A Multi-server ORAM Framework with Constant Client Bandwidth Blowup

THANG HOANG* and ATTILA A. YAVUZ, The Department of Computer Science and Engineering, University of South Florida
JORGE GUAJARDO, Robert Bosch LLC, Research and Technology Center, USA

Oblivious Random Access Machine (ORAM) allows a client to hide the access pattern when accessing sensitive data on a remote server. It is known that there exists a logarithmic communication lower bound on any passive ORAM construction, where the server only acts as the storage service. This overhead, however, was shown costly for some applications. Several active ORAM schemes with server computation have been proposed to overcome this limitation. However, they mostly rely on costly homomorphic encryptions, whose performance is worse than passive ORAM. In this article, we propose S^3ORAM, a new multi-server ORAM framework, which features $O(1)$ client bandwidth blowup and low client storage without relying on costly cryptographic primitives. Our key idea is to harness Shamir Secret Sharing and a multi-party multiplication protocol on applicable binary tree-ORAM paradigms. This strategy allows the client to instruct the server(s) to perform secure and efficient computation on his/her behalf with a low intervention thereby, achieving a constant client bandwidth blowup and low server computational overhead. Our framework can also work atop a general k-ary tree ORAM structure ($k \geq 2$). We fully implemented our framework, and strictly evaluated its performance on a commodity cloud platform (Amazon EC2). Our comprehensive experiments confirmed the efficiency of S^3ORAM framework, where it is approximately 10× faster than the most efficient passive ORAM (i.e., Path-ORAM) for a moderate network bandwidth while being three orders of magnitude faster than active ORAM with $O(1)$ bandwidth blowup (i.e., Onion-ORAM). We have open-sourced the implementation of our framework for public testing and adaptation.

ACM Reference Format:
Thang Hoang, Attila A. Yavuz, and Jorge Guajardo. 2019. A Multi-server ORAM Framework with Constant Client Bandwidth Blowup. *ACM Trans. Priv. Sec.* 1, 1 (November 2019), 34 pages. https://doi.org/0000001.0000001

1 INTRODUCTION

Recent data breach incidents (e.g., Apple iCloud, Equifax, British Airways) have shown the importance of preserving user privacy on the cloud. An important aspect of enhancing user privacy is applying encryption on sensitive data. For instance, standard encryption (e.g., AES) can provide data confidentiality. However, this might not be sufficient to preserve user privacy. Specifically, sensitive information can still be inferred by observing user access patterns, even when the query and the outsourced data are both encrypted [24]. To conceal the access pattern, Oblivious Random Access Machine (ORAM) has been proposed [19]. ORAM plays an important role for privacy-preserving

*Part of this work done while the first author was at Oregon State University.

Authors' addresses: Thang Hoang; Attila A. Yavuz, The Department of Computer Science and Engineering, University of South Florida, Tampa, Florida, 33620, hoangm@mail.usf.edu, attilaayavuz@usf.edu; Jorge Guajardo, Robert Bosch LLC, Research and Technology Center, USA, Pittsburgh, PA, 15223, Jorge.GuajardoMerchan@us.bosch.com.

2471-2566/2019/11-ART $15.00
https://doi.org/0000001.0000001

cloud applications thanks to its strong privacy features (e.g., confidentiality, access pattern obfuscation). However, some state-of-the-art ORAM schemes might still be costly for certain real-life applications due to their high communication and/or computation overhead [1, 6, 23, 30, 31]. In the following, we outline the state-of-the-art ORAM schemes and their limitations, and then present our research objective toward mitigating some of these limitations.

1.1 The Limitations of the State-of-the-art and Our Objectives

The first ORAM scheme was proposed by Goldreich *et al.* [19], who later proved that any ORAM construction must incur an asymptotically logarithmic communication lower bound of $\Omega(\log N)$, where N is the number of outsourced data blocks. Since then, a number of ORAM schemes have been proposed in attempts to achieve the established lower bound (e.g., [17, 33, 36, 40]). The most efficient and simple ORAM is Path-ORAM [41], whose bandwidth overhead met the Goldreich and Ostrovsky's ORAM lower bound (i.e., $O(\log N)$ transmitted blocks per access). Despite its merits, Path-ORAM has been shown costly for some cloud applications [6, 31, 37]. It has recently been re-confirmed that there is a logarithmic communication lower bound in any secure *passive* ORAM construction, where the server offers only the storage facility (i.e., no computation). Therefore, to bypass this communication barrier, several *active* ORAM schemes with server computation have been proposed (e.g., [11, 27, 34]). However, most of these ORAM designs cannot surpass the logarithmic bandwidth overhead asymptotically, some of which [11, 27] incur a significant computation cost such as single-server Private-Information Retrieval (PIR) [42]. To the best of our knowledge, the state-of-the-art ORAM schemes with $O(1)$ bandwidth overhead rely on either Fully or Partially Homomorphic Encryption (HE) (e.g., [32]). (e.g., Onion-ORAM [12], Bucket-ORAM [14], and [3]). Unfortunately, it has been shown that [1, 29] HE computation takes much longer execution time than streaming $O(\log N)$ blocks in Path-ORAM.

To avoid costly computation, ORAM in the distributed setting has been explored. Although the first multi-server ORAM scheme [38] can achieve $O(1)$ client-server communication with the cost of $O(\log N)$ server-server communication, it requires the client to store $O(\sqrt{N})$ blocks, which might not be suitable for storage-constraints clients such as mobile devices. Later multi-server ORAM schemes leveraged multi-server PIR (e.g., [9]) to reduce the bandwidth overhead asymptotically without using costly HE operations. Abraham *et al.* [1] indicated that there exists an asymptotically sub-logarithmic communication lower bound of $\Omega(\log_{cD} N)$ for the ORAM and PIR composition, where c, D are the numbers of blocks stored by the client and performed by PIR operations, respectively. As a result, although the scheme in [28] claimed to achieve $O(1)$ bandwidth overhead under $O(1)$ blocks of client storage, it has been shown to violate the bound with two concrete attacks [1] .

Our objectives. In many practical scenarios, it may not be possible to guarantee a reliable and high bandwidth network connection between the client and server. This is particularly true in the case of home networks and mobile devices with wireless network connectivity (e.g., Wi-Fi, LTE). Given that ORAM with $O(\log N)$ client bandwidth overhead (e.g., Path-ORAM [41]) may not be suitable for such contexts, there is a significant need to design a new ORAM scheme that can achieve $O(1)$ client bandwidth overhead. It is also important that the proposed ORAM is suitable for resource-limited clients and only incurs a low delay to provide a desirable quality of service.

*Our objective is to create an efficient ORAM scheme that simultaneously achieves (*i*) a low client communication overhead (i.e., $O(1)$ bandwidth overhead), (*ii*) low computational overhead at both client- and server-side, and (*iii*) low storage.*

Table 1. Summary of S^{3}ORAM schemes and some of their counterparts.

Scheme	Bandwidth Overhead†		Block Size*	Server Computation	Client Block Storage‡	# servers
	Client-server	Server-server				
Path-ORAM [41]	$O(\log N)$	-	$\Omega(1)$	-	$O(\log N)$	1
Ring-ORAM [34]	$O(\log N)$	-	$\Omega(1)$	XOR	$O(\log N)$	1
Onion-ORAM [12]	$O(1)$	-	$\Omega(\log^5 N)$	Additively HE [10]	$O(1)$	1
Dist. OblivStore [38]	$O(1)$	$O(\log N)$	$\Omega(1)$	Permutation and IND-CPA encryption	$O(\sqrt{N})$	2
2-Server ORAM [25]	$O(\log N)$	-	$\Omega(1)$	Permutation and cuckoo hashing	$O(1)$	2
S^{3}ORAM$^{\mathrm{O}}$	$O(1)$	$O(\log N)$	$\Omega(\log^2 N)$	Secure addition and multiplication of SSS values	$O(1)$	3
S^{3}ORAM$^{\mathrm{C}}$	$O(1)$	$O(\log N)$	$\Omega(\log N)$		$O(\log N)$	

• We refer reader to §7 for the detail experimental and analytical comparisons between S^{3}ORAM schemes and their counterparts.
† *Bandwidth overhead* denotes the number of blocks being transmitted between the client and the server(s) or between the servers. Due to the eviction, the server-server bandwidth overhead of S^{3}ORAM$^{\mathrm{O}}$ is $O(\lambda \log N)$, where λ is the statistical security parameter. Since the eviction is performed every $\lambda/2$ access requests, the amortized server-server bandwidth overhead of S^{3}ORAM$^{\mathrm{O}}$ is $O(\log N)$.
∗ This indicates the minimal block size needed to absorb the transmission cost of the retrieval query and the eviction data, thereby meeting the expected client-bandwidth overhead. In this table, we consider all the ORAM schemes in the *non-recursive* form, where the position map is stored at the client.
‡ *Client block storage* is defined as the number of data blocks being temporarily stored at the client. This is equivalent to the stash component used in [34, 41], which, therefore, does not include the cost of storing the position map of size $O(N \log N)$. Notice that all the ORAM schemes in this table, except the one in [25], require the position map component.

1.2 Our Contribution

In this paper, we present S^{3}ORAM, a new distributed ORAM framework, which features $O(1)$ client bandwidth blowup, low storage and efficient computation at both client- and server-side. Our proposed framework consists of two multi-server active ORAM schemes including S^{3}ORAM$^{\mathrm{O}}$ and S^{3}ORAM$^{\mathrm{C}}$, in which the former minimizes the client storage requirement while the latter optimizes the computation and storage overhead at the server-side. We first present our main idea and then outline the desirable properties of our proposed framework as follows.

Main idea. Most efficient ORAM schemes to-date follow the tree paradigm by Shi *et al.* [36]. In this paradigm, there are two main procedures for each ORAM access: retrieval and eviction. Our intuition is to harness the homomorphic properties of Shamir secret sharing along with a secure multi-party multiplication protocol to perform these procedures in an oblivious manner. To achieve $O(1)$ client-bandwidth overhead, it is imperative to ensure that each procedure only incurs a small constant number of data blocks to be transmitted between the client and the server(s). In the standard (single-server) ORAM setting, we observe that both Onion-ORAM [12] and Circuit-ORAM [43] schemes require low client storage and offer elegant retrieval and eviction strategies that can be further implemented with SSS homomorphic computation to achieve $O(1)$ client-bandwidth overhead. Therefore, the main idea of S^{3}ORAM$^{\mathrm{O}}$ and S^{3}ORAM$^{\mathrm{C}}$ schemes in our S^{3}ORAM framework is to harness SSS and SMM protocol to perform the retrieval and eviction operations in the line of Onion-ORAM and Circuit-ORAM, respectively, but in a significantly more computation- and client bandwidth-efficient manner. By doing this, S^{3}ORAM$^{\mathrm{O}}$ (*resp.* S^{3}ORAM$^{\mathrm{C}}$) inherits all desirable properties of Onion-ORAM (*resp.* Circuit-ORAM) regarding the low client storage cost, while achieving $O(1)$ client-bandwidth overhead *without* the costly homomorphic operations but instead requiring only a lightweight computation and suitability for small block sizes. Table 1 outlines a high-level comparison of S^{3}ORAM and its counterparts.

Desirable properties. Our S^{3}ORAM framework offers the following properties.

- *Low client-server communication:* All schemes in S^{3}ORAM framework offer $O(1)$ client bandwidth blowup, compared with $O(\log N)$ of Path-ORAM [41] and Ring-ORAM [34] (with a fixed number of servers). S^{3}ORAM schemes feature a smaller block size (i.e., $\Omega(\log^2 N)$ in S^{3}ORAM$^{\mathrm{O}}$, $\Omega(\log N)$ in S^{3}ORAM$^{\mathrm{C}}$), than state-of-the-art $O(1)$ bandwidth blowup ORAM schemes that

require Fully or Partially HE operations (e.g., $\Omega(\log^5 N)$ in Onion-ORAM [12], $\Omega(\log^6 N)$ in Bucket-ORAM [14]).

- *Low client and server computation:* S^{3}ORAM schemes require the servers to perform only lightweight modular additions and multiplications, which are much more efficient than partial HE operations (e.g., [10]). In particular, we show in §7 that, the server computation of S^{3}ORAM schemes is three orders of magnitude faster than that of Onion-ORAM.
  The client in S^{3}ORAM schemes only performs lightweight computations for retrieval and eviction operations. Thus, it is more efficient than Onion-ORAM, which requires a number of HE operations. For example, S^{3}ORAM requires only a few milliseconds compared to minutes of Onion-ORAM to generate an encrypted access query (see §7). Moreover, since data blocks in S^{3}ORAM schemes are single-layered "encrypted", the "decryption" process is less costly so that it is faster than other ORAMs (e.g., [12, 38]), whose blocks are multi-layered encrypted.
- *Low end-to-end delay:* Due to low bandwidth and computation overhead, S^{3}ORAM schemes are approximately three orders of magnitude faster than Onion-ORAM, while it is one order of magnitude faster than Path-ORAM in networks with *moderate* client bandwidth. We notice that for S^{3}ORAM to provide all its advantages, it is assumed that good network throughput is available between the servers. In our detailed analysis in §7.2.3, we show that if the inter-server bandwidth is limited and the client has access to a high-speed Internet connection, state-of-the-art (single-server) ORAM schemes (i.e., Path-ORAM, Ring-ORAM) are more efficient than S^{3}ORAM (see §7.2.3 for a detailed analysis).
- *Low client storage:* S^{3}ORAM$^{\mathrm{O}}$ scheme features $O(1)$ blocks of client storage, compared with $O(\lambda)$ blocks in Path-ORAM/Ring-ORAM, and $O(\sqrt{N})$ blocks in [40]. S^{3}ORAM$^{\mathrm{C}}$ scheme achieves the same block of client storage with Path-ORAM/Ring-ORAM (i.e., $O(\lambda)$)
- *High security*: All S^{3}ORAM schemes achieve information-theoretic statistical security. The statistical bit comes from the tree-paradigm by Shi *et al.* [36]. The information-theoretic property comes from SSS and its multi-party multiplication protocol.
- *Full-fledged implementation and experiments:* We fully implemented S^{3}ORAM$^{\mathrm{C}}$ and S^{3}ORAM$^{\mathrm{O}}$ schemes in our S^{3}ORAM framework and evaluated their performance in an actual cloud environment (i.e., Amazon EC2). The detailed experiments in §7 showed that both S^{3}ORAM schemes are efficient in practice and they can be deployed on mobile devices with a limited computation capacity and low network connection. We have released the source code of S^{3}ORAM framework for public use and testing (see §6).

Security limitations of S^{3}ORAM. S^{3}ORAM harnesses the distributed setting to achieve constant client bandwidth overhead with efficient computation and the server-side simultaneously. It should be clear, however, that the use of standard secret sharing techniques and, in particular, Shamir secret sharing, renders our protocol vulnerable to collusion attacks (as it is standard in this setting). It should be observed that this *vulnerability* does not exist in the standard single-server ORAM model. Therefore, as it is also standard in this model, we note that S^{3}ORAM cannot offer any security guarantee if the number of colluding servers exceeds the privacy threshold. Another limitation of S^{3}ORAM is that it only offers security in the semi-honest setting (see [12] for the exemplified active attack). To make S^{3}ORAM secure against malicious adversaries, we can apply the "cut-and-choose" trick proposed in [12]; however, this may incur very high communication and computation overhead at the client. We leave the efficient extension of S^{3}ORAM to malicious security as an open research question.

Improvements over the CCS'17 conference version [22]. This article is the extended version of [22], which includes the following improvements. First, from the algorithmic point of view, we introduce a new S^{3}ORAM variant called S^{3}ORAM^C, which reduces the storage and computation overhead at the server side at the cost of small client storage. Second, following recent optimizations on tree-ORAM (e.g., [1, 17]), we show that our S^{3}ORAM schemes can be extended to work on top of k-ary tree layout, where $k \geq 2$ is a free parameter, to achieve sub-logarithmic overhead. We also show that although allowing the tree degree to be adjusted provides asymptotic improvements in the complexity of the algorithm, it turns out that the practical improvement is not significant, in which the tree-ORAM only works best with the small k (i.e., $k \in \{2, 3\}$). Third, we improved the implementation of S^{3}ORAM and revised all the experiments with more appropriate parameters to better capture the security and the performance of S^{3}ORAM in real-world applications. In summary, our main objective in this article is to harness state-of-the-art efficient eviction strategies proposed for Tree-ORAM paradigm (i.e., [12, 43]) with multi-party computation techniques (i.e., Shamir secret sharing [35]), and demonstrate via extensive experiments that this integration with optimizations offers one of the most efficient ORAM frameworks for data outsourcing. Finally, we have released the improved source-code of all S^{3}ORAM schemes for public use and adaptation. The code is publicly available at

https://github.com/thanghoang/S3ORAM

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 dot 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 $\mathbf{v}$. $\mathbf{u}[i]$ denotes accessing the i-th component of vector $\mathbf{u}$. $\mathbf{I}[i][*]$ and $\mathbf{I}[*][j]$ denote accessing the row i and column j of matrix $\mathbf{I}$, respectively. $\mathbf{I}[*][i \ldots j]$ denotes accessing the columns from i to j of matrix $\mathbf{I}$.

2.1 Model of Computation

Following the literature in distributed secure computation (e.g., [5, 18]), 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 with, 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 [18] for similar definitions in the context of distributed PIR) if any set of at most t parties cannot compute after the protocol execution 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

SSS.Create(α, t):

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

SSS.Recover($\mathcal{A}$, t):

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

Fig. 1. Shamir secret sharing scheme [35].

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 Secret Sharing (SSS) scheme [35], which comprises two algorithms SSS.Create and SSS.Recover as presented in Figure 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 $x_i \in \mathbb{F}_p^*$ 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).

We extend the notion of secret share for a value into the share for a vector in a 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 [35] 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, $\forall m, m' \in \mathbb{F}_p$, $\forall \mathcal{I} \subseteq \{1, \ldots, \ell\}$ such that $|\mathcal{I}| \leq t$ 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 1 summarizes the homomorphic properties of SSS and it was first described in [5].

Lemma 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*

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

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

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

- 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: $\quad ([\![\beta]\!]_j^{(t)})_{j=1}^{\ell} \leftarrow \text{SSS.Create}([\![\beta]\!]_i^{(2t)}, t)$
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: $\quad [\![\beta]\!]_i^{(t)} \leftarrow \sum_{j=1}^{2t+1} \mathbf{V}^{-1}[1, j] \cdot [\![\beta]\!]_j^{(t)}$

Fig. 2. Multi-party multiplication protocol on SSS shares [16].

- *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 (Eq. 3) in Lemma 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[1].

2.3 Secure Multi-party Multiplication

Gennaro *et al.* [16] presented a Secure Multi-pary Multiplication (SMM) protocol 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 $1 \leq i \leq \ell$ 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 following structure.

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

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

Lemma 2 (SMM protocol privacy [16]). The SMM protocol in [16] (denoted as $\star$ operator) offers homomorphic property for full multiplication between two SSS-shares, whose result is t-private as:

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

[1] Benor *et al.* [5] proposed a secure multiplication protocol, however the protocol of Gennaro *et al.* [16] is more efficient and thus, is the subject of §2.3.

Table 2. Notations.

Symbol	Description
T	S^3ORAM tree structure.
Z	Bucket size.
T[i], T[i][j]	i-th bucket of S^3ORAM tree T and j-th slot in the i-th bucket of T.
$\|b\|$, b, c	Block size, block and block chunk, respectively.
N, m	Number of blocks and number of chunks in a block.
H	Height of the S^3ORAM tree.
pm	Position map.
(pID, pIdx) $\leftarrow$ pm[id]	Precise location (i.e., path ID and path index) of the block id.
$\mathcal{I} \leftarrow \mathcal{P}$(pID)	Set of indexes of buckets residing in the path pID.

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, 18] as follows.

Definition 1 (Multi-server PIR [4, 9, 18]). Let $\mathbf{DB} = (b_1, \ldots, b_n)$ 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 **DB** to be retrieved, the client creates queries $(e_1, \ldots, e_\ell) \leftarrow$ PIR.CreateQuery(i) and distributes e_j to server $\mathcal{S}_j$. Each server responds with an answer $a_j \leftarrow$ PIR.Retrieve(e_j, **DB**). Upon receiving ℓ answers, the client computes the value of item b_i by invoking the reconstruction algorithm $b \leftarrow$ PIR.Reconstruct($a_1, \ldots, a_\ell$).

The 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 [18].

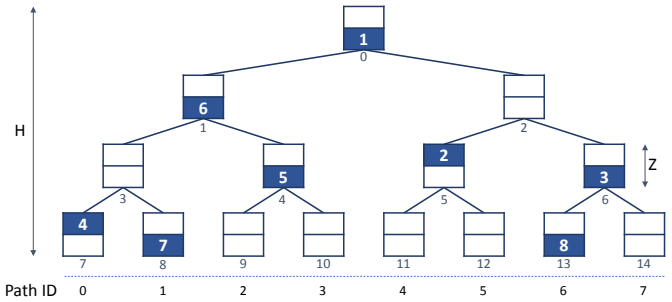
2.5 Multi-server ORAM Security

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 (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 $|\mathcal{I}| \leq t$, 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 (perfectly/statistically/computationally) indistinguishable.

3 THE PROPOSED S^3ORAM FRAMEWORK

S^3ORAM follows the typical procedure of tree-based ORAMs [36]. 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



POSITION MAP

Block ID	Path ID	Path Location
1	1	1
2	4	4
3	6	5
4	0	6
5	2	5
6	3	3
7	1	7

Fig. 3. Tree-ORAM paradigm by Shi *et al.* [36].

bucket. Finally, an eviction operation is performed in order to percolate data blocks to lower levels in the ORAM structure. *The intuition behind S³ORAM access protocol is as follows:* (1) We integrate SSS with a multi-server PIR protocol to perform a private retrieval operation with some homomorphic properties; (2) We leverage these homomorphic properties of SSS and a SMM 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. Notice that the idea of using PIR to implement the ORAM retrieval phase was first suggested in [27], and later in some subsequent works such as [1, 11, 12, 21]. In Table 2, we outline the notation used in the S³ORAM schemes and throughout the rest of the paper.

3.1 Overview of S³ORAM Framework

3.1.1 Data Structure. S³ORAM schemes follow the tree paradigm proposed by Shi *et al.* [36], in which the outsourced database is split into size-equal blocks and then organized to a balanced binary tree (T) with a height of H (Figure 3). Each node in T is called a *bucket* with Z slots so that it can store up to Z data blocks. Thus, T can store up to $N < Z \cdot 2^H$ data blocks.

At the client-side, the client maintains a position map component (pm) to keep track of the assigned path (pID) for each data block in the tree. Additionally, the client stores the location of each data block in its assigned path. Hence, pm is of structure pm := (id, ⟨pID, pIdx⟩), where id is the block ID, $1 \leq \text{pID} \leq 2^H$ is the assigned path of the block, and $1 \leq \text{pIdx} \leq Z \cdot (H+1)$ is the location of the block in its path. The client also maintains a so-called stash component (denoted as S) to temporarily store accessed block(s) from the tree.

In the S³ORAM framework, the tree structure is SSS-shared among ℓ servers. Figure 4 presents the Setup algorithm to construct data structures in S³ORAM schemes given a database input DB. First, the client organizes DB into N data blocks, and then initializes every slot in each bucket of the tree (T) with a 0's string of length $|b|$ (lines 1-2). The client arranges all blocks into T, wherein each block (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 $c_j \in \mathbb{F}_p$ (line 9)[2]. Finally, the client creates shares of T via SSS.Create algorithm for each chunk in each slot in T (line 10). The S³ORAM distributed data structure consists of ℓ shares of T as $\{[\![\mathsf{T}]\!]_1, \ldots, [\![\mathsf{T}]\!]_\ell\}$.

Figure 5 presents the general access operation of S³ORAM schemes following the tree-ORAM paradigm. Basically, there are two main subroutines in the S³ORAM.Access algorithm: Retrieve (line 1) and Evict (line 6). The former is to obliviously retrieve the block of interest from the

[2]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.

$\underline{\text{S}^3\text{ORAM.Setup(DB):}}$
1: Split DB into blocks $(b_1, \ldots, b_N)$ with corresponding IDs $(\text{id}_1, \ldots, \text{id}_N)$
2: $\text{T}[i][j] \leftarrow \{0\}^{|b|}$ for $1 \le i < 2^{H+1}$ and $1 \le j \le Z$
3: **for** $i = 1, \ldots, N$ **do**
4: $\quad z_i \xleftarrow{\$} \{1, \ldots, 2^H\}$
5: $\quad$ Put b_i into an empty slot indexed y of the leaf bucket in path z_i
6: $\quad \text{pm}[\text{id}_i] \leftarrow (z_i, H \cdot Z + y)$
7: **for** $i = 1, \ldots, 2^{H+1} - 1$ **do**
8: $\quad$ **for** $j = 1, \ldots, Z$ **do**
9: $\quad\quad (c_{i,j}^{(1)}, \ldots, c_{i,j}^{(m)}) \leftarrow \text{T}[i, j]$, where $c_{i,j}^{(k)} \in \mathbb{F}_p$
10: $\quad\quad (\llbracket c_{i,j}^{(k)} \rrbracket_1, \ldots, \llbracket c_{i,j}^{(k)} \rrbracket_\ell) \leftarrow \text{SSS.Create}(c_{i,j}^{(k)}, t)$ for $1 \le k \le m$
11: $\quad\quad \llbracket \text{T}[i, j] \rrbracket_l \leftarrow (\llbracket c_{i,j}^{(1)} \rrbracket_l, \ldots, \llbracket c_{i,j}^{(m)} \rrbracket_l)$ for $1 \le l \le \ell$
12: **return** $(\llbracket \text{T} \rrbracket_1, \ldots, \llbracket \text{T} \rrbracket_\ell)$ ▷ Send $\llbracket \text{T} \rrbracket_i$ to $\mathcal{S}_i$ for $1 \le i \le \ell$

Fig. 4. S^3ORAM setup algorithm.

$\underline{\text{S}^3\text{ORAM.Access(op, id, } b^*\text{):}}$
1: $b \leftarrow \text{S}^3\text{ORAM.Retrieve(id)}$
2: $\text{pm}[\text{id}].\text{pID} \xleftarrow{\$} \{1, \ldots, k^{H+1}\}$
3: **if** op = write **then**
4: $\quad b \leftarrow b^*$
5: $S \leftarrow S \cup b$
6: Execute $\text{S}^3\text{ORAM.Evict()}$
7: **return** b

Fig. 5. General access procedure in S^3ORAM schemes.

ORAM-tree stored on the cloud, while the latter is to obliviously write the retrieved block back to the ORAM-tree. Once the block is retrieved, it is assigned to a new random path (line 2), updated if needed (line 3), and then stored in the stash (line 5) to be pushed back later via the Evict protocol.

In our S^3ORAM framework, we select the eviction path deterministically, which follows the reverse lexicographical order proposed in [17]. Specifically, given a binary tree of height H, where edges in each level are indexed by either 0 (left) or 1 (right) as exemplified in Figure 3, the collection of edges of the eviction path at the n_e-th eviction operation is calculated by the following formula.

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

where DigitReverse_2 denotes the order-reversal of the binary string representation of the decimal integer input.

In the following, we present the main scheme in our S^3ORAM framework called $\text{S}^3\text{ORAM}^\text{O}$, which features the low client storage overhead. We describe another S^3ORAM scheme called $\text{S}^3\text{ORAM}^\text{C}$, which offers efficient computation and low server storage overhead with the cost of client storage afterward.

3.2 $\text{S}^3\text{ORAM}^\text{O}$: S^3ORAM with Low Client Storage

We introduce $\text{S}^3\text{ORAM}^\text{O}$, an S^3ORAM scheme that does not require the client to maintain the stash component, thereby saving a factor of $O(\lambda)$ client storage overhead. To achieve this, $\text{S}^3\text{ORAM}^\text{O}$ follows the Triplet Eviction strategy in [12]. To enable $O(1)$ client-bandwidth blowup, $\text{S}^3\text{ORAM}^\text{O}$ harnesses homomorphic properties of SSS, which allows the client to "instruct" the servers to

PIR.CreateQuery(j):
1: Let $\mathbf{e} := (e_1, \ldots, e_n)$, where $e_j \leftarrow 1$, $e_i \leftarrow 0$ for $1 \leq i \neq j \leq n$
2: **for** $i = 1, \ldots, n$ **do**
3: $(\llbracket e_i \rrbracket_1^{(t)}, \ldots, \llbracket e_i \rrbracket_\ell^{(t)}) \leftarrow$ 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 $1 \leq i \leq \ell$
5: **return** $(\llbracket \mathbf{e} \rrbracket_1^{(t)}, \ldots, \llbracket \mathbf{e} \rrbracket_\ell^{(t)})$

PIR.Retrieve($\llbracket \mathbf{e} \rrbracket_i^{(t)}$, $\llbracket \mathbf{DB} \rrbracket_i^{(t)}$):
6: $\llbracket b \rrbracket_i^{(2t)} \leftarrow \llbracket \mathbf{e} \rrbracket_i^{(t)} \cdot \llbracket \mathbf{DB} \rrbracket_i^{(t)}$
7: **return** $\llbracket b \rrbracket_i^{(2t)}$

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

Fig. 6. SSS-based PIR scheme.

perform efficient retrieval and eviction operations in a secure manner without having to download and upload $O(\log N)$ data blocks.

In the following, we describe in detail the retrieval and eviction protocol of S^{3}ORAM$^{\text{O}}$ scheme.

3.2.1 Retrieval subroutine. To achieve $O(1)$ client bandwidth blowup, the retrieval protocol in S^{3}ORAM$^{\text{O}}$ scheme requires an efficient PIR protocol to privately retrieve the block of interest. We first describe the PIR protocol based on SSS as follows.

SSS-based PIR scheme. Our objective is to privately retrieve a block of interest residing in the queried path on the S^{3}ORAM$^{\text{O}}$ tree. Recall that in the single-server HE-based ORAM schemes (e.g., [3, 12]), the PIR query is encrypted with additive/fully HE. In S^{3}ORAM$^{\text{O}}$, the tree is SSS-shared among ℓ servers, which features highly efficient additive and multiplicative homomorphic properties. We observe that the multi-server PIR scheme in [4, 18] relies on SSS to create PIR queries and, therefore, it can serve as a suitable private retrieval tool to be used for S^{3}ORAM$^{\text{O}}$ scheme. We describe SSS-based PIR scheme in Figure 6, and further outline it as follows:

Assume that each server $\mathcal{S}_i$ stores a share of the database **DB** containing n blocks denoted as $\llbracket \mathbf{DB} \rrbracket_i$, which can be interpreted as a vector with each i-th component being the share of the i-th item in **DB**. Let j be the index of the block in **DB** to be privately retrieved. The client executes the PIR.CreateQuery algorithm, which creates an n-dimensional unit vector with all zero coordinates except the j-th coordinate being set to 1 (line 1) and then, secret-shares it with SSS (lines 2–3). The client then distributes these shares to the corresponding servers, each answering with the result of the dot product between the received share vector and its share of **DB** by executing the PIR.Retrieve algorithm (line 6). Finally, the client executes the PIR.Reconstruct algorithm, which invokes the SSS.Recover algorithm over ℓ answers to recover the desired block (line 8). Since **DB** in this context is SSS-secret shared instead of plaintext as in [4, 18], our PIR.Reconstruct algorithm requires *at least* $2t + 1$ shares (instead of $t + 1$) to recover the item correctly.

We present the retrieval protocol in S^{3}ORAM$^{\text{O}}$ in Figure 7, which employs three algorithms of the above SSS-based PIR scheme. Given the block to be read, the client first determines its location in the S^{3}ORAM$^{\text{O}}$ tree via the position map pm (line 1) and then, privately retrieves it using the SSS-based PIR protocol. In this case, the server interprets all slots in the retrieval path as the database input **DB** in the PIR.Retrieve algorithm. Hence, the size of **DB** and the length of the

S^3ORAM^O.Retrieve(id):

Client:

1: $(s, j) \leftarrow \text{pm}[\text{id}]$
2: $(\llbracket \mathbf{e} \rrbracket_1^{(t)}, \ldots, \llbracket \mathbf{e} \rrbracket_\ell^{(t)}) \leftarrow \text{PIR.CreateQuery}(j)$
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: 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: $\llbracket c_j \rrbracket_i^{(2t)} \leftarrow \text{PIR.Retrieve}(\llbracket \mathbf{e} \rrbracket_i^{(t)}, \llbracket \mathbf{c}_j \rrbracket_i^{(t)})$
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: $c_j \leftarrow \text{PIR.Reconstruct}(\llbracket c_j \rrbracket_1^{(2t)}, \ldots, \llbracket c_j \rrbracket_\ell^{(2t)})$ for $1 \le j \le m$
10: $b \leftarrow (c_1, \ldots, c_m)$
11: **return** b

Fig. 7. S^3ORAM^O retrieval subroutine.

S^3ORAM^O.Evict():

1: $(c_1, \ldots, c_m) \leftarrow b$, where b is the block that has just been retrieved
2: $(\llbracket c_j \rrbracket_1, \ldots, \llbracket c_j \rrbracket_\ell) \leftarrow \text{SSS.Create}(c_j, t)$ for $1 \le j \le m$
3: 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 $\mathcal{S}_i$ for $1 \le i \le \ell$
4: $n_r \leftarrow n_r + 1 \mod A$ ▷ n_r is initialized with 0
5: **if** $n_r = 0$ **then**
6: $v \leftarrow \text{DigitReverse}_2(n_e \mod 2^H)$
7: Execute S^3ORAM^O.EvictAlongPath(v) protocol
8: $n_e \leftarrow n_e + 1 \mod 2^H$ ▷ n_e is initialized with 0

Fig. 8. S^3ORAM^O eviction subroutine.

query vector is $n = Z \cdot (H + 1)$. Since there are m separate chunks in each slot, the servers execute the PIR.Retrieve algorithm m times with the same PIR query but over different $\mathbf{DB}_j$, where each $\mathbf{DB}_j$ contains the j-th chunk of all slots in the retrieval path (lines 5–7). Finally, the client obtains the desired block by recovering all chunks upon receiving their corresponding shares using the PIR.Reconstruct algorithm (line 9).

3.2.2 Eviction subroutine. To eliminate the need of maintaining the stash component at the client-side, S^3ORAM^O follows the Triplet Eviction strategy proposed in [12]. The S^3ORAM^O.Evict algorithm in Figure 8 presents the eviction procedure in S^3ORAM^O scheme. Specifically, after the block is privately retrieved via the S^3ORAM^O.Retrieve protocol, the client creates new SSS-shares for it (lines 1–2), and then writes the share to an empty slot in the root bucket of the corresponding server (lines 3). After $A \le Z$ successive retrievals, the client selects a deterministic eviction path following the reverse lexicographical order (line 5) as presented in §3.1, and executes the S^3ORAM^O.EvictAlongPath protocol, to obliviously percolate the blocks from upper levels (e.g., root bucket) to deeper levels (e.g., leaf buckets).

According to the Triplet Eviction policy, for each level in the eviction path, all blocks from the source bucket ($\mathsf{T}[i]$) will be obliviously moved to all its children (i.e., $\mathsf{T}[2i]$, $\mathsf{T}[2i + 1]$). We follow the same terminology used in [12] to denote the buckets involved in each Triplet Eviction operation: If the child of the source bucket resides in the eviction path, it is called the *destination* bucket while

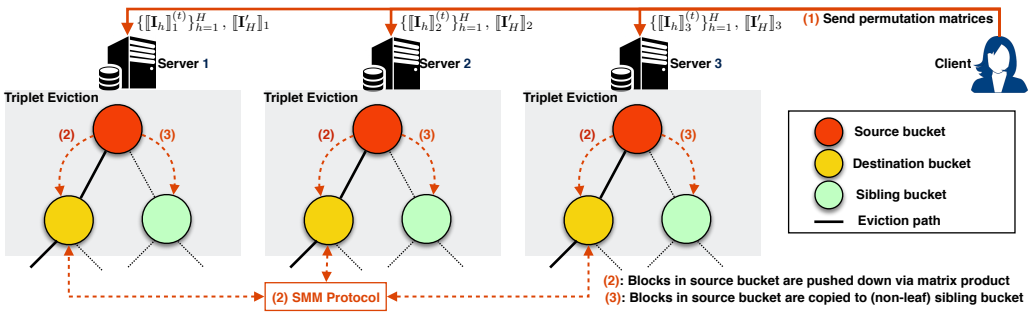


Fig. 9. The Triplet Eviction using SSS and SMM protocol.

the other child is called the *sibling* bucket (see Figure 9 for clarification). In our S^{3}ORAM$^{\mathrm{O}}$ scheme, the move is performed by computing the matrix product, in which the client creates permutation matrices and requests the servers to jointly perform the matrix product between such matrices and vectors containing data along the eviction path in the S^{3}ORAM$^{\mathrm{O}}$ tree. We present the algorithmic description of this strategy in the S^{3}ORAM$^{\mathrm{O}}$.EvictAlongPath protocol in Figure 10 with details as follows.

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 $\mathbf{I} \in \{0,1\}^{2Z\times Z}$ (line 2) such that the matrix product between $[\![\mathbf{u}]\!]$ 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 $[\![\mathbf{v}]\!]$ (line 7). As a result, $\mathbf{I}[i+Z,i] \leftarrow 1$ if the block currently at position i in $[\![\mathbf{v}]\!]$ remains (line 12). 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 $[\![\mathbf{I}]\!] \in \mathbb{F}_p^{2Z\times Z}$ (line 13). 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 and privacy of the S^{3}ORAM$^{\mathrm{O}}$ tree structure, servers will jointly perform the SMM protocol in §2.3 to reduce the degree of the polynomial of each component in $[\![\mathbf{v}]\!]$ from $2t$ to t (line 22 and line 26).

Source to sibling. We can apply the same trick as in the *source-to-destination* above to obliviously move real blocks in source buckets to their sibling buckets. However, since the non-leaf sibling buckets are guaranteed to be empty due to previous evictions passing on them (see Lemma 3), this process can be further optimized as discussed in [12] as follows. For each non-leaf sibling bucket in the eviction path, the client simply requests servers to copy all the data in the source bucket to the sibling bucket (line 19) and then, updates locally the path location of blocks in the position map (pm) accordingly (line 9–10). For the leaf sibling bucket, since it is not guaranteed to be empty at any time, we use the matrix permutation to move blocks from the source bucket to it as described above. This optimization can halve the client-server and server-server bandwidth cost as well as the server computation. 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 9 visualizes this new SSS-based Triplet Eviction strategy.

3.2.3 Asymptotic cost analysis. We analyze the cost of S^{3}ORAM$^{\mathrm{O}}$ pertaining to the block size ($|b|$), number of blocks (N), and statistical security parameter (λ). We consider other system parameters (e.g., prime field $\mathbb{F}_p$, number of servers ℓ) to be fixed.

S^3ORAM^O.EvictAlongPath(v):

Let $(u_1, \ldots, u_H)$ be the (ordered) indexes of source buckets along the eviction path v

Client:

1: **for** $h = 1, \ldots, H$ **do**
2: Let $\mathbf{I}_h$ be a $2Z \times Z$ matrix, set $\mathbf{I}_h[*, *] \leftarrow 0$
3: **for** each real block with id in the source bucket $\mathsf{T}[u_h]$ **do**
4: **if** id can legally reside in the *destination* bucket of $\mathsf{T}[u_h]$ **then**
5: (pID, pIdx) $\leftarrow$ pm[id]
6: Let y $(1 \le y \le Z)$ be the index of an empty slot in the *destination* bucket of $\mathsf{T}[u_h]$
7: $\mathbf{I}_h[x][Z + y] \leftarrow 1$, where $x \leftarrow$ pIdx $\mod Z$
8: pm[id′].pIdx $\leftarrow Z \cdot h + y$ ▷ Update the new location of the block in the path
9: **else** ▷ id can legally reside in the *sibling* bucket of $\mathsf{T}[u_h]$
10: pm[id].pIdx $\leftarrow$ pm[id].pIdx $+ Z$
11: **for** each real block with id′ in the *destination* bucket of $\mathsf{T}[u_h]$ **do**
12: $\mathbf{I}_h[x + Z][x] \leftarrow 1$, where $x \leftarrow$ pm[id′].pIdx $\mod Z$
13: $[\![\mathbf{I}_h[x, y]]\!]_1^{(t)}, \ldots, [\![\mathbf{I}_h[x, y]]\!]_\ell^{(t)} \leftarrow$ SSS.Create($\mathbf{I}_h[x, y]$, t) for $1 \le x \le 2Z$, $1 \le y \le Z$
14: Repeat lines 2–13 (excluded lines 9-10) to create $[\![\mathbf{I'}_H]\!]$, the share of permutation matrix for source to *sibling* bucket at the leaf level ($h = H$)
15: Send $([\![\mathbf{I'}_H]\!]_i^{(t)}, [\![\mathbf{I}_1]\!]_i^{(t)}, \ldots, [\![\mathbf{I}_H]\!]_i^{(t)})$ to $\mathcal{S}_i$, for $1 \le i \le \ell$

Server: each $\mathcal{S}_i \in \{\mathcal{S}_1, \ldots, \mathcal{S}_\ell\}$ receiving $([\![\mathbf{I'}_H]\!]_i^{(t)}, [\![\mathbf{I}_1]\!]_i^{(t)}, \ldots, [\![\mathbf{I}_H]\!]_i^{(t)})$ **do**

18: **for** $h = 1, \ldots, H$ **do**
19: Copy all data from source bucket $[\![\mathsf{T}[u_h]]\!]_i^{(t)}$ to its *non-leaf sibling* bucket
20: **for** $j = 1, \ldots, m$ **do**
21: Let $[\![\mathbf{c}_{h,j}]\!]_i^{(t)}$ be a vector containing j-th chunks of $[\![\mathsf{T}[u_h]]\!]_i^{(t)}$ and its *destination* bucket
22: $[\![\hat{\mathbf{c}}_{h,j}]\!]_i^{(t)} \leftarrow [\![\mathbf{c}_{h,j}]\!]_i^{(t)} \star [\![\mathbf{I}_h]\!]_i^{(t)}$
23: Update j-th chunks of the *destination* bucket of $[\![\mathsf{T}[u_h]]\!]_i^{(t)}$ with $[\![\hat{\mathbf{c}}_{h,j}]\!]_i^{(t)}$
24: **for** $j = 1, \ldots, m$ **do**
25: Let $[\![\mathbf{c'}_{H,j}]\!]_i^{(t)}$ be a vector containing j-th chunks of source bucket $[\![\mathsf{T}[u_H]]\!]_i^{(t)}$ and its *(leaf) sibling* bucket
26: $[\![\hat{\mathbf{c}}'_{H,j}]\!]_i^{(t)} \leftarrow [\![\mathbf{c}'_{H,j}]\!]_i^{(t)} \star [\![\mathbf{I'}_H]\!]_i^{(t)}$
27: Update j-th chunks of the *sibling* bucket of $[\![\mathsf{T}[u_H]]\!]_i^{(t)}$ with $[\![\hat{\mathbf{c}}'_{H,j}]\!]_i^{(t)}$

Fig. 10. S^3ORAM^O Triplet Eviction with SSS scheme and SMM protocol.

Communication. In the S^3ORAM^O retrieval phase, each PIR query being sent to ℓ servers is of size $(Z \cdot (H + 1) \cdot \lceil \log_2 p \rceil)$ bits. The client exchanges one block of size $|b|$ with each server. The Triplet Eviction is performed after every A subsequent retrievals. In this operation, the client sends $H + 1$ permutation matrices to ℓ servers. Each matrix is of size $2Z^2 \cdot \lceil \log_2 p \rceil$ bits. The servers exchange the shares of $H + 1$ buckets with each other, each being of size $Z \cdot |b|$ bits. Therefore, given $H = O(\log N), Z = A = O(\lambda)$ and ℓ, p are constants, the *amortized* client-server communication complexity is $O(|b| + \lambda \cdot \log N)$. The *amortized* server-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 cost of client-server communication by using ORAM to access the block *vs.* the base case where the block is insecurely accessed without ORAM. Our analyzed communication complexity of S^3ORAM^O above indicates that the size of the PIR query and the permutation matrices is *independent* of the block size parameter $|b|$. Therefore, the $O(1)$ client bandwidth blowup can be achieved in S^3ORAM^O by selecting a suitable value of $|b|$. That is, by selecting $|b| = \Omega(\lambda \cdot \log N)^3$, S^3ORAM achieves $O(1)$ client bandwidth blowup.

[3] In the ORAM community, $\lambda = O(\log N)$ is commonly used. With this assumption, the block size in S^3ORAM^O is $\Omega(\log^2 N)$

Computation. In the retrieval phase, the servers compute the dot product between the $Z \cdot (H+1)$-dimensional PIR query vector and the block vector containing $Z \cdot (H+1)$ blocks of size $|b|$. In the Triplet Eviction phase, the servers compute $H+1$ times the matrix product between a vector containing $2Z$ blocks of size $|b|$ and a permutation matrix of size $2Z \times Z$. The matrix product incurs re-sharing and computing the degree reduction in the SMM protocol on $Z \cdot (H+1)$ blocks each being of size $|b|$. In total, the *amortized* server computation complexity is $O(|b| \cdot \lambda \cdot \log N)$.

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

Storage. S^3ORAM^O layout is a full binary tree of height H, which has a total of $Z \cdot (2^{H+1} - 1)$ slots and can store up to $N \leq A \cdot 2^{H-1}$ real blocks. Given $A = Z = \Theta(\lambda)$ for statistical security (see Lemma 3), the server storage blowup cost is $O(1)$. 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^3ORAM is constant and does not increase after a sequence of access operations.

Similar to Onion-ORAM, S^3ORAM^O does not require the stash component since the retrieved block is immediately written back to the root bucket. Hence, the client block storage in S^3ORAM^O is $O(1)$. The client locally stores the position map whose cost is $O(N \cdot (\log N + \log\log N))$.

• *Achieving $O(1)$ client storage via recursion:* For theoretical interest, S^3ORAM^O can achieve (in total) $O(1)$ client storage by storing the position map in smaller ORAMs using the recursion technique in [40] and the bucket metadata structure in [12]. Specifically, for each bucket in the S^3ORAM^O tree, we create a metadata that stores the current index (pIdx) and the assigned path (pID) of blocks residing in it. For each S^3ORAM access, the metadata of buckets along the retrieval/eviction path will be read first to get the path and the location of blocks of interest. This information will be used to create the PIR query and permutation matrices. Next, we construct a series of S^3ORAM^O structures $S^3ORAM^O{}_0, \ldots, S^3ORAM^O{}_{\log_r N}$, where $S^3ORAM^O{}_0$ stores database blocks and each block j in $S^3ORAM^O{}_{i+1}$ stores the path information (pID) of the blocks $(j-1)r, \ldots, jr$ in $S^3ORAM^O{}_i$ and $r \geq 2$ is the compression ratio. We refer the reader to [12, 40] for the detailed descriptions.

For simplicity, we assume that $r = 2$ and let $H = \log N$ be the height of $S^3ORAM^O{}_0$. In $S^3ORAM^O{}_i$ ($i \geq 1$), the size of meta-data is $\lambda(H-i)$, the block size is $2(H-i+1)$, and the path length is $H-i$. There are $\log N$ recursive levels so that the total bandwidth overhead for each recursive S^3ORAM^O retrieval is $\lambda \sum_{i=0}^{H-1} i^2 + \sum_{i=1}^{H} 2(H-i+1) = O(\lambda \log^3 N)$. Due to amortization, the asymptotic cost of eviction is similar to the retrieval as analyzed above. Therefore, to achieve $O(1)$ client bandwidth blowup, the block size of $S^3ORAM^O{}_0$ needs to be $\Omega(\lambda \cdot \log^3 N)$. So, using the recursion technique to get rid of the client position map increases the regular block size a factor of $O(\log^2 N)$ and $O(\log N)$ communication *rounds*.

The regular block size in recursive S^3ORAM^O is a factor of $\log^2 N$ times larger than other (recursive) tree-based ORAM schemes featuring $O(\log N)$ bandwidth (e.g., Path ORAM, Ring-ORAM, Tree-ORAM) and (at least) $\log N$ times smaller than (recursive) tree-based ORAM with $O(1)$ bandwidth (e.g., Onion-ORAM [12], Bucket-ORAM [14], OVS [3]) due to the following reasons. As analyzed above, to keep the original asymptotic communication overhead intact when applying the recursion technique, the regular block size must be large enough to absorb the cost of transmitting the blocks and the meta-data components from $O(\log N)$ small (recursive) ORAM structures. In ORAM schemes with $O(\log N)$ bandwidth, since the size of small blocks in their recursive structures is $O(\log N)$, the regular block size is $\Omega(\log^2 N)$ to absorb the cost of downloading $O(\log^2 N)$ small blocks (there is no meta-data component in these schemes). On the other hand, the regular block size of $O(1)$-bandwidth ORAM schemes does not increase when applying the recursion, since it is

already larger than the total amount needed to absorb the cost of downloading the blocks and the meta-data of small ORAM structures (e.g., $\Omega(\log^5 N)$-$\Omega(\log^6 N)$ block size *vs.* $\Omega(\log^3 N)$ needed).

Given that the recursion technique significantly increases the regular block size, it is recommended to maintain the position map locally assuming that its size is small enough. This choice allows the implementor to gain the full performance advantages that S^{3}ORAM offers in practice.

3.3 S^{3}ORAM$^{\mathrm{C}}$: S^{3}ORAM with Low Server Storage and Computation Overhead

In this section, we present S^{3}ORAM$^{\mathrm{C}}$, a S^{3}ORAM scheme that achieves lower computational complexity than S^{3}ORAM$^{\mathrm{O}}$ due to its smaller bucket size parameter Z (e.g., $O(1)$ *vs.* $O(\lambda)$). The price to pay for such achievement is that it requires maintaining the stash at the client-side to temporarily store blocks that cannot be pushed back to the tree due to the small bucket size. The intuition of S^{3}ORAM$^{\mathrm{C}}$ is to implement the access protocol of Circuit-ORAM proposed by Wang *et al.* [43] using the homomorphic properties of SSS as follows.

3.3.1 Retrieval subroutine. S^{3}ORAM$^{\mathrm{C}}$ has the same retrieval procedure like S^{3}ORAM$^{\mathrm{O}}$ scheme, where we leverage SSS-based PIR Scheme presented in §3.2.1 to privately retrieve the block in the retrieval path of the S^{3}ORAM$^{\mathrm{C}}$ tree (Figure 11).

S^{3}ORAM$^{\mathrm{C}}$.Retrieve(id)
1: $b \leftarrow$ S^{3}ORAM$^{\mathrm{O}}$.Retrieve(id)
2: **return** b

Fig. 11. S^{3}ORAM$^{\mathrm{C}}$ retrieval subroutine.

3.3.2 Eviction subroutine. S^{3}ORAM$^{\mathrm{C}}$ implements the eviction principle in Circuit-ORAM scheme with additive and multiplicative homomorphic properties of SSS. Similar to S^{3}ORAM$^{\mathrm{O}}$ scheme, S^{3}ORAM$^{\mathrm{C}}$ selects a deterministic eviction path following the reverse lexicographical order (Eq. 6) proposed in [17] (Figure 12), which was proven to achieve the negligible overflow probability with a lower bucket size parameter compared with the random path (e.g., 2 *vs.* 3).

Intuitively, the client first scans the position map to prepare the target array that indicates which blocks to be pushed down to which levels in the eviction path. Afterward, the client goes through each level of the eviction path, picks the desired block and drops it to the target level. Notice that at any time, the client holds and drops at most one block. This policy is guaranteed by computing a target array that indicates whether to pick/drop the block in each level by scanning the position map. We refer the reader to [43] for the detailed description and explanation.

Figure 13 visualizes the high-level idea of the eviction in S^{3}ORAM$^{\mathrm{C}}$, which implements the push-down strategy in [43] using SSS and SMM protocol. Figure 14 describes the detailed algorithm with the high-level idea as follows. For each level (h) in the eviction path, the client creates a permutation matrix ($\mathbf{I}_h$) of size $(Z+1)\times(Z+1)$. We use the last column of the matrix ($\mathbf{I}_h[*][Z+1]$)

S^{3}ORAM$^{\mathrm{C}}$.Evict():
1: $v \leftarrow \mathrm{DigitReverse}_2(n_e \mod 2^H)$
2: Execute S^{3}ORAM$^{\mathrm{C}}$.EvictAlongPath(v) protocol
3: $n_e \leftarrow n_e + 1 \mod 2^H$
4: Repeat lines 1-3

Fig. 12. S^{3}ORAM$^{\mathrm{C}}$ eviction subroutine.

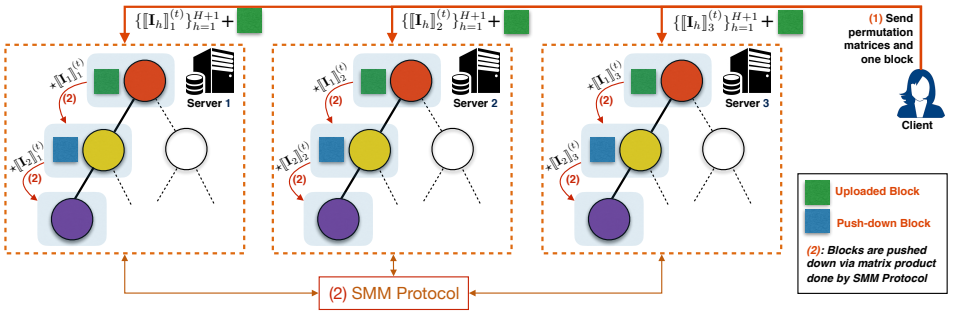


Fig. 13. S^{3}ORAM^O eviction based on [43] using SSS and SMM protocol.

to indicate the block to be picked, while the other columns $\mathbf{I}_h[*][j]$ ($1 \le j \le Z$) is to indicate the block to be moved to or hold at j-th slot of the h-leveled bucket. The data vector $[\![\mathbf{c}_h]\!]$, which will be computed the matrix product with $\mathbf{I}_h$, is of size $Z+1$ containing the holding block ($[\![\mathbf{c}_h[1]]\!]$) and the data from Z slots of the h-leveled bucket ($[\![\mathbf{v}_h[2]]\!] \ldots, [\![\mathbf{v}_h[Z+1]]\!]$). So, the client sets $\mathbf{I}_h[x][Z+1] \leftarrow 1$ to pick the block at slot x (line 15), and $\mathbf{I}_h[1][x] \leftarrow 1$ to drop the holding block to the x-th slot of the h-leveled bucket (line 10). If the currently holding block is moved to the next level (i.e., no pickup/drop-off at this level), the client sets $\mathbf{I}_h[1][Z+1] \leftarrow 1$ (line 13). Similar to S^{3}ORAM^O scheme, the client sets $\mathbf{I}_h[x+1][x] \leftarrow 1$ to keep blocks indexed x in the h-leveled bucket in position (lines 17-20). Finally, the client creates the SSS-shares for such permutation matrices (line 21) and for the block being picked-up in the stash (if any) (line 22), and distributes the shares to the corresponding servers (line 23). Similar to S^{3}ORAM^O, for each level in the eviction path, the servers jointly perform the matrix product between the share of data vector and the share of permutation matrix via local addition and the secure multiplication protocol (line 23), and update the bucket with the newly computed vector (line 25).

3.3.3 Asymptotic cost analysis. Similar to S^{3}ORAM^O, we analyze the cost of S^{3}ORAM^C regarding the block size ($|b|$), number of blocks (N), and statistical security parameter (λ), while other system parameters (e.g., prime field $\mathbb{F}_p$, number of servers ℓ) are treated as constants. S^{3}ORAM^O has the same tree layout, an identical retrieval phase and a similar eviction procedure with the S^{3}ORAM^O scheme. S^{3}ORAM^C only differs from S^{3}ORAM^O in terms of the bucket size parameter (Z) and the eviction frequency, which happens after *every* retrieval instead of A as in S^{3}ORAM^O. S^{3}ORAM^C also incurs at most three blocks (one for retrieval and two for eviction) to be transmitted in each ORAM access. Given $Z = O(1)$ in S^{3}ORAM^C, we summarize the asymptotic cost of S^{3}ORAM^O as follows.

Communication. The client-server communication complexity is $O(|b| + \log N)$. The server-server communication overhead is $O(|b| \cdot \log N)$. To achieve $O(1)$ client-server bandwidth blowup, the minimal block size is $\Omega(\log N)$, which is a factor of λ times smaller than that of S^{3}ORAM^O.

Computation. The server computation is $O(|b| \cdot \log N)$. The client computation complexity is $O(|b| + \log N)$.

Storage. S^{3}ORAM^C layout is a full binary tree of height H, which has a total of $Z \cdot (2^{H+1} - 1)$ slots and can store up to $N \le 2^H$ real blocks. Since $Z = O(1)$, the server storage blowup in S^{3}ORAM^C is $O(1)$ similar to S^{3}ORAM^O asymptotically, but its constant overhead factor is smaller (i.e., 2 *vs.* 8). The client requires to maintain the stash, which costs $O(\lambda)$ to achieve negligible overflow probability. The position map costs $O(N(\log N + \log\log N))$. Therefore, the total client storage is $O(\lambda + N(\log N + \log\log N))$. It is possible to achieve $O(1)$ client storage by storing the stash (via

$\mathsf{S^3ORAM^C}$.EvictAlongPath(v):

Let $(u_1, \ldots, u_{H+1})$ be the (ordered) bucket indexes along the eviction path v from the root to the leaf level

Client:

1: (deepest, deepestIdx) $\leftarrow$ PrepareDeepest(v); target $\leftarrow$ PrepareTarget(v)
2: hold $\leftarrow \perp$, dest $\leftarrow \perp$, $(c_1, \ldots, c_m) \leftarrow 0^{|b|}$
3: **if** target[0] $\neq \perp$ **then** ▷ target[0] and deepestIdx[0] denote the stash component
4: hold $\leftarrow$ deepestIdx[0], dest $\leftarrow$ target[0]
5: $(c_1, \ldots, c_m) \leftarrow S$[hold], S[hold] $\leftarrow \{\}$
6: **for** $h = 1, \ldots, H+1$ **do**
7: Let $\mathbf{I}_h$ be a $(Z+1) \times (Z+1)$ matrix, set $\mathbf{I}_h[*, *] \leftarrow 0$.
8: **if** hold $\neq \perp$ **then**
9: **if** i = dest **then** ▷ Drop the holding block to this level
10: $\mathbf{I}_h[1][x] \leftarrow 1$ where x is the index of an empty slot in the bucket $\mathsf{T}[u_h]$
11: hold $\leftarrow \perp$, dest $\leftarrow \perp$
12: **else** ▷ Move the holding block to the next level
13: $\mathbf{I}_h[1][Z+1] \leftarrow 1$
14: **if** target[i] $\neq \perp$ **then** ▷ Pick a block at this level
15: $\mathbf{I}_h[x][Z+1] \leftarrow 1$ where $x \leftarrow$ deepestIdx[h]
16: hold $\leftarrow x$, dest $\leftarrow$ target[i]
17: **for** each real block id in $\mathsf{T}[u_h]$ **do** ▷ Hold the position of other real blocks at this level
18: $x \leftarrow$ pm[id].pIdx mod Z
19: **if** $x \neq$ deepestIdx[h] **then**
20: $\mathbf{I}_h[x+1][x] \leftarrow 1$
21: $[\![\mathbf{I}_h[*, *]]\!]_1^{(t)}, \ldots, [\![\mathbf{I}_h[*, *]]\!]_\ell^{(t)} \leftarrow$ SSS.Create($\mathbf{I}_h[*, *]$, t) for $1 \le h \le H+1$
22: $([\![c_j]\!]_1^{(t)}, \ldots, [\![c_j]\!]_\ell^{(t)}) \leftarrow$ SSS.Create(c_j, t) for $1 \le j \le m$
23: Send $\left(\langle [\![c_1]\!]_i^{(t)}, \ldots, [\![c_m]\!]_i^{(t)} \rangle, \langle [\![\mathbf{I}_1]\!]_i^{(t)}, \ldots, [\![\mathbf{I}_{H+1}]\!]_i^{(t)} \rangle\right)$ to $\mathcal{S}_i$, for $1 \le i \le \ell$

Server: each $\mathcal{S} \in \{\mathcal{S}_1, \ldots, \mathcal{S}_\ell\}$ receiving $(\langle [\![c_1]\!], \ldots, [\![c_m]\!] \rangle, \langle [\![\mathbf{I}_1]\!], \ldots, [\![\mathbf{I}_{H+1}]\!] \rangle)$ **do**

18: $[\![x]\!]_j \leftarrow [\![c]\!]_j$ for $1 \le i \le m$
19: **for** $h = 1, \ldots, H+1$ **do**
20: **for** $j = 1, \ldots, m$ **do**
21: Let $[\![\mathbf{c}_{h,j}]\!]$ be a Z-dimensional vector containing j-th chunks of bucket $[\![\mathsf{T}[u_h]]\!]$
22: $[\![\hat{\mathbf{c}}_{h,j}]\!] := ([\![x_j]\!], [\![\mathbf{c}_{h,j}]\!])$ ▷ Concatenate $[\![x_j]\!]$ with $[\![\mathbf{c}_{h,j}]\!]$ resulting in a $(Z+1)$-dimensional vector
23: $[\![\hat{\mathbf{c}}'_{h,j}]\!] \leftarrow [\![\hat{\mathbf{c}}_{h,j}]\!] \star [\![\mathbf{I}_h]\!]$
24: $([\![\mathbf{c}'_{h,j}]\!], [\![x_j]\!]) := [\![\hat{\mathbf{c}}'_{h,j}]\!]$ ▷ Assign the last component of vector $[\![\hat{\mathbf{c}}']\!]_{h,j}$ to $[\![x_j]\!]$
25: Update j-th chunks of bucket $[\![\mathsf{T}[u_h]]\!]$ with $[\![\mathbf{c}'_{h,j}]\!]$

Fig. 14. $\mathsf{S^3ORAM^C}$ eviction protocol based on [43]. The details of PrepareDeepest(v) and PrepareTarget(v) subroutines (line 1) are presented in Figure 21 in Appendix.

SSS shares) on the servers and using the recursion technique to keep the position map in smaller ORAMs. However, this will significantly increase the computation and communication overhead for oblivious access to the stash and the position map, respectively.

4 SECURITY ANALYSIS

In this section, we analyze the security of two $\mathsf{S^3ORAM}$ schemes as follows.

$\mathsf{S^3ORAM^O}$. It follows the Triplet Eviction strategy originally proposed in Onion-ORAM [12]. Therefore, it achieves the same failure probability with Onion-ORAM. We refer the reader to [12] for the detail of the proof.

Lemma 3 ($\mathsf{S^3ORAM^O}$ Bucket Overflow Probability). If $Z \ge A$ and $N \le A \cdot 2^{H-1}$, the probability that a bucket overflows after a Triplet Eviction operation is bounded by $e^{-\frac{(2Z-A)^2}{6A}}$, where $Z = A = \Theta(\lambda)$.

Proof. We refer the reader to [12]. □

It is easy to see that Lemma 3 implies the following fact.

Corollary 1 (Non-leaf Destination Bucket Load). All non-leaf destination buckets are always empty after the Triplet Eviction takes place, except with a negligible probability.

We present the main security of S^{3}ORAM$^{\mathrm{O}}$ as follows.

Theorem 1 (S^{3}ORAM^O Security). S^{3}ORAM^O is correct and information-theoretically (statistically) t-secure by Definition 2.

Proof. S^{3}ORAM$^{\mathrm{O}}$ is correct iff (*i*) the S^{3}ORAM$^{\mathrm{O}}$.Retrieve($\cdot$) protocol returns the correct value of the retrieved block and (*ii*) the S^{3}ORAM$^{\mathrm{O}}$.Evict($\cdot$) function is consistent.

• *Correctness of S^{3}ORAM^O.Retrieve($\cdot$).* 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). The share of the PIR 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 retrieval 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 hidden by a random $2t$-degree polynomial. Our system model presented in §2 follows this and, therefore, the client always computes the correct value of chunk c_t by $c_t \leftarrow$ SSS.Recover($[\![c_t]\!]_1^{(2t)}, \ldots, [\![c_t]\!]_\ell^{(2t)}, 2t$). Since all chunks of b are correctly computed, b is properly retrieved with the probability 1.

• *Consistency of S^{3}ORAM^O.Evict($\cdot$)* Corollary 1 shows that the root bucket is empty after the triplet eviction. The client writes the retrieved block to an empty slot in the root bucket sequentially (line 3, Figure 8). Since $Z \ge A$, the root always has enough empty slots to contain all the blocks to be written before the triplet eviction happens, thereby avoiding the overwritten and inconsistency issues. After A accesses, the client executes the triplet eviction algorithm (Figure 10) to move blocks from upper levels (e.g., root) to deeper levels (e.g., leaf). Corollary 1 also shows that non-leaf sibling buckets are empty due to previous triplet evictions and, therefore, they can contain all data moved from their source bucket without creating any inconsistency issue. Real blocks from source buckets are moved to destination buckets via matrix products. These computations are correct due to homomorphic properties of two-share addition and multiplication offered by SSS and the SMM protocol, respectively, which were proven correct in [16].

We now prove the security of S^{3}ORAM$^{\mathrm{O}}$ as follows.

Given a request sequence $\mathbf{x}$ of length q, where $x_j = (\mathrm{op}_j, \mathrm{id}_j, \mathrm{data}_j)$ as in Definition 2, let S^{3}ORAM$^{\mathrm{O}}{}_i(\mathbf{x})$ be the S^{3}ORAM$^{\mathrm{O}}$ client's sequence of interactions with the server $\mathcal{S}_i$ including a sequence of *retrievals* (Figure 7), *write-to-root* (line 3, Figure 8) and *triplet eviction* operations (Figure 10). We have that the *write-to-root* operation is deterministic, which is performed right after the retrieval. In this operation, the previously retrieved block is written to a publicly known slot in the root bucket as shown above. The *triplet eviction* is also deterministic, which is performed after every A successive accesses regardless of any data being requested. Since all these operations (i.e.,

retrieval, *write-to-root*, *triplet eviction*) are independent of each other, they can be considered as separate sequences observed by $\mathcal{S}_i$ in $\text{S}^3\text{ORAM}^{\text{O}}{}_i(\mathbf{x})$ as follows

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

where $\vec{R}_i(\tilde{\mathbf{x}})$, $\vec{W}_i(\tilde{\mathbf{x}})$ and $\vec{E}(\bar{\mathbf{x}})$ denote the *retrieval*, *write-to-bucket* and *triplet eviction* sequences, given a 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

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

We show that for any two access sequences $\mathbf{x}$ and $\mathbf{x}'$ of the same length (i.e., $|\mathbf{x}| = |\mathbf{x}'|$), 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 (line 4, Figure 7) and all data generated by the SSS-based PIR scheme (lines 5-7).

The access pattern of $\text{S}^3\text{ORAM}^{\text{O}}$ is identical to all other secure tree-based ORAM schemes. Specifically, each block in $\text{S}^3\text{ORAM}^{\text{O}}$ is assigned to a leaf bucket selected randomly and independently from each other. Once a block is accessed, it is assigned to a new bucket leaf selected randomly and independently. Such random assignment along with the selected bucket size parameter (Z) may result in the bucket(s) in the $\text{S}^3\text{ORAM}^{\text{O}}$ tree being overflowed with a negligible probability thereby, impacts the security (see Lemma 3). 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 $\text{S}^3\text{ORAM}^{\text{O}}$ retrieval as follows. For each retrieval, the client sends to the servers PIR queries generated by PIR.CreateQuery algorithm. Such queries are SSS shares and, therefore, is t-private. The inner product is also t-private due to Lemma 1 with addition and partial multiplicative homomorphic properties by Eq. 3 and Eq. 1, respectively. So, any data generated in $\text{S}^3\text{ORAM}^{\text{O}}$ 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 $\text{S}^3\text{ORAM}^{\text{O}}$ 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 the sequential order and, therefore, the write pattern is deterministic and public. Such written data are SSS-shared with new random polynomials so that they are t-private. Therefore, the *write-to-root* transcripts are identically distributed.

• *Triplet eviction transcripts:* the access patterns of $\{E_{i\in\mathcal{I}}^{(j)}\}$ and $\{E_{i\in\mathcal{I}}^{(j')}\}$ are public because the triplet eviction is deterministic, which follows reverse lexicographical order like Onion-ORAM (e.g., [12]), We show that data generated in such triplet evictions are identically distributed as follows. For each triplet eviction, the client sends $(H+1)$ permutation matrices, which are SSS-shares and, therefore, they are all t-private and uniformly distributed. Data in sibling buckets are t-private and uniformly distributed because they are merely copied from source buckets deterministically

(line 19, Figure 10). The matrix product (line 22, Figure 10) is also t-private due to the security of SMM protocol by Lemma 2 . Therefore, given two request sequences $\mathbf{x}$, $\mathbf{y}$ with $|\mathbf{x}| = |\mathbf{y}|$, the corresponding deterministic triplet eviction sequences observed by $\{\mathcal{S}_{i\in I}\}$ are

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

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

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

• *Final indistinguishability argument:* Given any data request sequence, S^{3}ORAM$^{\mathrm{O}}$ generates (*i*) access patterns statistically indistinguishable from a random request sequence of the same length, and (*ii*) identically (uniform) distributed data in the presence of up to t colluding servers. This indicates that S^{3}ORAM$^{\mathrm{O}}$ scheme achieves information-theoretic statistical t-security according to Definition 2. □

S^{3}ORAM$^{\mathrm{C}}$. It follows the Circuit-ORAM eviction strategy [1] so that it inherits the same failure probability as Circuit-ORAM as follows.

Lemma 4 (S^{3}ORAM^C Stash Overflow Probability). Let the bucket size $Z \ge 2$. Let st(*S^{3}ORAM^C*[s]) *be a random variable denoting the stash size of S^{3}ORAM^C scheme after an access sequence s. Then, for any access sequence s,* $\Pr[\mathrm{st}(S^3ORAM^C[s]) \ge R] \le 14 \cdot e^{-R}$.

Proof. We refer the reader to [43]. □

The security of S^{3}ORAM$^{\mathrm{C}}$ is given in the following theorem.

Theorem 2 (S^{3}ORAM^C Security). S^{3}ORAM^C is correct and information-theoretically (statistically) t-secure by Definition 2.

Proof. The correctness and security proof of S^{3}ORAM$^{\mathrm{C}}$ can be easily derived from that of S^{3}ORAM$^{\mathrm{O}}$ scheme so that we will not present it in detail due to the space constraint and the significant overlap with the proof of Theorem 1. Intuitively, S^{3}ORAM$^{\mathrm{C}}$ leverages the same principles as S^{3}ORAM$^{\mathrm{O}}$, i.e., SSS-based PIR scheme and permutation matrix, to implement the retrieval and eviction phases, which were proven correct and consistent due to homomorphic properties of SSS and SMM protocol. We also proved that all the data generated by these operations are t-private. The access pattern in S^{3}ORAM$^{\mathrm{C}}$ is *statistically* indistinguishable due to its negligible stash overflow probability by Lemma 4. All these properties indicate that S^{3}ORAM$^{\mathrm{C}}$ scheme achieves information-theoretic statistical t-security by Definition 2. □

5 GENERALIZATION OF S^{3}ORAM OVER k-ARY TREE

It is possible to execute ORAM over a general k-ary tree layout to achieve a sub-logarithmic asymptotic overhead (i.e., $O(\log_k N)$, where k is a free parameter). Our proposed S^{3}ORAM schemes can also be easily extended to work over a general k-ary ORAM layout. However, we later show that increasing the value of k does not bring much benefit to the actual performance of tree-based ORAM schemes. We present S^{3}ORAM schemes on the general k-ary tree layout, and then provide the analytical analysis to show that their cost achieves the best at $k \in \{2, 3\}$ as follows.

k-ary S^{3}ORAM$^{\mathrm{O}}$. We can leverage the concepts of SSS homomorphic computation and the permutation matrix presented in §3.2 to implement the eviction strategy in [1], which is the generalization of the Triplet Eviction used in S^{3}ORAM$^{\mathrm{O}}$. Generally speaking, this strategy requires

to organize each bucket in the S^3ORAM^O k-ary tree layout into k slides, each being of size as a function of the security parameter (i.e., $O(\lambda)$). In other words, $Z = O(k \cdot \lambda)$. Each bucket at the leaf level is connected with a so-called auxiliary bucket of size $O(\lambda)$. The eviction path for k-ary tree is determined by modifying Eq. 6 to output the order-reversal of base-k digits instead of the binary string. Once the eviction path is determined, we travel from the root to the leaf and obliviously move all blocks from the (non-leaf) source bucket to a deterministic slide of *all* of its children. At the leaf level, we obliviously move all blocks from the leaf bucket to its corresponding auxiliary bucket. All these oblivious moves can be implemented using the SSS matrix product principle described in §3.2. Notice that in this context, the retrieval phase in S^3ORAM^O scheme remains unchanged.

k-ary S^3ORAM^C. S^3ORAM^C scheme in §3.3 supports the k-ary tree layout naturally without modifying the retrieval and the eviction subroutines. We only need to change Eq. 6 as similar to the k-ary S^3ORAM^O scheme as discussed above to get the eviction path in the k-ary tree layout. It also only requires to adjust the bucket size parameter (Z) to be a function of the tree degree to achieve a negligible stash overflow probability. In other words, $Z = O(k)$ for the statistical security.

Cost analysis. We now treat k as a parameter in the asymptotic cost. The cost of k-ary S^3ORAM^O can be derived from §3.2.3, where the bucket size parameter (Z) in this context is $O(\kappa \cdot \lambda)$ instead of $O(\lambda)$. The k-ary S^3ORAM^O layout is a tree of height $O(\log_k N)$. The eviction in each level move all blocks from the source bucket of size $O(\kappa \cdot \lambda)$ to a slide (sized $O(\lambda)$) of its k children buckets. Thus, the (amortized) client-server and server-server *bandwidth* is $O(|b| + \lambda \cdot k \cdot \log_k N)$ and $O(|b| \cdot \lambda \cdot k \cdot \log_k N)$, respectively. The (amortized) server and client *computation* is $O(|b| \cdot \lambda \cdot k \cdot \log_k N)$ and $O(|b| + k \cdot \log_k N)$, respectively.

Similarly, we can easily derive the cost of k-ary S^3ORAM^C scheme from §3.3.3 , where the bucket size now becomes $Z = O(k)$. So, the client-server- and server-server-bandwidth of k-ary S^3ORAM^C are $O(|b| + k \cdot \log_k N)$ and $O(|b| \cdot k \cdot \log_k N)$, respectively. The server- and client-computation are $O(|b| \cdot k \cdot \log_k N)$ and $O(|b| + k \cdot \log_k N)$, respectively.

So, given that $|b|, \lambda, N$ are unchanged in this context, the computation/bandwidth overhead of k-ary S^3ORAM schemes can be written as a function of k as

$$f(k) = \alpha + \beta \cdot k \cdot \log_k N = \alpha + \beta \cdot \frac{k}{\ln k} \cdot \ln N \tag{8}$$

where $\alpha \in \{0, |b|\}, \beta \in \{1, |b|, \lambda, \lambda \cdot |b|\}$. Since $k \geq 2$ and $k \in \mathbb{N}$, it is easy to see that $f(k)$ is minimal at $k = 3$.

For $k = 2$, our S^3ORAM^O using the Triplet Eviction outperforms a constant factor of two compared with using the generalization strategy in [1]. This is because in this case, each source bucket only has one sibling bucket. Due to Eq. 6, once a bucket is being treated as the sibling bucket, it will be later considered as the destination bucket before being treated as the sibling bucket again. Meanwhile, once the (non-leaf) bucket is treated the destination bucket, it is always guaranteed to be empty after the eviction (see Corollary 1). In other words, the bucket is always empty before being considered as the sibling bucket and, therefore, given a fixed size of $O(\lambda)$, it always has enough slots to keep the expected load below its capacity. The eviction in [1] does not exploit this special role-switching when $k = 2$, but focuses on the general case for any $k > 2$, where one bucket must serve as the sibling bucket $k - 1$ times before being empty. As a result, each bucket must have k slides each being of size $O(\lambda)$, and the eviction only touches one slide of the bucket to achieve the sub-logarithmic overhead. Therefore, when $k = 2$, its (eviction and retrieval) overhead is doubled compared with that of the Triplet Eviction. Moreover, given the fact that the generalization eviction with $k = 3$ only gains 6% improvement over $k = 2$, while its $k = 2$ case is

two times less efficient than the Triplet Eviction as shown above, we conclude that the S^{3}ORAM^O achieves the best performance with $k = 2$ and the Triplet Eviction strategy.

For S^{3}ORAM^C scheme, it achieves the best performance at $k = 3$ according to Eq. 8. This is because it supports k-ary layout naturally without modifying eviction and retrieval subroutine but only adjusting the bucket size parameter. We further demonstrate empirical results to support such analytical analyses in §7.2.4.

6 IMPLEMENTATION

We fully implemented two S^{3}ORAM schemes in C++ consisting of roughly 5,000 lines of code. We used two external libraries in our implementation: (1) The Shoup's NTL library v9.10.0[4] for the pseudo-random number generation and arithmetic operations due to its low-level optimization for modular multiplication and cross product functions; (ii) the ZeroMQ library[5] for the network communication. Our implementation supports parallelization via multi-threading to take full advantage of multi-core CPUs at the server side. We also implemented k-ary tree layout generalization for both S^{3}ORAM schemes. The implementation of our S^{3}ORAM framework is publicly available at

https://github.com/thanghoang/S3ORAM

7 EXPERIMENTAL EVALUATION

In this section, we evaluate the performance of S^{3}ORAM framework in comparison with its counterparts on commodity cloud environments. Our main evaluation metric is the end-to-end delay and we seek to answer the following questions.

- How is the end-to-end delay of S^{3}ORAM schemes compared with recent ORAM schemes for different database and block sizes? (§7.2.1)
- What factors impact the overall delay of S^{3}ORAM schemes? (§7.2.2)
- In which context S^{3}ORAM will be outperformed by other ORAM schemes? (§7.2.3)
- Does increasing the tree degree help enhance the end-to-end delay of S^{3}ORAM in practice? (§7.2.4)
- What is the storage overhead of S^{3}ORAM schemes? (§7.2.5)

We first describe the configuration and methodology to conduct our experiments as follows.

7.1 Configuration and Methodology

Hardware setting. We used a 2015 Macbook Pro laptop as the client, which was equipped with an Intel Core i5-5287U CPU @ 2.90GHz and 16 GB RAM. On 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 1TB SSD.

Network setting. We located three servers to be geographically close to each other (same region) as well as to our client machine, which results in the network latency between them being approximately 15 ms. The servers were connected to each other via a dedicated network whose throughput for both download and upload is approximately 1 Gbps. The client used Wi-Fi connecting to the Internet via a home data plan, which offers the latency of 20 ms and the download/upload throughput of 55/6 Mbps to the servers.

Database size. We evaluated the performance of all ORAM schemes with a randomly generated database of size from 0.5 GB to 40 GB and block sizes from 4 KB to 1024 KB.

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

Selected S^{3}ORAM counterparts. We selected Path-ORAM [41] and Onion-ORAM [12] as the main counterparts of S^{3}ORAM framework since the former is the most optimal $O(\log N)$-bandwidth ORAM (without server computation) while the latter achieves $O(1)$ bandwidth blowup (with server computation). We also chose Ring-ORAM [34] as it is an efficient $O(\log N)$-bandwidth ORAM scheme with server computation. We consider all ORAM schemes (including our S^{3}ORAM framework) under their *non-recursive form*, where the position map is stored locally at the client. This is because storing the position map at the server will incur $O(\log N)$ number of communication rounds of accessing $O(\log N)$ smaller S^{3}ORAM, which may result in high overhead. In practice, it is likely that the position map is small enough to be stored locally at the client. Moreover, since S^{3}ORAM is only secure in the *semi-honest setting*, we only compared its performance with the semi-honest version of Path-ORAM, Ring-ORAM and Onion-ORAM. We did not consider alternatives that (*i*) failed to achieve $O(1)$ client communication blowup but incurred more delay (e.g., [11, 27]), (*ii*) were shown to be insecure (e.g., [28, 30]), or (*iii*) incurred more cost than the selected ORAM counterparts above regarding to our configuration and experimental settings (e.g., [3]). We also did not explicitly compare the performance of S^{3}ORAM against the multi-server ORAM scheme in [38] because of the major difference in terms of client block storage between the two schemes ($O(1)$ *vs.* $O(\sqrt{N})$). Given a very large outsourced database, the storage required by [38] might not be suitable for resource-limited devices such as a mobile phone. Moreover, if $O(\sqrt{N})$ block storage is acceptable, then the lower bound in [1] might imply a better ORAM strategy than our S^{3}ORAM schemes, in which leveraging only PIR technique suffices to achieve $O(1)$ client bandwidth blowup.

Evaluation methodology. We present the parameter choice and methodology to measure the performance of S^{3}ORAM schemes and their counterparts as follows.

- *S^{3}ORAM*: For the S^{3}ORAM$^{\mathrm{O}}$ scheme, we selected the bucket size $Z = 74$ and $A = Z/2 = 37$ and to achieve the negligible overflow probability of 2^{-80} by Lemma 3. We measured the cost for each S^{3}ORAM$^{\mathrm{O}}$ access as the retrieval delay plus the write-to-root delay plus the *amortized* delay of the eviction operation. For the S^{3}ORAM$^{\mathrm{C}}$ scheme, we selected the bucket size $Z = 2$ suggested in [43] for the negligible stash overflow probability by Lemma 4. We also investigated the performance of S^{3}ORAM framework with k-ary structure, where $k > 2$. In this case, we fixed the database size and varied the tree height (H) and the bucket size (Z) parameters. (see §7.2.4 for the detailed configuration).
- *Path-ORAM*: We selected the bucket size $Z = 4$ to achieve the negligible stash overflow probability of 2^{-80}. We measured the delay of Path-ORAM as the time to download and upload $4 \cdot \log_2 N$ blocks plus the delay of IND-CPA decryption and re-encryption of these blocks at the client. We used `libtomcrypt`[6] to implement AES-CTR as the IND-CPA encryption.
- *Ring-ORAM*: We selected Ring-ORAM parameters (i.e., $Z = 16$, $S = 25$ and $A = 20$) as stated in [34] for a negligible stash overflow probability of 2^{-80}. We measured the delay of Ring-ORAM as the total time of (*i*) one block transmission, (*ii*) XOR and IND-CPA encryption/decryption operations at the client, (*iii*) XOR operations at the server and (*iv*) the amortized cost of eviction and early shuffles based on the formula $(H + 1)(2Z + S)/A \cdot (1 + \mathsf{PoissCDF}(S, A))$ given in [34].
- *Onion-ORAM*: We selected the size of RSA modulus to be 1024 bits for AHE according to [2]. We selected the bucket size and the eviction frequency of Onion-ORAM as $Z = 74$ and $A = Z/2 = 37$ for the negligible bucket overflow probability of 2^{-80}. We measured the overall delay of Onion-ORAM as the time to (1) perform homomorphic computations at the client and server and (2)

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

Table 3. The amount of data to be sent by the client and processed by the server(s) in the retrieval and eviction phases of S^{3}ORAM^O and S^{3}ORAM^C schemes.

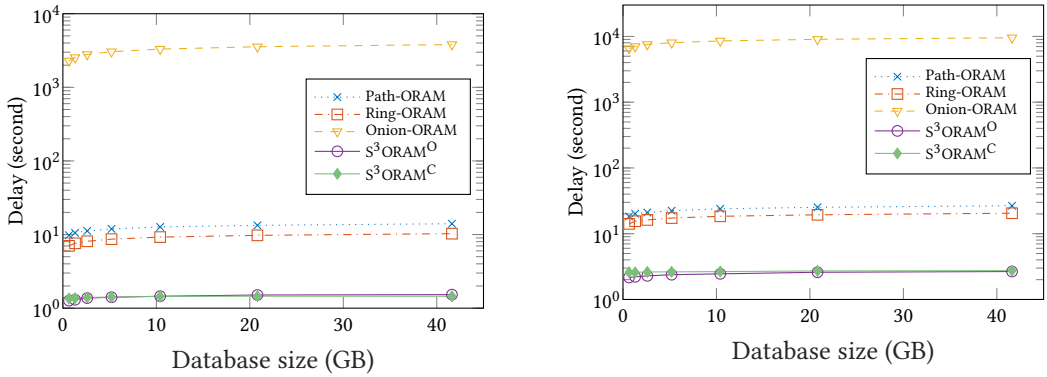
# Blocks	Retrieval Phase				Eviction Phase			
	Query Size (KB)		# Computed Blocks		Permutation Matrix Size (KB)		# Computed Blocks	
	S^3ORAM^O	S^3ORAM^C	S^3ORAM^O	S^3ORAM^C	S^3ORAM^O	S^3ORAM^C	S^3ORAM^O	S^3ORAM^C
10^3	4.05	0.17	518	20	598.93	1.55	76,664	198
10^4	6.36	0.23	814	28	941.19	2.11	120,472	270
10^5	8.09	0.28	1,036	34	1,197.88	2.53	153,328	324
10^6	9.82	0.33	1,258	40	1,454.56	2.95	186,184	378
10^7	12.14	0.39	1,554	48	1,796.81	3.52	229,992	450
10^8	13.88	0.44	1,776	54	2,053.50	3.94	262,848	504
10^9	15.61	0.48	1,998	60	2,310.19	4.36	295,704	558

transfer $O(1)$ blocks and PIR queries, plus the amortized cost of eviction operation. Since Onion-ORAM is extremely computationally costly, measuring its delay even on a medium database takes an insurmountable amount of time. Therefore, we only measured its delay on a small database (i.e., 1 MB) first, and then extrapolate the delay for larger database sizes.

7.2 Experimental Results

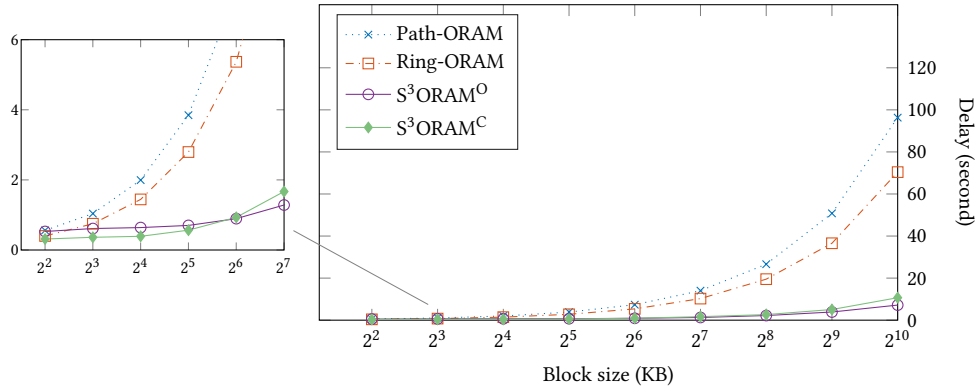
7.2.1 End-to-end delay. We first present the analytical communication and computation overhead of S^{3}ORAM schemes with databases containing a various number of data blocks in Table 3. We can see that the size of the retrieval query and permutation matrices by the client as well as the amount of the data to be computed by the server are much lower than S^{3}ORAM^O for each access. This is mainly because S^{3}ORAM^C has a much smaller bucket size than S^{3}ORAM^O (i.e., 2 *vs.* 74). However, the eviction in S^{3}ORAM^O is only performed after every 37 accesses compared with one in S^{3}ORAM^C scheme. In fact, this accumulative strategy allows S^{3}ORAM^O to be less impacted by the network congestion control (i.e., TCP slow-start) and the client-server/ server-server communication latency than S^{3}ORAM^C. Moreover, S^{3}ORAM^O incurs only one block to be uploaded per access compared with two in S^{3}ORAM^O scheme. All these factors result in the *amortized* end-to-end delay of S^{3}ORAM^O scheme being comparable with the actual delay of S^{3}ORAM^C as shown in Figure 15, even though its analytical overhead looks worse than that of S^{3}ORAM^C. We also show in Figure 15 the simulated delay of S^{3}ORAM counterparts with different database sizes (from 0.5 to 40 GB) and block sizes (128 KB and 256 KB). Both S^{3}ORAM schemes took only 1.3-1.4 (*resp.* 2.1-2.7) seconds to access a 128 KB (*resp.* 256 KB) block in the database of size up to 40 GB. This resulted in S^{3}ORAM schemes being approximately 9.3 and 6.4 times faster than Path-ORAM and Ring-ORAM, where they took 7-14 (*resp.* 14-26) seconds for each 128 KB (*resp.* 256 KB) block access. Compared with Onion-ORAM, our S^{3}ORAM schemes were three orders of magnitude faster. This is mainly due to the fact that S^{3}ORAM schemes only rely on simple arithmetic operations (e.g., modular addition/ multiplication), while Onion-ORAM leverages Partially/Fully HE (see §7.2.2). One might also observe from Figure 15 that choosing a larger block size has a small impact on the delay of S^{3}ORAM schemes. This is clearly illustrated in Figure 16, where we present the impact of block size on the end-to-end delay of S^{3}ORAM schemes compared with their counterparts. Given any block size ranging from 4 KB to 1024 KB, S^{3}ORAM schemes always maintain a constant factor of 9.3 and 6.4 times faster than Ring-ORAM and Path-ORAM, respectively. This presents an advantage to S^{3}ORAM schemes over their counterparts for applications requiring large block sizes such as image or video storage services.

7.2.2 S^{3}ORAM cost breakdown analysis. In this section, we dissect the overall delay of S^{3}ORAM to explore the factors that contribute the most to the total delay. Figure 17 shows the detailed cost



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

Fig. 15. End-to-end delay of S^{3}ORAM and its counterparts on a laptop with home network.



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

Fig. 16. End-to-end for varying block sizes for a 40GB DB.

factors of two S^{3}ORAM schemes according to 0.5-40 GB DB with 128 KB blocks. There are five factors that affect the overall delay of S^{3}ORAM schemes as follows.

(1) *Client computation:* In both S^{3}ORAM schemes, the client computed the SSS shares of the retrieval query and the permutation matrices, recovered the requested block and re-shared the block with SSS. All these incurred only some modular addition and multiplication operations. These computations are extremely lightweight so that the client computation contributed only a minimal amount to the total delay (i.e., $< 1\%$), which is hard to observe in both Figure 17a and Figure 17b.

(2) *Server computation:* In both S^{3}ORAM schemes, the servers computed the ORAM tree data with the retrieval query via the dot product and with permutation matrices via the matrix product, which also incurred a series of modular addition and multiplication operations. However, unlike the client computation, the cost of these operations at the server-side depends on the block size. As a result, the server computation contributed a higher amount to the total delay (i.e., 7-11%) than the client computation. Compared between two S^{3}ORAM schemes, we can see that S^{3}ORAM^O had a higher server computation delay than S^{3}ORAM^C

(a) S^{3}ORAM^C (b) S^{3}ORAM^O

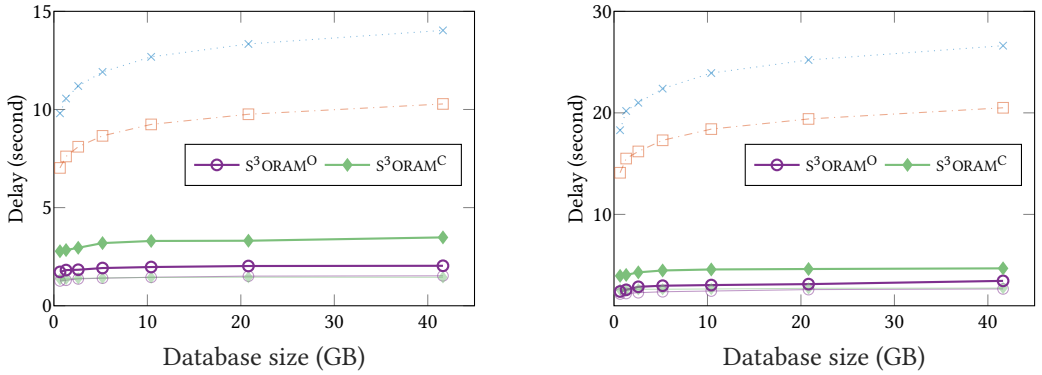
Fig. 17. Detailed cost breakdown of S^{3}ORAM on a laptop with home network.

scheme. This is because the block size of S^{3}ORAM^O is much larger than S^{3}ORAM^C (i.e., 74 *vs.* 2), which significantly impacts the SSS-based PIR computation in the retrieval phase.

(3) *Client-server communication:* In both S^{3}ORAM schemes, this operation contributed the most to the total delay (over 90%). For each S^{3}ORAM access, the client downloaded one block from the servers and uploaded 1-2 blocks along with one retrieval query and some permutation matrices. We can observe from Figure 17 that the time to upload the retrieval query and permutation matrices (*yellow-patterned green bars*) was much faster than the time to download and upload a 128 KB data block (*unpatterned green bars*). This clearly reflects the theoretical insight of S^{3}ORAM schemes, where the client communication overhead is constant and mostly dominated by the data block with the poly-logarithmic size. We can also observe that S^{3}ORAM^O (Figure 17b) took a longer time to transmit the retrieval query and permutation matrices than S^{3}ORAM^C (Figure 17a). This is because the bucket size parameter in S^{3}ORAM^O scheme is much larger than in S^{3}ORAM^C as explained above, which impacts the size of the retrieval vector and the eviction matrices. On the other hand, the block transmission time in S^{3}ORAM^C was doubly slower than in S^{3}ORAM^C. This is because the eviction in S^{3}ORAM^C requires to transmit *two* blocks for each access, compared with only *one* in S^{3}ORAM^O.

(4) *I/O access:* Due to the cache miss issue and the infrastructure of the selected Amazon EC2 instances (i.e., `c4.4xlarge`), the disk I/O access caused a considerable delay especially in S^{3}ORAM^O scheme. Specifically, we stored the S^{3}ORAM tree in a network storage unit called "Elastic Block Storage" (EBS), which was connected to Amazon EC2 computing unit with a maximum throughput of 160 MB/s. This resulted in the I/O access being limited by this throughput, and therefore, causing a high delay. To reduce the I/O access overhead, one solution is to store the S^{3}ORAM tree structure on a local storage unit with high throughput (e.g., NVMe). Another solution is to apply a caching strategy, where h-top levels of the S^{3}ORAM tree are stored directly on RAM. As explained above, S^{3}ORAM^O has a larger bucket size than S^{3}ORAM^C so that its reported I/O delay was higher than S^{3}ORAM^C.

(5) *Server-server communication:* This overhead was caused by the SMM protocol when the servers performed the matrix product operation in the eviction phase. In S^{3}ORAM^O scheme, the reported communication delay between the servers was very low, and significantly faster than in S^{3}ORAM^C scheme. This is because of the amortization in S^{3}ORAM^O scheme, where the eviction was performed after every $A = 37$ subsequent retrievals. This context allowed



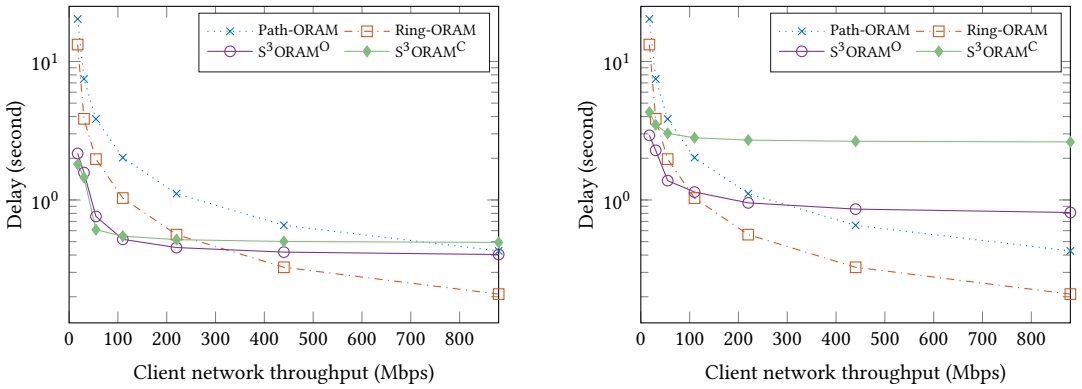
(a) Block size = 128 KB (b) Block size = 256 KB

Fig. 18. The delay of S^{3}ORAM schemes when servers were geographically distant from each other. The blurred plots indicate the performance of S^{3}ORAM and their counterparts in the original setting.

the network latency (i.e., 15 ms) to be amortized and minimized the impact of the TCP slow start scheme. In S^{3}ORAM^C, the eviction must be performed right after each retrieval so that its reported delay was significantly impacted by those factors.

7.2.3 The impacts of network quality. We first investigated the impact of inter-server network quality on the performance of S^{3}ORAM schemes. In this setting, we set up three Amazon EC2 servers to be geographically distant to each other (in the form of a triangle between California, Ohio and Central Canada regions). The average network round-trip latency and throughput between the servers were 78 ms and 295 Mbps, respectively. The round-trip latency between the client and the farthest server was 80 ms, while the client throughput to all servers remained unchanged (i.e., 55/6 Mbps of download/upload speed). Figure 18 presents the delay of S^{3}ORAM in this setting compared with the previous one. We can see that S^{3}ORAM schemes performed 0.3–2 s (2× at most) slower than in the original setting, where all servers were in the same region and close to the client. This slowdown is mostly due to (*i*) the higher latency and lower throughput of the inter-server network link and (*ii*) the latency when the client communicates with the farthest server. However, as shown in Figure 18, S^{3}ORAM still outperformed the performance of Path-ORAM and Ring-ORAM in the original setting (i.e., server was placed close to the client). This is because the server-server communication only contributed a small fraction in the total delay, especially in S^{3}ORAM^O scheme (due to the amortization) as already analyzed in §7.2.2. On the other hand, S^{3}ORAM incurred only one communication round between the client and the servers so that the impact of the client's high round-trip latency was minimal. Moreover, the client throughput remained unchanged and therefore, it did not impact much on the delay of S^{3}ORAM schemes in this context. We observed that S^{3}ORAM^C was more impacted by the inter-server high network round-trip latency than S^{3}ORAM^O. This is because S^{3}ORAM^C performed eviction right after each access, where the servers communicated with each other in $O(\log N)$ rounds. Meanwhile, these rounds were performed once every $A = 37$ accesses in S^{3}ORAM^O and therefore, their total latency was amortized.

Given that the low network quality at both client and inter-server sides did not impact much on the delay of S^{3}ORAM schemes, we now show that if the client can have a high-speed network setting, our S^{3}ORAM framework might no longer be an ideal choice. We conducted an experiment to demonstrate that ORAM schemes featuring $O(\log N)$ bandwidth overhead are better than S^{3}ORAM



(a) 1 Gbps inter-server network throughput (b) 295 Mbps inter-server network throughput

Fig. 19. The impact of client network throughput.

after a certain threshold of network bandwidth. Figure 19 presents simulated performance of S^{3}ORAM schemes and their counterparts with different client network bandwidth settings regarding 40 GB database containing 128 KB blocks. With 1 Gbps inter-server network throughput (servers were at the same region), Path-ORAM and Ring-ORAM surpassed S^{3}ORAM schemes for a client network throughput of approximately 720 Mbps and 380 Mbps, respectively (Figure 19a). Given servers were set up geographically distant to each other, the corresponding numbers were 80-300 Mbps and 50-100 Mbps (Figure 19b). This is because Path-ORAM and Ring-ORAM feature $O(\log N)$ bandwidth overhead so that they receive a more benefit from the high network speed. On the other hand, S^{3}ORAM schemes feature $O(1)$ bandwidth overhead and therefore, get less benefit.

7.2.4 The Impact of k-ary tree layout. We performed an empirical analysis to confirm our finding in §5 that increasing the degree of the ORAM tree receives very little benefit if not worse than the default setting (i.e., binary tree).

Figure 20 presents the actual end-to-end delay of S^{3}ORAM schemes with varied tree degrees under the fixed 1TB DB with 128 KB blocks configuration. We can see that the actual delay of k-ary S^{3}ORAM schemes likely matched with the expected overhead (*the dash-dotted line*). As discussed, the performance of S^{3}ORAM^O following the generalized eviction in [1] achieved the best at $k = 3$ (*the solid purple line*). Remark that for the special case where $k = 2$, such generalized eviction did

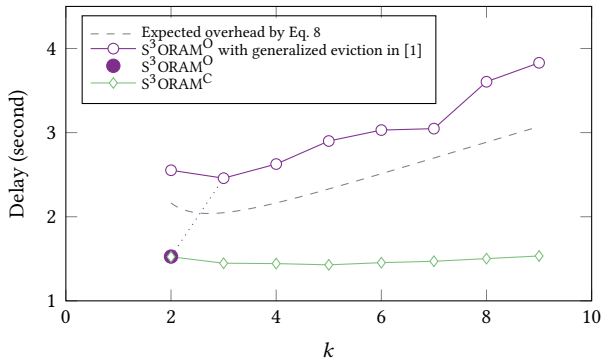


Fig. 20. The impact of tree degree.

not take into account the bucket load characteristic after each eviction for optimization. Meanwhile, this characteristic was fully exploited in the Triplet Eviction strategy, which allowed reducing the end-to-end delay by half (*the solid purple point with the dashed purple line*). In summary, considering a little gain that $k = 3$ can offer and the optimization that can be done with $k = 2$, we can see that S^{3}ORAM$^{\text{O}}$ scheme achieved the best performance at the default setting (i.e., binary tree layout with the Triplet Eviction). In S^{3}ORAM$^{\text{C}}$ scheme, the actual performance followed closely to the analytical result, which achieved the best performance at around $k \in \{3, 4, 5\}$. However, the gain was not so considerable compared with $k = 2$ (i.e., $< 6.5\%$). At $k > 5$, the delay of S^{3}ORAM$^{\text{C}}$ scheme started to increase and became worst than $k = 2$.

7.2.5 S^{3}ORAM storage overhead. At the client-side, S^{3}ORAM$^{\text{O}}$ does not require the stash component similar to Onion-ORAM. On the other hand, S^{3}ORAM$^{\text{C}}$ requires the stash of size $O(\lambda \cdot |b|)$ similar to Path-ORAM and Ring-ORAM. Therefore, given a database containing 512 KB blocks, S^{3}ORAM$^{\text{C}}$ scheme needs around 32-33 MB of the client storage for the stash, while S^{3}ORAM$^{\text{O}}$ requires nothing. The storage cost for the position map component in both (non-recursive) S^{3}ORAM schemes is slightly higher than their non-recursive counterparts. For instance, with a 16 TB database of 512-KB blocks ($N = 33,554,432$), S^{3}ORAM schemes cost 119 MB while the others (e.g., Onion-ORAM, Ring-ORAM, Path-ORAM) cost 100 MB. This is because we store not only the path information but also the specific location of each block in its assigned path.

At the server side, *each* server storage overhead in S^{3}ORAM$^{\text{O}}$ scheme increases by a factor of eight (i.e., $(8N - A) \cdot |b|$ bits) by Lemma 3. The server storage overhead for S^{3}ORAM$^{\text{C}}$ scheme increases by a factor of two (i.e., $2N \cdot |b|$), which is equal to Circuit-ORAM. Recall that all S^{3}ORAM schemes need at least three servers. The server storage for Path-ORAM and Ring-ORAM is $4N \cdot |b|$ bits and $6N \cdot |b|$ bits, respectively. The server storage for Onion-ORAM is similar to S^{3}ORAM$^{\text{O}}$ for one server but will increase after a sequence of access operation due to the ciphertext expansion of Additively HE.

Analytical comparison with other distributed ORAM schemes. We analytically compare S^{3}ORAM schemes with state-of-the-art multi-server ORAM schemes for data outsourcing. The most notable ORAM relevant to our framework is Multi-Cloud Oblivious Storage (MCOS) [38] as it also features $O(1)$ client communication overhead at the cost of $O(\log N)$ server-server bandwidth overhead like S^{3}ORAM. MCOS is better than S^{3}ORAM in the several aspects as follows. First, it does not require a minimal block size to achieve the constant client communication overhead, while S^{3}ORAM requires $\Omega(\log^2 N)$ - $\Omega(\log^3 N)$ block size. Second, it needs two servers to operate while S^{3}ORAM requires at least three servers. The main downside of MCOS over S^{3}ORAM is that it requires the client to store $O(\sqrt{N})$ data blocks compared with $O(1) - O(\log N)$. For instance, with 256 TB database with 2^{32} blocks, the client storage is 15 GB (*vs.* $\{0, 8\}$ MB in S^{3}ORAM). Another distributed ORAM relevant to S^{3}ORAM is the two-server ORAM scheme by Lu and Ostrovsky *et al.* [25]. Due to the hierarchical ORAM paradigm [20], the main advantage of this scheme is that the client does not need to maintain the position map and the stash components as in partition-based and tree-based ORAM schemes including S^{3}ORAM and MCOS. However, it incurs $O(\log N)$ client communication overhead as opposed to S^{3}ORAM and MCOS. As a result, it can operate on any block size and all the servers do not need to communicate with each other.

8 RELATED WORK

Single-server passive ORAM (without computation). The first ORAM proposed by Goldreich *et al.* [19] was in the context of software protection and followed by refinements (e.g., [20]). The recent ORAM schemes mainly have been considered in the client-server model to hide the data access pattern over a remote server (e.g., [33]). Preliminary ORAMs were costly in terms

of both communication and storage overhead, but recent ORAMs (e.g., [36, 40, 41, 43]) showed significant improvements. Path-ORAM [41], which follows the tree structure of [36], 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., [45]), secure computation (e.g., ([43], [44]), Parallel ORAM [8]) and secure processor [26]. However, Path-ORAM based schemes inherit its logarithmic communication blowup [6, 31].

Single-server active ORAM (with server computation). Ring-ORAM [34] reduced the communication cost of Path-ORAM by 2.5x given that the server performs XOR computations. Some other alternatives (e.g., [3, 12, 14, 27, 30]) 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 [27] used PIR scheme in [42] with Additively HE (AHE) (i.e., [32]) on top of tree ORAM structure [36]. 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 [42] on top of ObliviStore [39], which is based on Partition-ORAM in [40]. The TWORAM scheme in [15] constructed a garbled circuit [46] over the tree ORAM structure, which allows the client and server to perform the secure computation to access the block.

Multi-server active ORAM. Multi-server ORAM schemes were proposed to eliminate highly costly fully/partial HE operations. CHf-ORAM [28] attempted to use four non-colluding servers to achieve $O(1)$ bandwidth blowup under $O(1)$ blocks of client storage. However, CHf-ORAM [28] (as well as its predecessor [30]) 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 XOR computations for block retrieval over a k-ary ORAM tree structure. Stefanov *et al.* were among the first to propose a multi-server ORAM scheme [38] that leverages two non-colluding computational-capable servers to reduce the client-server bandwidth of Partition ORAM [40]. Very recently, Chan *et al.* proposed a perfectly secure 3-server ORAM scheme [7] based on the Hierarchical ORAM paradigm in [20]. Gordon *et al.* in [21] proposed a simple and efficient 2-server tree-based ORAM, which achieves $O(\log N)$ bandwidth overhead with $O(1)$ communication round. In this scheme, the position map is static meaning that the path assigned for each data block is deterministic and unchanged, which can be computed by a pseudo-random function. In a different line of research, distributed ORAM schemes were proposed for secure multi-party computation (e.g., [13, 25]). 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 [46]) and, therefore, their cost is higher than classical client-server ORAM model.

9 CONCLUSION

We developed a new distributed ORAM framework called S^{3}ORAM that is comprised of two multi-server ORAM schemes. Our schemes achieve $O(1)$ client bandwidth blowup with low client storage and low end-to-end delay while avoiding costly HE operations. The main idea is to exploit the homomorphic properties of Shamir secret sharing and a secure multi-party multiplication protocol to efficiently realize retrieval and eviction in tree-based ORAMs. We assessed the efficiency of S^{3}ORAM schemes by measuring their actual delay when deployed on a commodity cloud system (i.e., Amazon EC2) with various network settings and database sizes. Our experiments confirmed the effectiveness of S^{3}ORAM schemes compared with state-of-the-art ORAM schemes in many practical settings.

ACKNOWLEDGMENT

This work was 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.

[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] T-H Hubert Chan, Jonathan Katz, Kartik Nayak, Antigoni Polychroniadou, and Elaine Shi. 2018. More is less: Perfectly secure oblivious algorithms in the multi-server setting. In *International Conference on the Theory and Application of Cryptology and Information Security*. Springer, 158–188.

[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, 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.

[17] 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.

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

[19] 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.

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

[21] S Dov Gordon, Jonathan Katz, and Xiao Wang. 2018. Simple and efficient two-server ORAM. In *International Conference on the Theory and Application of Cryptology and Information Security*. Springer, 141–157.

[22] Thang Hoang, Ceyhun D. Ozkaptan, Attila A. Yavuz, Jorge Guajardo, and Tam Nguyen. 2017. S3ORAM: A Computation-Efficient and Constant Client Bandwidth Blowup ORAM with Shamir Secret Sharing. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS '17)*. ACM, New York, NY, USA, 491–505. https://doi.org/10.1145/3133956.3134090

[23] 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.

[24] Mohammad Saiful Islam, Mehmet Kuzu, and Murat Kantarcioglu. 2012. Access Pattern disclosure on Searchable Encryption: Ramification, Attack and Mitigation.. In *NDSS*.

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

[26] 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.

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

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

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

[30] 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.

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

[32] 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.

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

[34] 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.

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

[36] 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.

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

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

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

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

[41] 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.

[42] 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.

[43] 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.

[44] 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.

[45] 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.

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

APPENDIX

Figure 21 presents the subroutines of Circuit-ORAM [43] that are called in the eviction of S^{3}ORAM^C scheme presented in Figure 13.

PrepareTarget(x):

dest $\leftarrow \perp$; src $\leftarrow \perp$, target $\leftarrow (\perp, \ldots, \perp)$
for $i = H, \ldots, 0$ **do**
  if i = src **then**
    target[i] $\leftarrow$ dest; dest $\leftarrow \perp$; src $\leftarrow \perp$
  if ((dest = $\perp$ and $\mathcal{P}(x, i)$ has empty slot) or (target[i] $\neq \perp$)) and (deepest[i] $\neq \perp$) **then**
    src $\leftarrow$ deepest[i]
    dest $\leftarrow i$
return target

PrepareDeepest(x):

deepest $\leftarrow (\perp, \ldots, \perp)$; deepestIdx $\leftarrow (\perp, \ldots, \perp)$; src $\leftarrow \perp$; goal $\leftarrow -1$
if stash S is not empty **then**
  src $\leftarrow 0$
  goal $\leftarrow$ Deepest level that a block in the stash S can legally reside on path $\mathcal{P}(x)$
  deepestIdx[0] $\leftarrow j$, where j is the index of the selected deepest block in S
for $i = 1, \ldots, H$ **do**
  if goal $\geq i$ **then**
    deepest[i] $\leftarrow$ src
  $\ell \leftarrow$ Deepest level that a block in $\mathcal{P}(x, i)$ can legally reside on path $\mathcal{P}(x)$
  deepestIdx[i] $\leftarrow j$, where j is the index of the selected deepest block in $\mathcal{P}(x, i)$
  if $\ell >$ goal **then**
    goal $\leftarrow \ell$
    src $\leftarrow i$
return (deepest, deepestIdx)

Fig. 21. Circuit-ORAM eviction subroutines.