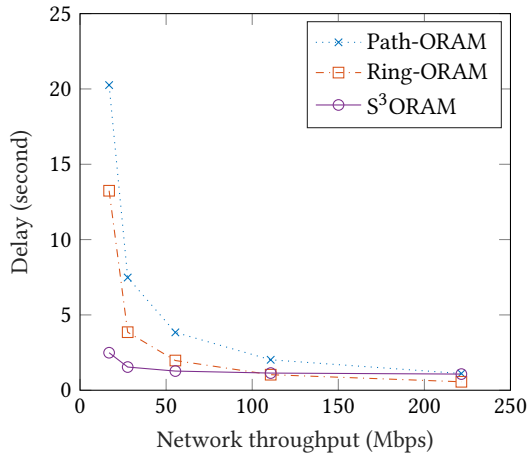


(a) Block size = 128 KB **(b) Block size = 256 KB** **(c) Block size = 512 KB**

Figure 7: Detailed cost breakdown of S^{3}ORAM on a mobile client device with LTE network.

we executed S^{3}ORAM and its counterparts several thousand times for increasing network bandwidth values. Figure 8 presents the performance of ORAM schemes with a database size of 40 GB and a block size of 128 KB with varying network throughputs. Observe that Path-ORAM and Ring-ORAM surpassed S^{3}ORAM for a network throughput of approximately 240 Mbps and 110 Mbps, respectively. This is because Path-ORAM and Ring-ORAM are $O(\log N)$ bandwidth blowup ORAMs and they receive a high benefit from increasing network speeds. However, S^{3}ORAM is $O(1)$ bandwidth blowup ORAM and receives a less benefit from a fast network.

Storage Overhead. Since S^{3}ORAM and Onion-ORAM feature $O(1)$ block storage, their client storage cost is lower than that of their counterparts. Given a database with 512 KB blocks, Path-ORAM and Ring-ORAM require around 32-33 MB for the stash, while S^{3}ORAM and Onion-ORAM do not require the stash. The storage cost for a position map in non-recursive S^{3}ORAM is slightly higher than its non-recursive counterparts. For instance, with a 16 TB database of



*We exclude the Onion-ORAM since its plot is far beyond the limit of the y-axis.

Figure 8: Delay for varying network throughput.

512-KB blocks ($N = 33554432$), S^{3}ORAM costs 119 MB while the others (e.g., Onion-ORAM, Ring-ORAM, Path-ORAM) cost 100 MB.

In S^{3}ORAM, for a database with N blocks of size B bits, the server storage overhead is $4N \cdot B$ bits for *each* server (recall that S^{3}ORAM needs at least three servers). The server storage for Path-ORAM and Ring-ORAM is $10N \cdot B$ bits and $6N \cdot B$ bits, respectively. The server storage for Onion-ORAM is similar to S^{3}ORAM for one server but will increase after a sequence of access operation due to the ciphertext expansion of AHE.

6 RELATED WORK

Single-server ORAM without computation. The first ORAM proposed by Goldreich et al. [20] was in the context of software protection and followed by refinements (e.g., [21]). The recent ORAM schemes mainly have been considered in the client-server model to hide data access pattern over a remote server (e.g., [32]). Preliminary ORAMs were costly in terms of both communication and storage overhead, but recent ORAMs (e.g., [35, 39, 40, 42]) showed significant improvements. Path-ORAM [40], which follows the tree structure of [35], achieves $O(\log N)$ communication blowup. Various ORAMs relying on Path-ORAM have been proposed for specific applications such as oblivious data structure (i.e., [44]), secure computation (e.g., ([42], [43]), Parallel ORAM [8]) and secure processor [25]. However, Path-ORAM based schemes inherit its logarithmic communication blowup [6, 30].

Single-server ORAM with computation. Ring-ORAM [33] reduced the communication cost of Path-ORAM by 2.5x given that the server performs XORs. Some other alternatives (e.g., [3, 12, 14, 26, 29]) leveraged single-server PIR or fully/partial HE to further reduce the communication cost. For instance, Onion-ORAM [12] achieves $O(1)$ bandwidth blowup, where the client and server interactively run partial HE operations. Path-PIR scheme in [26] used PIR scheme in [41] with Additively HE (AHE) (i.e., [31]) on top of tree ORAM structure [35]. Bucket-ORAM in [14] used AHE on top of the underlying ORAM structure composed of tree ORAM and hierarchical ORAM. The scheme in [11] used PIR scheme in [41] on top of ObliviStore [38], which is based on Partition-ORAM in [39]. The TWORAM scheme in [15] constructed a garbled circuit [45] over the tree ORAM structure, which allows the client and server to perform secure computation to access the block.

Multi-server (Distributed) ORAM. Distributed ORAM schemes were proposed to eliminate highly costly fully/partial HE operations. CHf-ORAM [27] attempted to use four non-colluding servers to achieve $O(1)$ bandwidth blowup under $O(1)$ blocks of client storage. However, CHf-ORAM [27] (as well as its predecessor in [29]) was broken by Abraham et al. in [1] which also showed an asymptotically tight sub-logarithmic communication bound for composing ORAM with PIR. Abraham et al. in [1] also presented a scheme using two non-colluding servers to perform XORs for block retrieval over a k-ary ORAM tree structure. Stefanov et al. in [37] proposed an ORAM that uses two non-colluding computational-capable servers to reduce the client-server bandwidth of Partition ORAM [39]. In a different line of research, distributed ORAM schemes were proposed for secure multi-party computation (e.g., [13, 24]). In these works, the access patterns are hidden from all parties so that such ORAM schemes are integrated with some secure computation protocol (e.g., Yao's garbled circuit [45]) and, therefore, their cost is higher than classical client-server ORAM model.

7 CONCLUSION

We developed a new distributed ORAM scheme that we named S^{3}ORAM, which achieves $O(1)$ client-server bandwidth blowup under $O(1)$ client block storage and a low end-to-end delay by avoiding costly HE operations. S^{3}ORAM harnesses Shamir Secret Sharing, tree-based ORAM structure, a new triplet eviction strategy, and a secure multi-party multiplication protocol in an effective manner to achieve these objectives. We performed detailed experiments in an actual cloud environment with a resource-limited mobile client to assess the effectiveness of S^{3}ORAM for various networking settings such as high bandwidth, home and wireless (i.e., Wi-Fi, LTE) networks. Our experiments showed that S^{3}ORAM is three orders of magnitude faster than the existing single-server ORAM with $O(1)$ client communication/storage blowup complexity (Onion-ORAM). S^{3}ORAM is also one order of magnitude faster than Path-ORAM on a moderate network bandwidth quality, which is typical for various real-life settings (e.g., home, wireless networks and inter-state cloud deployments).

ACKNOWLEDGMENT

We would like to thank the anonymous reviewers for their insightful comments and suggestions to improve the quality of this work. This work is supported by the NSF CAREER Award CNS-1652389 and an unrestricted gift from Robert Bosch LLC.

REFERENCES

[1] Ittai Abraham, Christopher W Fletcher, Kartik Nayak, Benny Pinkas, and Ling Ren. 2017. Asymptotically Tight Bounds for Composing ORAM with PIR. In *IACR International Workshop on Public Key Cryptography*. Springer, 91–120.
[2] Anastasov Anton. 2016. Implementing Onion ORAM: A Constant Bandwidth ORAM using AHE. https://github.com/aanastasov/onion-oram/blob/master/doc/report.pdf. (2016).
[3] Daniel Apon, Jonathan Katz, Elaine Shi, and Aishwarya Thiruvengadam. 2014. Verifiable oblivious storage. In *International Workshop on Public Key Cryptography*. Springer, 131–148.

[4] Amos Beimel and Yoav Stahl. 2002. Robust information-theoretic private information retrieval. In *International Conference on Security in Communication Networks*. Springer, 326–341.

[5] M. Ben-Or, S. Goldwasser, and A. Wigderson. 1988. Completeness Theorems for Non-Cryptographic Fault-Tolerant Distributed Computation (Extended Abstract). In *Proceedings of the 20th Annual ACM Symposium on Theory of Computing*, Janos Simon (Ed.). ACM, 1–10.

[6] Vincent Bindschaedler, Muhammad Naveed, Xiaorui Pan, XiaoFeng Wang, and Yan Huang. 2015. Practicing oblivious access on cloud storage: the gap, the fallacy, and the new way forward. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM, 837–849.

[7] Elette Boyle and Moni Naor. 2016. Is There an Oblivious RAM Lower Bound?. In *Proceedings of the 2016 ACM Conference on Innovations in Theoretical Computer Science*. ACM, 357–368.

[8] Binyi Chen, Huijia Lin, and Stefano Tessaro. 2016. Oblivious parallel ram: Improved efficiency and generic constructions. In *Theory of Cryptography Conference*. Springer, 205–234.

[9] Benny Chor, Eyal Kushilevitz, Oded Goldreich, and Madhu Sudan. 1998. Private information retrieval. *Journal of the ACM (JACM)* 45, 6 (1998), 965–981.

[10] Ivan Damgård and Mads Jurik. 2001. A generalisation, a simpli. cation and some applications of paillier's probabilistic public-key system. In *International Workshop on Public Key Cryptography*. Springer, 119–136.

[11] Jonathan Dautrich and Chinya Ravishankar. 2015. Combining ORAM with PIR to minimize bandwidth costs. In *Proceedings of the 5th ACM Conference on Data and Application Security and Privacy*. ACM, 289–296.

[12] Srinivas Devadas, Marten van Dijk, Christopher W Fletcher, Ling Ren, Elaine Shi, and Daniel Wichs. 2016. Onion oram: A constant bandwidth blowup oblivious ram. In *Theory of Cryptography Conference*. Springer, 145–174.

[13] Sky Faber, Stanislaw Jarecki, Sotirios Kentros, and Boyang Wei. 2015. Three-party ORAM for secure computation. In *International Conference on the Theory and Application of Cryptology and Information Security*. Springer, 360–385.

[14] Christopher Fletcher, Muhammad Naveed, Ling Ren, Elaine Shi, and Emil Stefanov. 2015. *Bucket ORAM: single online roundtrip, constant bandwidth oblivious RAM.* Technical Report. IACR Cryptology ePrint Archive, Report 2015, 1065.

[15] Sanjam Garg, Payman Mohassel, and Charalampos Papamanthou. 2015. *TWORAM: round-optimal oblivious RAM with applications to searchable encryption.* Technical Report. IACR Cryptology ePrint Archive, 2015: 1010.

[16] Rosario Gennaro, Yuval Ishai, Eyal Kushilevitz, and Tal Rabin. 2001. The round complexity of verifiable secret sharing and secure multicast. In *Proceedings of the thirty-third annual ACM symposium on Theory of computing*. ACM, 580–589.

[17] Rosario Gennaro, Michael O Rabin, and Tal Rabin. 1998. Simplified VSS and fast-track multiparty computations with applications to threshold cryptography. In *Proceedings of the seventeenth annual ACM symposium on Principles of distributed computing*. ACM, 101–111.

[18] Craig Gentry, Kenny A Goldman, Shai Halevi, Charanjit Julta, Mariana Raykova, and Daniel Wichs. 2013. Optimizing ORAM and using it efficiently for secure computation. In *International Symposium on Privacy Enhancing Technologies Symposium*. Springer, 1–18.

[19] Ian Goldberg. 2007. Improving the robustness of private information retrieval. In *2007 IEEE Symposium on Security and Privacy (SP'07)*. IEEE, 131–148.

[20] Oded Goldreich. 1987. Towards a theory of software protection and simulation by oblivious RAMs. In *Proceedings of the nineteenth annual ACM symposium on Theory of computing*. ACM, 182–194.

[21] Oded Goldreich and Rafail Ostrovsky. 1996. Software protection and simulation on oblivious RAMs. *Journal of the ACM (JACM)* 43, 3 (1996), 431–473.

[22] Thang Hoang, Attila Altay Yavuz, and Jorge Guajardo. 2016. Practical and secure dynamic searchable encryption via oblivious access on distributed data structure. In *Proceedings of the 32nd Annual Conference on Computer Security Applications*. ACM, 302–313.

[23] Ang Li, Xiaowei Yang, Srikanth Kandula, and Ming Zhang. 2010. CloudCmp: comparing public cloud providers. In *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement*. ACM, 1–14.

[24] Steve Lu and Rafail Ostrovsky. 2013. Distributed oblivious RAM for secure two-party computation. In *Theory of Cryptography*. Springer, 377–396.

[25] Martin Maas, Eric Love, Emil Stefanov, Mohit Tiwari, Elaine Shi, Krste Asanovic, John Kubiatowicz, and Dawn Song. 2013. Phantom: Practical oblivious computation in a secure processor. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. ACM, 311–324.

[26] Travis Mayberry, Erik-Oliver Blass, and Agnes Hui Chan. 2014. Efficient Private File Retrieval by Combining ORAM and PIR.. In *NDSS*. Citeseer.

[27] Tarik Moataz, Erik-Oliver Blass, and Travis Mayberry. [n. d.]. CHf-ORAM: A Constant Communication ORAM without Homomorphic Encryption. ([n. d.]).

[28] Tarik Moataz, Erik-Oliver Blass, and Travis Mayberry. 2015. *Constant communication ORAM without encryption.* Technical Report. IACR Cryptology ePrint Archive, Report 2015/1116.

[29] Tarik Moataz, Travis Mayberry, and Erik-Oliver Blass. 2015. Constant communication ORAM with small blocksize. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM, 862–873.

[30] Muhammad Naveed. 2015. The Fallacy of Composition of Oblivious RAM and Searchable Encryption. *IACR Cryptology ePrint Archive* 2015 (2015), 668.

[31] Pascal Paillier. 1999. Public-key cryptosystems based on composite degree residuosity classes. In *International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 223–238.

[32] Benny Pinkas and Tzachy Reinman. 2010. Oblivious RAM revisited. In *Advances in Cryptology–CRYPTO 2010*. Springer, 502–519.

[33] Ling Ren, Christopher W Fletcher, Albert Kwon, Emil Stefanov, Elaine Shi, Marten van Dijk, and Srinivas Devadas. 2014. Ring ORAM: Closing the Gap Between Small and Large Client Storage Oblivious RAM. *IACR Cryptology ePrint Archive* 2014 (2014), 997.

[34] Adi Shamir. 1979. How to share a secret. *Commun. ACM* 22, 11 (1979), 612–613.

[35] Elaine Shi, T-H Hubert Chan, Emil Stefanov, and Mingfei Li. 2011. Oblivious RAM with O ((logN) 3) worst-case cost. In *Advances in Cryptology–ASIACRYPT 2011*. Springer, 197–214.

[36] Emil Stefanov, Charalampos Papamanthou, and Elaine Shi. 2014. Practical Dynamic Searchable Encryption with Small Leakage.. In *NDSS*, Vol. 71. 72–75.

[37] Emil Stefanov and Elaine Shi. 2013. Multi-cloud oblivious storage. In *2013 ACM SIGSAC conference on Computer & communications security*. ACM, 247–258.

[38] Emil Stefanov and Elaine Shi. 2013. Oblivistore: High performance oblivious cloud storage. In *Security and Privacy (SP), 2013 IEEE Symposium on*. IEEE, 253–267.

[39] Emil Stefanov, Elaine Shi, and Dawn Song. 2011. Towards practical oblivious RAM. *arXiv preprint arXiv:1106.3652* (2011).

[40] Emil Stefanov, Marten Van Dijk, Elaine Shi, Christopher Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. 2013. Path ORAM: an extremely simple oblivious RAM protocol. In *Proceedings of the 2013 ACM SIGSAC conference on Computer and Communications security*. ACM, 299–310.

[41] Jonathan Trostle and Andy Parrish. 2010. Efficient computationally private information retrieval from anonymity or trapdoor groups. In *International Conference on Information Security*. Springer, 114–128.

[42] Xiao Wang, Hubert Chan, and Elaine Shi. 2015. Circuit oram: On tightness of the goldreich-ostrovsky lower bound. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM, 850–861.

[43] Xiao Wang, Yan Huang, TH Hubert Chan, Abhi Shelat, and Elaine Shi. 2014. SCORAM: oblivious RAM for secure computation. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 191–202.

[44] Xiao Shaun Wang, Kartik Nayak, Chang Liu, TH Chan, Elaine Shi, Emil Stefanov, and Yan Huang. 2014. Oblivious data structures. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 215–226.

[45] Andrew C Yao. 1982. Protocols for secure computations. In *23rd Annual Symposium on Foundations of Computer Science, 1982*. IEEE, 160–164.

APPENDIX

We present the proof of theorems presented in Section 4 as follows:

Proof of Theorem 4.2. We see that S^{3}ORAM is correct iff (i) the S^{3}ORAM.Retrieve subroutine returns the correct value of the retrieved block, (ii) the write-to-root operation (step 10, Protocol 2) is consistent, and (iii) the S^{3}ORAM.SSS-SMP-TripletEviction subroutine is consistent.

• *Correctness of S^{3}ORAM.Retrieve.* For each data request x, let b be the block to be retrieved and j be the location of b in its path (i.e., $j :=$ pm[id].pIdx where id is the identifier of b). So, the share of select query for server $\mathcal{S}_i$ is of form: $[\![\mathbf{e}]\!]_i^{(t)} = ([\![e]\!]_1^{(t)}, \ldots, [\![e]\!]_n^{(t)})$, where $n = Z \cdot (H+1)$ and $e_i = 0$ for $1 \le i \ne j \le n, e_j = 1$. Let $[\![\mathbf{c}_u]\!] = ([\![c_{u1}]\!], \ldots, [\![c_{un}]\!])$ be the vector consisting of the share of u-th chunks taken from Z slots in every bucket residing in the read path. For $1 \le u \le m$, the answer of each server $\mathcal{S}_i$ is of form:

$$
\begin{aligned}
[\![\mathbf{e}]\!]_i^{(t)} \cdot [\![\mathbf{c}_u]\!]_i^{(t)} &= \sum_{k=1}^{n} \left([\![e_k]\!]^{(t)} \cdot [\![c_{u,k}]\!]^{(t)} \right) \\
&= \sum_{k=1}^{n} [\![e_k \cdot c_{u,k}]\!]^{(2t)} && \text{by Eq. (3)} \\
&= [\![c_{u,j}]\!]^{(2t)} && \text{by Eq. (1)}
\end{aligned}
$$

By SSS scheme, at least $2t+1$ shares are required to recover the secret encrypted by a random $2t$-degree polynomial. Our system

model presented in Section 2 follows this and, therefore, the client always computes the correct value of chunk c_t by $c_t \leftarrow \text{SSS.Recover}(\llbracket c_t \rrbracket_1^{(2t)}, \ldots, \llbracket c_t \rrbracket_\ell^{(2t)})$. Since all chunks of b are correctly computed, b is properly retrieved with probability 1.

• *Consistency of write-to-root (step 10, Protocol 2):* Lemma 4.1 implies that the root bucket is empty after eviction. The client writes the retrieved block to an empty slot in the root bucket according to the subsequent order. Since $Z = A$, this ensures that slots containing retrieved blocks are not overwritten before the eviction happens.

• *Consistency of S³ORAM.SSS-SMP-TripletEviction:* Lemma 4.1 implies that sibling buckets are empty due to previous evictions and, therefore, they can hold all data moved from source buckets without creating inconsistency. Moving data from source buckets to destination buckets is achieved via matrix products. These computations are correct due to homomorphic properties of two-share addition and multiplication offered by SSS and the SMP protocol which was proven to be correct in [17], respectively. □

Proof of Theorem 4.3. Given a request sequence $\mathbf{x}$ of length q, where $x_j = (\text{op}_j, \text{id}_j, \text{data}_j)$ as in Definition 2.4, let $\text{S}^3\text{ORAM}_i(\mathbf{x})$ be the S^3ORAM client's sequence of interactions with server $\mathcal{S}_i$ including a sequence of retrieval, write-to-root and eviction operations. We have that write-to-root operation is deterministic and is performed after retrieval where the previously retrieved block is written to a publicly known slot in the root bucket (step 10, Protocol 2). The eviction is also deterministic which is performed after every A successive accesses regardless of any data being requested (step 12). Due to the independence between retrieval, write-to-bucket and eviction operations, we consider $\text{S}^3\text{ORAM}_i(\mathbf{x})$ to contain separate sequences of these operations observed by $\mathcal{S}_i$ as:

$$\text{S}^3\text{ORAM}_i(\mathbf{x}) = \begin{cases} \vec{R}_i(\mathbf{x}) & = & \left(R_i^{(x_1)}, \ldots, R_i^{(x_q)}\right) \\ \vec{W}_i(\tilde{\mathbf{x}}) & = & \left(W_i^{(\tilde{x}_1)}, \ldots, W_i^{(\tilde{x}_q)}\right), \\ \vec{E}_i(\bar{\mathbf{x}}) & = & \left(E_i^{(\bar{x}_1)}, \ldots, E_i^{(\bar{x}_{q/A})}\right) \end{cases} \tag{7}$$

where $\vec{W}_i(\tilde{\mathbf{x}})$ and $\vec{E}(\bar{\mathbf{x}})$ denote the deterministic write-to-bucket and eviction sequences, given data access sequence $\mathbf{x}$, respectively.

Assume that there is a coalition of up t servers $\{\mathcal{S}_{i\in\mathcal{I}}\}$ sharing their own transcripts with each other. Let $\mathcal{I} \subseteq \{1, \ldots, \ell\}$ such that $|\mathcal{I}| \leq t$. The view of $\{\mathcal{S}_{i\in\mathcal{I}}\}$ can be derived from Eq. (7) as:

$$\left\{\text{S}^3\text{ORAM}_{i\in\mathcal{I}}(\mathbf{x})\right\} = \begin{cases} \{\vec{R}_{i\in\mathcal{I}}(\mathbf{x})\} & = & \left(\{R_{i\in\mathcal{I}}^{(x_1)}\}, \ldots, \{R_{i\in\mathcal{I}}^{(x_q)}\}\right) \\ \{\vec{W}_{i\in\mathcal{I}}(\tilde{\mathbf{x}})\} & = & \left(\{W_{i\in\mathcal{I}}^{(\tilde{x}_1)}\}, \ldots, \{W_{i\in\mathcal{I}}^{(\tilde{x}_q)}\}\right), \\ \{\vec{E}_{i\in\mathcal{I}}(\bar{\mathbf{x}})\} & = & \left(\{E_{i\in\mathcal{I}}^{(1)}\}, \ldots, \{E_{i\in\mathcal{I}}^{(q/A)}\}\right) \end{cases}$$

We show that, for any two access sequences $\mathbf{x}$ and $\mathbf{x}'$ of the same length ($|\mathbf{x}| = |\mathbf{x}'|$), the pairs $\left\langle \{\vec{R}_{i\in\mathcal{I}}(\mathbf{x})\}, \{\vec{W}_{i\in\mathcal{I}}(\tilde{\mathbf{x}})\}, \{\vec{E}_{i\in\mathcal{I}}(\mathbf{x})\} \right\rangle$ and $\left\langle \{\vec{R}_{i\in\mathcal{I}}(\mathbf{x}')\}, \{\vec{W}_{i\in\mathcal{I}}(\tilde{\mathbf{x}}')\}, \{\vec{E}_{i\in\mathcal{I}}(\bar{\mathbf{x}}')\} \right\rangle$ are identically distributed.

• *Retrieval transcripts:* For each access request $x_j \in \mathbf{x}$, $\{\mathcal{S}_{i\in\mathcal{I}}\}$ observes a transcript $\{R_{i\in\mathcal{I}}^{(x_j)}\}$ consisting of a retrieval path $\mathcal{P}_{x_j}$ (access pattern) which is identical for all servers (step 4, Subroutine 1) and data generated in SSS-based select scheme (steps 5–8).

The access pattern of S^3ORAM is identical for all other tree-based ORAM schemes. Specifically, each block in S^3ORAM is assigned to a leaf bucket selected randomly and independently from each other. Once a block is accessed, its position is assigned to a new bucket leaf selected randomly and independently. Therefore, access patterns generated by any data request sequences of the same length are statistically indistinguishable.

We next analyze the probability distribution of data observed at the server side in each S^3ORAM retrieval as follows. For each retrieval, the client sends to servers select queries generated by PIR.CreateQuery algorithm. Such queries are SSS shares and, therefore, achieve t-privacy. The inner product is also t-private due to Lemma 2.1 with addition and partial multiplication homomorphic properties (1) and (3). So, any data generated in S^3ORAM retrievals are identically distributed in the presence of t colluding servers.

By these properties, for any data request sequence $\mathbf{x}$, the corresponding transcripts (including access patterns) generated in the S^3ORAM retrieval phase are information-theoretically (statistically) indistinguishable from random access sequence in the presence of up to t colluding servers.

• *Write-to-root transcripts:* Data are written to slots in the root bucket according to subsequent order and, therefore, the access pattern is deterministic. Such written data are SSS-shared with new random polynomials so that they are t-private. Therefore, any data request sequence generates write-to-root transcripts which are identically distributed.

• *Eviction transcripts:* Since eviction is deterministic which follows publicly-known reverse lexicographical order like in Onion-ORAM (e.g., [12]), the access patterns of $\{E_{i\in\mathcal{I}}^{(j)}\}$ and $\{E_{i\in\mathcal{I}}^{(j')}\}$ are independent to each other for any $(j, j') \in \{0, \ldots, 2^H\}$. We next show that data generated in independent evictions are identically distributed.

For each eviction, the client sends H permutation matrices which are SSS-share and, therefore, they are all t-private and uniformly distributed. Data in sibling buckets are t-private and uniformly distributed since they are merely copied from source buckets deterministically (step 19, Subroutine 3). The matrix product computations (step 23) are also t-private according to Lemma 2.1 with properties (1) and (3). Finally, the SMP protocol ensures that data in destination buckets are t-private and uniformly distributed (step 24) as shown in Lemma 2.2.

Given two request sequences $\mathbf{x}$, $\mathbf{y}$ with $|\mathbf{x}| = |\mathbf{y}|$, the corresponding deterministic eviction sequences observed by $\{\mathcal{S}_{i\in\mathcal{I}}\}$ are:

$$\{\vec{E}_{i\in\mathcal{I}}(\bar{\mathbf{x}})\} = \left(\{E_{i\in\mathcal{I}}^{(\bar{x}_1)}\}, \ldots, \{E_{i\in\mathcal{I}}^{(\bar{x}_{q/A})}\}\right)$$

$$\{\vec{E}_{i\in\mathcal{I}}(\bar{\mathbf{y}})\} = \left(\{E_{i\in\mathcal{I}}^{(\bar{y}_1)}\}, \ldots, \{E_{i\in\mathcal{I}}^{(\bar{y}_{q/A})}\}\right)$$

where $(\bar{x}_j, \bar{y}_j) \in \{0, \ldots, H\}$ for $1 \leq j \leq q/A$. Since data yielded in $\{E_{i\in\mathcal{I}}^{(\bar{j}_j)}\}$ and $\{E_{i\in\mathcal{I}}^{(\bar{x}_{j'})}\}$ are identically distributed for all $(j, j') \in \{\bar{x}_1, \ldots, \bar{x}_{q/A}\} \cup \{\bar{y}_1, \ldots, \bar{y}_{q/A}\}$ as shown above, $\{\vec{E}_{i\in\mathcal{I}}(\bar{\mathbf{x}})\}$ and $\{\vec{E}_{i\in\mathcal{I}}(\bar{\mathbf{y}})\}$ are identically distributed.

• *Final indistinguishability argument:* Given any data request sequences of the same length, S^3ORAM generates *(i)* access patterns statistically indistinguishable from random request sequence, and *(ii)* identically (uniform) distributed data in the presence of up to t colluding servers. This indicates that S^3ORAM scheme achieves (information-theoretic) t-security according to Definition 2.4.

□