Write-Only Oblivious RAM based Privacy-Preserved Access of Outsourced Data

Lichun Li, Anwitaman Datta
School of Computer Engineering, Nanyang Technological University
Singapore
Email: {lcli, anwitaman}@ntu.edu.sg

Abstract

Oblivious RAM (ORAM) has recently attracted a lot of interest since it can be used to protect the privacy of data user's data access pattern from (honest but curious) outsourced storage. This is achieved by simulating each original data read or write operation with some read and write operations on some real and dummy data items. This paper proposes two single-server write-only ORAM schemes and one multi-server write-only ORAM scheme, which simulate only the write operations and protect only the write pattern. The reduction of functions however allows to build much simpler and efficient (in terms of communication cost and storage usage) write-only ORAMs. Write-only ORAM can be used in conjunction with Private Information Retrieval (PIR), which is a technique to protect data user's read patterns, in order to protect both write and read patterns. Write-only ORAM may be used alone too, when only write patterns need protection. We study two usage scenarios: (i) data publishing/sharing: where a data owner shares the data with others, who only consume the published information. Data consumers should not have write access to the outsourced data, and thus cannot use ORAM to protect their read patterns in this scenario. To hide access patterns from the outsourced storage, the data owner can use ORAM to write data, and data consumers use PIR to read data. Alternatively, for some applications, a data consumer can trivially download all data once or regularly, and neither the data owner nor data consumers mind that the outsourced storage learns such read pattern. Compared with using traditional ORAM, using the simpler write-only ORAM here produces much less communication cost and/or client-side storage usage. Our single-server write-only ORAM scheme produces *lower (typically one order lower) communication cost with the same client-side storage usage, or requires much less (typically at least one order less) client-side storage to achieve the same level of communication cost* than the best known single-server full functional ORAM schemes do. Compared with the best known multi-server ORAM scheme, our write-only ORAM schemes have *lower (typically one order lower) communication cost, or achieve the same communication cost with the same client-side storage usage in single-server setting.* (ii) the data owner's personal use: Our write-only ORAM schemes combined with PIR can be used as building blocks for some existing full functional ORAM schemes. This leads to *the reduction of the communication costs for two full-functional ORAM schemes by the factors of* $O(\log N)$ *and* $O(\sqrt{\log N} \times \log\log N)$, where N is the maximum data item count. One of these resulting schemes has a communication cost of $O(l)$, where l is data item length. This is *typically one order lower than the previous best known ORAM scheme's cost,* which is $O(\log N \times l)$. The other resulting scheme also achieves $O(\log N \times l)$ communication cost, but its *client-side storage usage is several orders lower* than the best known single-server ORAM's.

I. Introduction

Oblivious RAM (Random Access Machine: an abstract computer model) was proposed in [1] to hide a software's data access pattern, for the purpose of protecting software from reverse engineering in 1980s. A software can be re-complied to use ORAM (Oblivious RAM) technique, and simulate the original software's data operations on the memory using ORAM's methods like

accessing dummy data items (dummy items for short) or unnecessary real data items, re-encrypting a data/dummy item into a different ciphertext after accessing it, and obliviously shuffling real and dummy data items. From the view of the adversary observing the software's memory access, whether a simulated operation is read or write is indistinguishable. ORAM also hides individual data item's access frequency and the linkage between the simulated operations operating on the same data item. Here, the CPU executing the software's operations and the CPU's internal storage, i.e. registers, are considered secure. The adversary can't see the data in CPU.

In recent years, ORAM has attracted a lot of research interest, e.g. [2–5], since it can protect the privacy of data user's data access pattern from the honest but curious outsourced storage. The confidentiality of the data user's data in outsourced storage can be protected by encryption. But sometimes encryption is not enough to protect privacy as data access pattern could leak sensitive information [2]. Suppose that operations on certain data items is always followed by a specific stock action of the user. Then a curious server can predict the user's stock action by monitoring its data access pattern. In [6–8] it is further elaborated how access pattern may leak sensitive information.

This paper is focused on ORAM's outsourced storage usage instead of software protection usage. In ORAM's outsourced storage usage, the data user's node plays the roles of "software" and "CPU", while the outsourced storage plays the role of "memory". In the literature and this paper, such a user node is called ORAM client (client for short), and such an outsourced storage is called ORAM server (server for short).

PIR (Private Information Retrieval), first proposed in [9], is another technique used to protect data user's privacy of read patterns from honest but curious outsourced storages. PIR allows users to retrieve a data item from a database without letting the database server know which item is being retrieved. Such a privacy primitive is used in diverse settings including patent databases [10], pharmaceutical databases [10], email systems [11], e-commerce [12], P2P file sharing systems [13], etc. to hide an user's interest/profile.

Motivation. This paper proposes write-only ORAM, and discusses how to use it with/without PIR to protect data access pattern. Existing ORAM schemes are all full functional, which can simulate both write and read operations of standard non-oblivious RAM. In another word, existing ORAM schemes can hide the access (write/read) pattern of simulated operations and the type (write or read) of any simulated operation. This paper's write-only ORAM simulates only write operations, and protects only write patterns. As far as we know, this is the first work on write-only ORAM. The reduction of functions allows us to build simpler ORAM with better performance. Compared with the best known full functional ORAM schemes, this paper's write-only ORAM schemes have lower (typically one order lower) communication cost, or require much less (typically at least one order less) client-side storage to achieve the same level of communication cost. Write-only ORAM can be used in three situations: 1) If only write pattern needs protection, write-only ORAM can be used alone; 2) If read and write patterns need protection but operation type doesn't need protection, write-only ORAM and PIR can be used; 3) If access pattern and operation type need protection, full functional ORAM can be built based on write-only ORAM and PIR. Write-only ORAM is useful in the below two scenarios, which contain these situations.

- The first scenario is data sharing via outsourced storage. The data owner shares its data with consumers using an outsourced storage. The data owner's write pattern and data consumers' read pattern can be hidden from the outsourced storage using full functional ORAM. However, in that case the data consumers would need to have write access at the outsourced storage. It's usually unacceptable because a malicious data consumer can use this right to tamper data items. To allow data consumers reading data obliviously but without write access, PIR can be used instead. Then ORAM and PIR are used to protect data owner's write pattern

and data consumer's read pattern respectively. Alternatively, for some applications, a data consumer that needs all data can trivially download all the data from the outsourced storage once or regularly. Therefore, a possible data sharing solution is: the data owner uses ORAM to write (add or update) data at the outsourced storage; data consumers use PIR to retrieve data from the outsourced storage, or trivially download all data from the outsourced storage. Compared with using full functional ORAM, using the simpler write-only ORAM with lower cost here is better. For some applications, the data owner may need to read data items from outsourced storage too. If the data owner doesn't mind to expose its operation type (read or write), it can use PIR to retrieve data items or download all data items trivially like a data consumer. Otherwise, one should use full functional ORAM which can be achieved by composing write-only ORAM and PIR differently, as discussed next.

- The second scenario is outsourced storage for the data owner's own use, or when multiple parties own and collaborate over the data, and even the type of operation (read or write) is confidential, i.e., a fully functional ORAM is needed. Both read and write operations are carried out by data owner(s), and full functional ORAM is used to protect access pattern. Based on write-only ORAM, two kinds of full functional ORAM can be built: (a) full functional ORAM built by directly combining write-only ORAM with PIR; (b) full functional ORAM built by combining write-only ORAM with PIR and using it as a building block in [4, 5]'s ORAM frameworks. Compared with traditional full functional ORAM, these two new kinds of full functional ORAMs make a tradeoff of computational cost for communication cost and client-side storage usage. We focus on full functional ORAM (b) in this paper, because ORAM (b)'s computational cost is lower than ORAM (a)'s. We use write-only ORAM and PIR as a building block to improve two full functional ORAM schemes [4, 5]. By using write-only ORAM and PIR in [4] and [5]'s ORAM frameworks, the communication cost can be reduced by a factor of $O(\sqrt{\log N} \times \log\log N)$ and $O(\log N)$ respectively where N denotes the maximum number of data items the ORAM system can store.

Contributions. The main contributions of this paper are as follows:
(i) Two novel single-server write-only ORAM schemes (basic write-only and advanced write-only ORAM) and a multi-server write-only ORAM scheme are proposed. The advanced single-server as well as the multi-server schemes leverage on the basic scheme. The advantages of advanced write-only ORAM depend on proper optimization of parameters, and is explored in the paper. Though write-only ORAM is simpler than traditional full functional ORAM, this paper's contribution (i) is not trivial. New algorithms and new data organizations are used to realize efficient designs. Moreover, our design supports multiple ORAM clients and concurrent access, which is atypical in most existing full functional ORAMs.
(ii) This paper studies and improves the method of supporting PIR clients in ORAM. This method is not only useful for write-only ORAM in above two scenarios, but also needed if using full functional ORAM and PIR together in the data sharing via outsourced storage scenario.
(iii) We demonstrate how a full functional ORAM can be directly build based on write-only ORAM and using write-only ORAM as a building block.

Organization. Next, in Section II we discuss the necessary background and related works. We give an overview of our write-only schemes in Section III. In Sections IV and V we present our basic and advanced write-only ORAM schemes respectively. In Section VI, we show how to extend the basic write-only ORAM scheme to a multi-server write-only ORAM scheme. We discuss how to use write-only ORAM as a building block to improve two full functional ORAM schemes [4, 5] in Section VII. Finally, we conclude along with a discussion of future works in Section VIII.

II. Background and Related Work

A. Oblivious RAM

Most ORAM schemes are single-server schemes. Very few schemes, e.g. [20], are multi-server schemes, that assume k independent and non-colluding outsourced servers.

In Table I we list the best traditional ORAM schemes that we have identified. In this table and the rest of this paper, N denotes the maximum number of data items the system is designed to store at any time, and l is the data item length in bits. The last ORAM scheme in table [20] is the best multi-server scheme, while the other schemes are single-server ones. Among traditional fully functional ORAM schemes, they have the lowest communication costs under different levels of storage usages.

Table I
Traditional ORAM

scheme	client storage	server storage	amortized communication/computational cost
[3, 16, 17]*	$O(N^{1/r} \times l)$	$O(N \times l)$	$O(\log N \times l)$
[4]**	$O(\log N \times N + l)$	$O(\log N \times N \times l)$	$O(\log N \times \sqrt{\log N} \times \log\log N \times l)$
[18]	$O((\log N)^2/\log l \times l)$ $O(\log N \times l)$	$O(N \times l)$	$O((\log N)^2/\log l \times l + \log N \times l)$
[19]	$O(l)$	$O(N \times l)$	$O((\log N)^2/\log\log N \times l)$
[20]***	$O(l)$	$O(N \times l)$	$O(\log N \times l)$

* r is a small constant.
** [4] has two ORAM constructions. This result is achieved by its basic ORAM construction.
*** This is a two-server (and also the best multi-server) ORAM scheme.

B. Private Information Retrieval (PIR)

Broadly, there are two kinds of PIR techniques — itPIR (information-theoretic PIR) and cPIR (computational PIR), providing unconditional and computational hardness based privacy respectively. The privacy in cPIR is guaranteed subject to computational bounds on the server, while all communication efficient itPIR schemes are multi-server based, and assume that not all the servers collude together. Neither cPIR nor itPIR require that the data items in the database are encrypted. In cPIR, the user's PIR client encrypts the wanted data item's position in the database, and sends the encrypted position as a query to the database server; the database server computes an answer using the encrypted position and all the data items in the database; the client decrypts the answer and obtains the desired data item. There is no single-server itPIR scheme apart the trivial one, i.e., downloading the whole database. Efficient multi-server itPIR schemes can be built under the conditions that each server holds a replication of the database and not all servers collude. A multi-server itPIR scheme is called a t-private k-server itPIR scheme if it requires k ($k > 1$) database servers and resists up to t ($k > t \geq 1$) colluding servers. Suppose a database has N data items, a t-private k-server itPIR works in the following way: the user's PIR client uses the position of the wanted data item to generate k queries, and sends each server a query; each server computes an answer using its received query and $O(N)$ data items in the database, and sends the answers to the client; the client recovers the wanted data item from received answers. Any t or less servers together can't learn any non-trivial information of the wanted data item's position from their received queries. In most itPIR schemes, a secret sharing scheme is used to generate shares from the secret position, and each share is a query [21, 25].

The computational costs of cPIR and itPIR are both $O(N \times l)$. However, the constant in cPIR's $O(N \times l)$ cost is much higher, and itPIR's performance is better than cPIR's [36]. Originally, cPIR was considered impractical for normal database sizes [37]. Subsequently, efficient cPIR schemes were invented and considered computationally practical for restricted database sizes [24, 36, 38]. Recently, cPIR and itPIR schemes exploiting parallelization of cloud/cluster computing [40, 41] or

GPU [39] were proposed to improve PIR's performance and make it practical for bigger databases. The parallelization of cloud computing can be exploited in this paper's outsourced storage scenarios, and some cloud providers like Amazon also provide virtual machines with GPU.

Compared with traditional ORAM, PIR has lower communication cost in most cases for the outsourced data sharing and outsourced storage for data owner scenarios (where database size is at most a few PB, and individual data item size is at least a KB) considered in this paper. The communication costs of most PIR schemes are between $O(\log N + l)$ and $O(N + l)$. [23]'s cPIR scheme can achieve $O(\log N + l)$ communication cost. Normally, in the scenarios considered in this paper, $O(\log N + l) = O(l)$. Quite a few PIR schemes like [21, 22] can achieve a communication cost of $O(N^{1/d} + l)$, where $d \geq 1$. For many typical setting of N and l, most of these PIR schemes can make $O(N^{1/d}) \leq O(l)$, and achieve a communication cost of $O(l)$. For example, suppose $N = 2^{42}$ and $l = 2^{14}$ (2KB). The database size is 8PB. Then $N^{1/d} \leq l$ if $d \geq 3$, which can be easily achieved by some schemes. For example, d depends on k and t, e.g. $d \times t + 1 = k$ in two of [21]'s itPIR schemes (main PIR protocol and binary PIR protocol). For another example, d depends on N, e.g. $d = O(\log \log N)$ in [22]. So the best communication cost of PIR is $O(l)$ in many cases. In this paper, we care more about the comparison between the communication costs of traditional full functional ORAM and PIR when PIR is used with write-only ORAM to improve [4, 5]'s ORAM schemes. As introduced later in Section III and elaborated in Section VII, in this usage, PIR is used to retrieve data items from a partition of the database containing $O(\sqrt{N})$ or $O(\log N)$ data items instead of from the whole database. For a 1PB database with 1KB data item size, the partition size is at most several GB or hundreds of KB respectively, and the database partition can be viewed as a much smaller database. Then it's much easier to achieve $O(l)$ communication cost for many PIR schemes. Therefore, in this paper, we use $O(l)$ as PIR's communication cost when evaluating the overall communication cost of our designs.

In this paper, write-only ORAM may be used with single-server or multi-server PIR to protect access pattern. Please note that a single-server ORAM scheme can be used with a multi-server PIR scheme though they require different number of servers. A multi-server PIR scheme requires that each server has a replication of the database. The ORAM client can run a single-server ORAM scheme in one server, and synchronize the changes of the database to every other server. Similarly, a multi-server ORAM scheme may be used together with a single-server PIR scheme. However, multi-server PIR scheme is computationally more efficient than a single-server one. If using multi-server ORAM scheme, multi-server PIR scheme should be chosen over single-server PIR scheme. The details of using write-only ORAM and PIR together will be given later.

C. Combining ORAM with PIR

Most trusted-hardware-assisted PIR schemes [26–29] are based on ORAM. These schemes share the same basic principle. A trusted coprocessor at the database server works as a representative of the PIR client. The trusted coprocessor uses ORAM to read the data item wanted by the PIR client from the server's storage obliviously, and returns the data item to the PIR client. Compared with PIR schemes without trusted hardware, these schemes have lower computational and communication costs. In this paper, we use normal PIR with write-only ORAM, and don't require trusted hardware at the server side. Also, the PIR discussed in this paper except this paragraph is normal PIR without trusted hardware.

In [15], a privacy-preserving data sharing over outsourced storage is proposed using full functional ORAM to protect the data owner's data update pattern and PIR to protect data consumers' read patterns. The design also considers the needs of pricing and access control, which allows the data owner to control which data items are accessible to a specific data consumer. In this paper, we borrow and improve [15]'s method of supporting PIR clients in ORAM when combining write-only ORAM with PIR, to support the above scenario more efficiently.

The ORAM scheme in [4]'s is improved by [30] using PIR and PIR-writing [31] as building blocks in [4]'s ORAM framework. PIR-writing can update a data item without letting the database server knowing which data item is updated. PIR-writing utilizes homomorphic encryption to modify every data item's ciphertext, and only one data item's plaintext is updated. [30] improves the communication cost to $O((\log N)^3 + (\log N)^2 \times l)$ with $O(l)$ client-side storage. Inspired by [30], we further improve upon [4] by using PIR and write-only ORAM as building blocks in [4]'s ORAM framework, and reduce communication cost to $O(\log N \times l)$ with $O(l + N^{1/r} \times \log N)$ client-side storage. r is a small constant here. Compared with using PIR-writing, using write-only ORAM produces less computational cost.

III. Overview of Write-only ORAM Schemes

A. System Setting

The outsourced storage, i.e. the ORAM server, is considered honest but curious. The input of an ORAM client or a group of ORAM clients is a sequence of data operations (simulated data operations), and the i-th operation is denoted as (op_i, q_i, x_i). op_i is the type of data operation, which could be read or write. A read operation retrieves the value of the data item indexed q_i, while a write operation sets the value of the data item indexed q_i to x_i. A full functional ORAM system is considered secure if, for any two equal-length sequences of data operations, which sequence is chosen by the ORAM client(s) as input is computationally indistinguishable for the ORAM server(s). In contrast, like the name suggests, in write-only ORAM the data operation type is always write. Thus to say, both full functional and write-only ORAM hide individual data item's access frequency and the linkage between the simulated operations operating on the same data item. Full functional ORAM also hides the type of any data operation, which our write-only ORAM operations do not. The ORAM server however knows how many operations are executed during any time period.

In contrast to most existing works in the literature which consider single ORAM client and single ORAM server, our approach supports the possibility of data consumers' non-ORAM clients, multiple ORAM clients and multiple ORAM servers. In such a setting, the (write-only) ORAM mechanisms do not prevent the data consumers' clients from learning about the data owner's write pattern since they have access to read the data, and thus they can repeatedly read the whole data to identify the differences. Likewise, when multiple ORAM clients collaboratively mutate a collection of data, they can (as well as need to) also see what changes are being made by others. (For example, the data owner is a company and the ORAM clients of multiple employees access the data stored in the ORAM server.) However, a data consumer can hide its read pattern from everyone by using PIR.

Finally, for the case of multiple ORAM servers, our scheme requires that not all servers collude. More specifically, the multi-server scheme can be characterized as a t-private k-server scheme ($t < k$), which is secure only if at most t of all the k servers collude.

B. Design Overview

Our *basic write-only ORAM* has a very simple structure containing two storage areas. In contrast, the most efficient traditional ORAMs have a $O(\log N)$-tier pyramid structure or a $O(\log N)$-level tree structure. To support PIR in ORAM, an additional table named slot mapping table containing the mappings from item indices to locations is maintained additionally. The slot mapping table design is based on [15], to which we make two major modifications to reduce the client-side storage usage for maintaining this table.

The *advanced write-only ORAM* uses the basic write-only ORAM as a building block. Advanced write-only ORAM's construction contains multiple buckets, and each bucket is implemented as a

basic write-only ORAM. We optimize the bucket count to reduce client-side storage usage in different configurations of maximal data item count N and item length l. The advanced write-only ORAM also needs to maintain a table named bucket mapping table containing the mappings from item indices to bucket numbers. The bucket mapping table is maintained at the client-side as if there is only a single ORAM client. If there are multiple ORAM clients, the table is maintained at the server-side as an additional basic write-only ORAM.

The *multi-server write-only ORAM* also builds on the basic write-only ORAM, but uses a different *oblivious merge* mechanism to update values of data items. Basic write-only ORAM uses an oblivious merge algorithm based on a recently proposed oblivious sort algorithm [44]. Giving the assumption that not all servers collude, the multi-server write-only ORAM adopts a low-cost multi-server oblivious merge based on oblivious remove and oblivious scramble.

Compared with traditional ORAMs, the amortized communication costs in write-only ORAMs are lower. In traditional ORAMs, oblivious shuffle is the most costly operation performed after every oblivious read/write operation to hide the positions of the data items being accessed. In contrast, in write-only ORAMs, oblivious merge is the most costly (but less costly than oblivious shuffle) operation, and it is used to update data items obliviously with new written values. The simpler structure and less costly oblivious merge makes write-only ORAM's amortized communication cost of oblivious write lower than traditional ORAM's amortized cost of oblivious access, assuming a given size of storage space.

Based on write-only ORAM, two kinds of *full functional ORAMs* can be built: (a) full functional ORAM built by directly combining write-only ORAM with PIR; (b) full functional ORAM built by combining write-only ORAM with PIR and using it as a building block in [4, 5]'s ORAM frameworks. In full functional ORAM (a), a read operation of a data item can be simulated by using PIR to retrieve the data item and a following oblivious write of a dummy data item. Similarly, a write operation of a given data item can be simulated by using PIR to retrieve a random item and a following oblivious write of the target data item.

One can also use write-only ORAM and PIR as building blocks to improve two full functional ORAM frameworks [4, 5]. Both frameworks proposed ORAMs containing multiple buckets, but [4]'s bucket is a full functional ORAM storing at most $O(\log N)$ data items, while [5]'s bucket is a full functional ORAM storing at most $O(\sqrt{N})$ data items. By reusing [4] and [5]'s ORAM frameworks and implementing their buckets based on write-only ORAM and PIR, the communication cost is reduced by a factor of $O(\sqrt{\log N} \times \log\log N)$ and $O(\log N)$ respectively. To combine their ORAM frameworks and our write-only ORAM, we make slight modifications to write-only ORAM and [4]'s ORAM framework.

Compared with traditional full functional ORAM, these two new kinds of full functional ORAMs have higher computational cost but lower communication cost or client-side storage usage. Lower communication cost and client-side storage usage are important to clients with limited bandwidth and storage space. Higher computational cost may decrease data operation throughput and increase data access latency. The higher computational cost is due to the use of PIR, and the cost depends on the total size of data/dummy items hiding the retrieved item. So the computational cost depends on the whole database size in full functional ORAM (a), while the cost depends on the bucket size in full functional ORAM (b). Because ORAM (b)'s computational cost is lower than ORAM (a)'s, we focus on full functional ORAM (b), and defer the design of full functional ORAM (a) in Appendix A for the sake of completeness. For diverse applications, the corresponding typical N and l values are such that the full functional ORAM (b)'s computation/communication/client-side storage cost trade-offs are practical. To pinpoint quantitatively the tradeoffs under different conditions (different values of N and l; different PIR schemes; different hardware; different bandwidth), further implementation and benchmarking oriented work is still needed, particularly to

evaluate how data operation throughput and data access latency are affected. We provide a back of the envelope estimation below based on the performance data reported in the literature, but defer any further experimental work for the future.

Since the database size in most experiments reported by the ORAM and PIR literatures is not more than 1TB, let's assume the database size is 1TB in the performance estimation. To the best of our knowledge, ObliviStore [35] is the fastest oblivious RAM implementation for outsourced storage to date. Performance tests on a 1TB database with 4KB sized data items stored in a commodity machine (the ORAM server) yielded throughput and response latency of 364 KB/s and 196ms respectively. Recall that a bucket of [4] and [5]'s ORAMs store $O(\log N)$ and $O(\sqrt{N})$ items respectively. As shown in [5, 45], the constants in $O(\log N)$ and $O(\sqrt{N})$ can be lower than 2 and 3 in practice. We can reasonably assume [4] and [5]'s bucket sizes are $6 \times \log N \times l$ and $4 \times \sqrt{N} \times l$ respectively considering the storage expansion of using write-only ORAM. Thus, for 1TB database with 4KB sized data items, [4] and [5]'s bucket sizes are 672KB and 256MB. For the 1TB database with 1MB sized data items, [4] and [5]'s bucket sizes are 80MB and 4GB. To generate a response to a PIR request, a server of full functional ORAM (b) would need to process all data in a bucket. Performance study [36] showed that a commodity machine's database processing speed in the fastest itPIR scheme [9] is 1GB/s or more depending on the database size and memory size. Likewise, [39] tested several cPIR schemes in three kinds of commodity machines with GPU. Among these cPIR schemes, Gentry and Ramzan's scheme [23] is the most desirable one with a data processing speed of 187.5KB/s. Though the speed of [38]'s scheme is faster, [38]'s scheme is less communication efficient than the best traditional ORAM. ([38] is still a good option for outsourced data sharing scenario.) The outsourced storage can use multiple commodity machines to parallelize and speed up the database processing. Using these figures, we extrapolate that the outsourced storage can use an acceptable number of machines to achieve acceptable throughput and response latency. We may estimate PIR's performance in our full functional ORAM from [40] and [41] too. [40]'s showed that the average query time of [42]'s itPIR scheme is lower than 1 second when a PIR server is a cloud with 320 machines and the database size is 4GB. [41] tested its cPIR scheme using 20 Amazon instances. It took a PIR client about 3 minutes to retrieve a 1MB sized data item from a 10GB database. If the bucket size is small or more machines are used, [41]'s cPIR scheme could have an acceptable performance compared with traditional ORAM's performance.

C. Our Results

Our write-only ORAM schemes' communication costs and storage usages are shown in Table II, III and IV. Compared with basic write-only ORAM, advanced write-only ORAM requires less client-side storage usage to achieve $O(l)$ communication cost when $O(N \times \log N) < O(N^{1/r} \times l)$. Compared with single-server write-only ORAMs, multi-server write-only ORAM can achieve $O(l)$ communication cost with less client-side storage usage when $O(l+N) < O(N^{1/r} \times l)$. We show the communication costs, computational costs and storage usages of two write-only ORAM based full functional ORAMs using [4] and [5]'s ORAM frameworks in Table V. We can see that our results are better than traditional full functional ORAM in terms of communication cost and client-storage usage.

For data sharing via outsourced storage, the write-only ORAM schemes with/without supporting PIR can be used. Let's consider ORAM without supporting PIR first. Please note that traditional ORAM's costs in Table I are the costs without supporting PIR. As shown in Table II, III and IV, the best communication cost of write-only ORAMs is $O(l)$. In contrast, $O(\log N \times l)$ is the best communication cost of single-server and multi-server traditional ORAMs, which requires $O(N^{1/r} \times l)$ and $O(l)$ client-side storage usage respectively to achieve this cost. Compared with the traditional multi-server ORAM, our single-server ORAM can achieve the same communication cost and client-side storage usage using only one server, which doesn't require multiple servers and

Table II
BASIC WRITE-ONLY ORAM

	client storage	server storage	amortized communication cost	
			oblivious write	PIR read
supporting trivial download only	$O(l)$	$O(N \times l)$	$O(\log N \times l)$	
	$O(N^{1/r} \times l)$	$O(N \times l)$	$O(l)$	
supporting PIR and trivial download	$O(l)$*	$O(N \times l)$	$O(\sqrt{N} \times (\log N)^2)$	$O(\sqrt{N} \times \log N)$
	$O(l + N^{1/r} \times \log N)$	$O(N \times l)$	$O(\log N \times (l + \sqrt{N}))$	$O(l + \sqrt{N} \times \log N)$
	$O(N^{1/r} \times l)$	$O(N \times l)$	$O(l + \sqrt{N} \times \log N)$	$O(l + \sqrt{N} \times \log N)$

$r \geq 2$ and r is a small constant.
* This row's costs are under the condition of $O(l) < O(N^{1/r} \times \log N)$. If $O(l) \geq O(N^{1/r} \times \log N)$, please check the next row.

Table III
ADVANCED WRITE-ONLY ORAM

PIR support	case	client storage	server storage	amortized communication cost	
				oblivious write	PIR read
no	1	$O(N \times \log N)$	$O(N \times l)$	$O(l)$	
	2	$O(N \times \log N + l)$	$O(N \times l)$	$O(l)$	
yes	1	$O(N \times \log N)$	$O(N \times l)$	$O(l + \sqrt{N} \times \log N)$	$O(l + \sqrt{N} \times \log N)$
	2	$O(N \times \log N + l)$	$O(N \times l)$	$O(l + \sqrt{N} \times \log N)$	$O(l + \sqrt{N} \times \log N)$

case 1: There are multiple ORAM clients and $N \geq l$.
case 2: There is only one ORAM client or $N < l$.

the assumption of not all servers colluding. Compared with the traditional single-server ORAMs achieving this cost, our single-server ORAM can achieve this cost with much less client-side storage usage ($O(l)$ only), or use at most the same client-side storage usage ($O(N^{1/r} \times l)$) to achieve lower communication cost ($O(l)$). Therefore, our single-server write-only ORAM reduces communication cost by a factor of $\log N$ or client-side storage usage by a factor of $N^{1/r}$. Typically, communication cost and client-side storage usage can be reduced by one order and several orders respectively. For example, if $N = 2^{30}$ and $r = 3$, $\log N = 30$ and $N^{1/r} = 1024$. Write-only ORAM and traditional ORAM can use the same way to support PIR in the data sharing scenario. If using $O(N^{1/r} \times \log N)$ (or $O(\log N)$) client-side storage to support PIR, the communication costs of both write-only ORAM and traditional ORAM will be increased by $O(\sqrt{N} \times \log N)$ (or $O(\sqrt{N} \times (\log N)^2)$).

For the outsourced storage used by the data owner, the full functional ORAM schemes using

Table IV
MULTI-SERVER WRITE-ONLY ORAM

PIR support	client storage	server storage	amortized communication cost	
			oblivious write	PIR read
no	$O(N + l)$	$O(N \times l)$	$O(l)$	
yes	$O(N + l)$	$O(N \times l)$	$O(l + \sqrt{N} \times \log N)$	$O(l + \sqrt{N} \times \log N)$

Table V
FULL FUNCTIONAL ORAM USING WRITE-ONLY ORAM AS A BUILDING BLOCK

	client storage	server storage	amortized comm. cost	amortized comp. cost
[5]'s ORAM framework	$O(\sqrt{N} \times l + N \times \log N)$	$O(N \times l)$	$O(l)$	$O(\sqrt{N} \times l)$
[4]'s ORAM framework	$O(l + N^{1/r} \times \log N)$	$O(N \times l)$	$O(\log N \times l)$	$O((\log N)^2 \times l)$

write-only ORAM as a building block can be used. As shown in Table V, the first scheme improves communication cost to $O(l)$, and the second scheme achieves $O(\log N \times l)$ communication cost, which is equal to the best communication cost of single-server and multi-server traditional ORAMs. As shown in Table I, traditional single-server ORAM schemes achieving this communication cost require at least $O(N^{1/r} \times l)$ client-side storage usage. In contrast, the second scheme uses $O(l + N^{1/r} \times \log N)$ client-side storage, which typically is several orders lower.

IV. Basic Single-server Write-only ORAM

A. Preliminaries

Similar to most full functional ORAM schemes, this paper's single-server and multi-server write-only ORAM schemes are based on the following assumptions. The maximum number of data items in the outsourced storage, i.e. server, is N. During the use of the outsourced storage, data items may be added or deleted. Thus, at any given time, the actual data count is equal to or less than N. Data items' indices, i.e. IDs, are in $\{1, 2, ...N\}$, and all data items have same length: l bits. The server won't tamper stored data, or tampering will be detected using techniques like MAC (Message Authentication Code) and signature.

Furthermore, as in full functional ORAM schemes, dummy items are used to hide access pattern, and data/dummy items uploaded to the server are encrypted with a semantically secure probabilistic encryption [43] scheme and therefore two encrypted copies of the same item look different. The server cannot identify whether these two copies correspond to the same item or not.

B. Construction

As shown in Figure 1, the server-side storage contains two areas: a main part and a write cache. Both areas have N slots, and each slot can store one encrypted data item or dummy item. All items in the main part and the write cache are encrypted. The main part is always filled with items so that the server cannot tell the actual data item count in the main part. There may be N or less data items in the main part, and the remaining items, if any, are dummy items. Dummy items are used to hide data item count and access pattern from the server. At the beginning, the ORAM client initializes the main part by uploading N encrypted items to it. These N items include all data items outsourced at the beginning and maybe some dummy items for hiding current data item count. Some meta data of an item, e.g. data item index, item type (real data item or dummy item) and freshness level, are encrypted and stored along with the item's content. Freshness level is optional. It will be introduced later. The ORAM client can determine whether an item in the main part is a data item or dummy item only after decrypting it, and accordingly get the item's index and freshness level if the item is a data item. The write cache stores recently written data items. The write cache is empty at the beginning. Every time the ORAM client does an oblivious write operation, it uploads an encrypted data item, and the item is put in the first empty slot in the write cache.

If the write cache is full after writing a data item, the ORAM client does an oblivious merge: it obliviously updates the main part with the recently written data items in the write cache, and empties the write cache. After the oblivious merge, newly added data items are put in the main part, and the outdated data items in the main part are replaced with the most updated data items. Please note that the server cannot detect whether an item's plaintext has changed or not after the oblivious merge because every item's ciphertext changes due to use of a semantically secure probabilistic encryption scheme.

C. Oblivious Merge Algorithms

There are two oblivious merge algorithms for basic write-only ORAM: oblivious merge based on oblivious sort and trivial oblivious merge. Oblivious merge based on oblivious sort is used

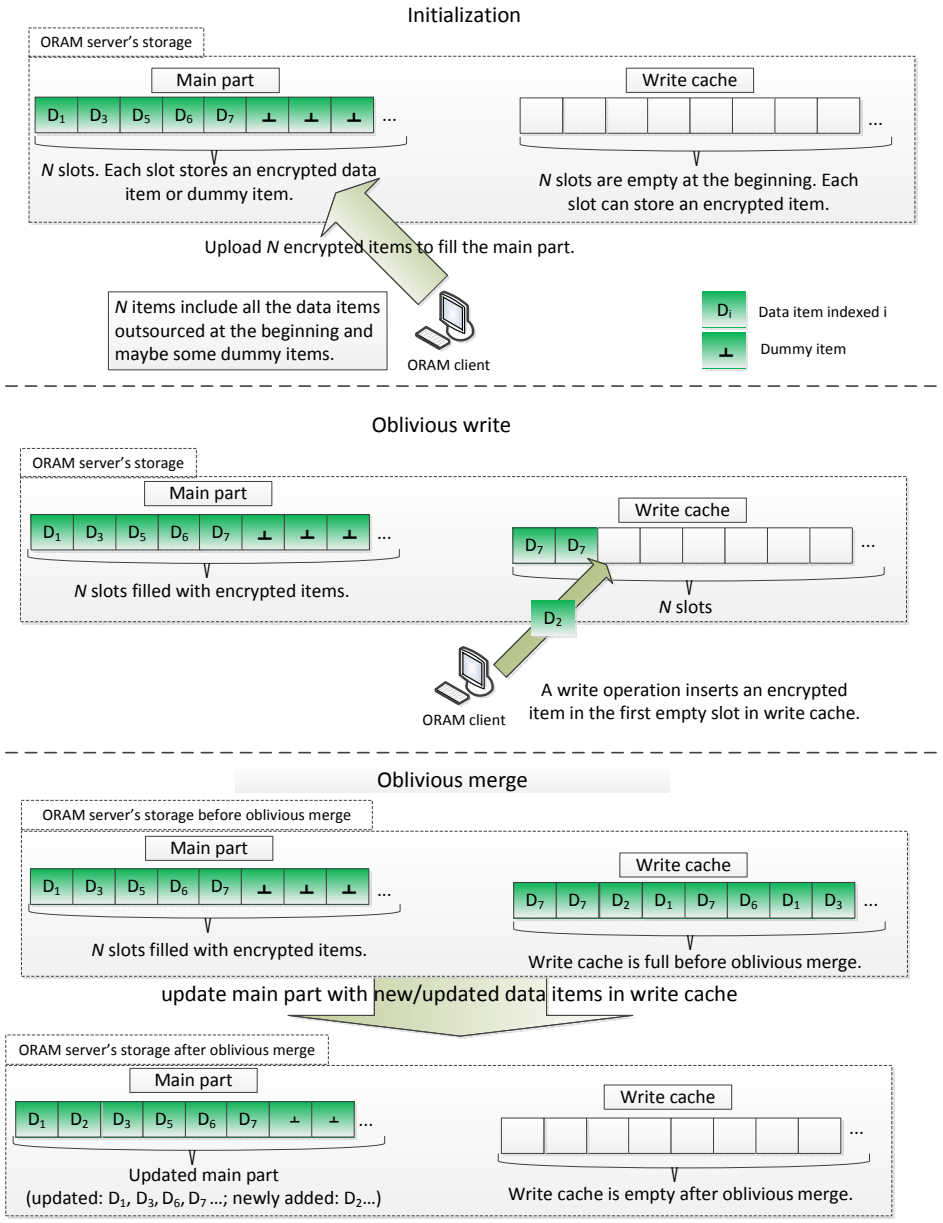

Figure 1. Basic write-only ORAM

in most cases. When the basic write-only ORAM is used as a building block in other ORAM schemes, the trivial oblivious merge may be used.

The ORAM client does the oblivious merge based on oblivious sort by following steps, and a toy example is shown in Figure 2.

- First, the client scans all items, and tags each data item with a freshness level. For a data item in the main part, its freshness level is 0. For a data item in the write cache, its freshness level is its position in the write cache (positions are 1-based here). The client downloads the items in the main part and write cache one by one. If a downloaded item is a data item, the client tags it. Then the client uploads the downloaded item (dummy item or data item) back to the server after re-encrypting it. A data item's tag is encrypted along with its content so that the

server can't learn any data item's freshness level. Because every scanned item is re-encrypted with a semantically secure probabilistic encryption scheme, the server can't learn any scanned item's type.

- Second, the client sorts the items in the server using a data-oblivious external-storage sort algorithm. In our case, the outsourced storage is the "external-storage". We use [44]'s oblivious sort algorithm. After sorting, items of the sorted list are in the order that: 1) data items before dummy items; 2) the data item with smaller index before the data item with bigger index; 3) for data items with the same index, items with higher freshness levels are before items with lower freshness levels.
- Third, the client scans the sorted list, removes tags, and replaces outdated data items with dummy items. The client detects outdate items by comparing adjacent items' indices and freshness levels. If two items have the same index, the item with lower freshness level is outdated. The client scans the sorted list from head to tail. The client downloads items in the list one by one to scan. If current scanned item is a data item, the client compares current scanned item's index and freshness level with last scanned item's index and freshness level. If current scanned item is an outdated data item, the client uploads a new dummy item to the server. Otherwise, the client re-encrypts the current item, and uploads it.
- Fourth, the client sorts all items again obliviously. [44]'s oblivious sort algorithm is used again here. After sorting, items are in the order that data items are before dummy items.
- Fifth, the client asks the server to put the first N items of the sorted item list in the main part.

It's easy to find out that each of above steps is oblivious and above oblivious merge algorithm is secure. The second and fourth steps are the most costly parts of the oblivious merge based on oblivious sort. Under the assumption of $l \geq (\log N)^{\varepsilon}$ ($\varepsilon > 0$ is a small constant), [44]'s oblivious sort algorithm produces following communication cost for sorting a list of N items: $O(N \times \log N \times l)$ if using $O(l)$ client-side storage ; $O(N \times r \times l) = O(N \times l)$ if using $O(N^{1/r} \times l)$ client-side storage. Here, $r \geq 1$ and r is a small constant can be omitted in $O(N \times r \times l)$. Normally, $l \geq (\log N)^{\varepsilon}$ stands. It's also acceptable to choose a value for r that makes $r \geq 2$. In this paper, we assume $r \geq 2$, which will simplify the analysis of costs. The most computationally costly operations in the oblivious merge are decrypting downloaded items and encrypting items to be uploaded. So, the communication and computational costs are the same in the second and fourth steps. Both costs of the other steps are $O(N \times l)$, which is less than the cost of the second and fourth steps. Therefore, the oblivious merge's communication and computational costs are both: $O(N \times \log N \times l)$ if using $O(l)$ client-side storage; $O(N \times r \times l) = O(N \times l)$ if using $O(N^{1/r} \times l)$ client-side storage. Here, $r \geq 2$ and r is a small constant, which can be omitted in $O(N \times r \times l)$.

If the ORAM client has $O(N \times l)$ storage space, it can do a trivial oblivious merge locally. First, the client downloads all items, and picks the most updated data items. Second, the client re-encrypts the picked data items, and uploads them to the server. Third, if the number of uploaded data items is less than N, the client creates one or some dummy items, and uploads the dummy item(s) to the server to make uploaded item count be N. The server will put all received items in the main part. A client having $O(N \times l)$ storage space is usually not practical. However, when the basic write-only ORAM is used as a building block in other ORAM schemes, this trivial oblivious merge may be useful.

Concurrent read/write during oblivious merge. An oblivious merge may take a long time when there are many items. To enable concurrent read/write operations during the oblivious merge, as shown in Figure 3, the server allocates space for a new main part and a new write cache. The oblivious merge outputs the merged items to new main part. A concurrent ORAM write can put the data item in the new write cache. As discussed in Section I, a data reader could be a node of

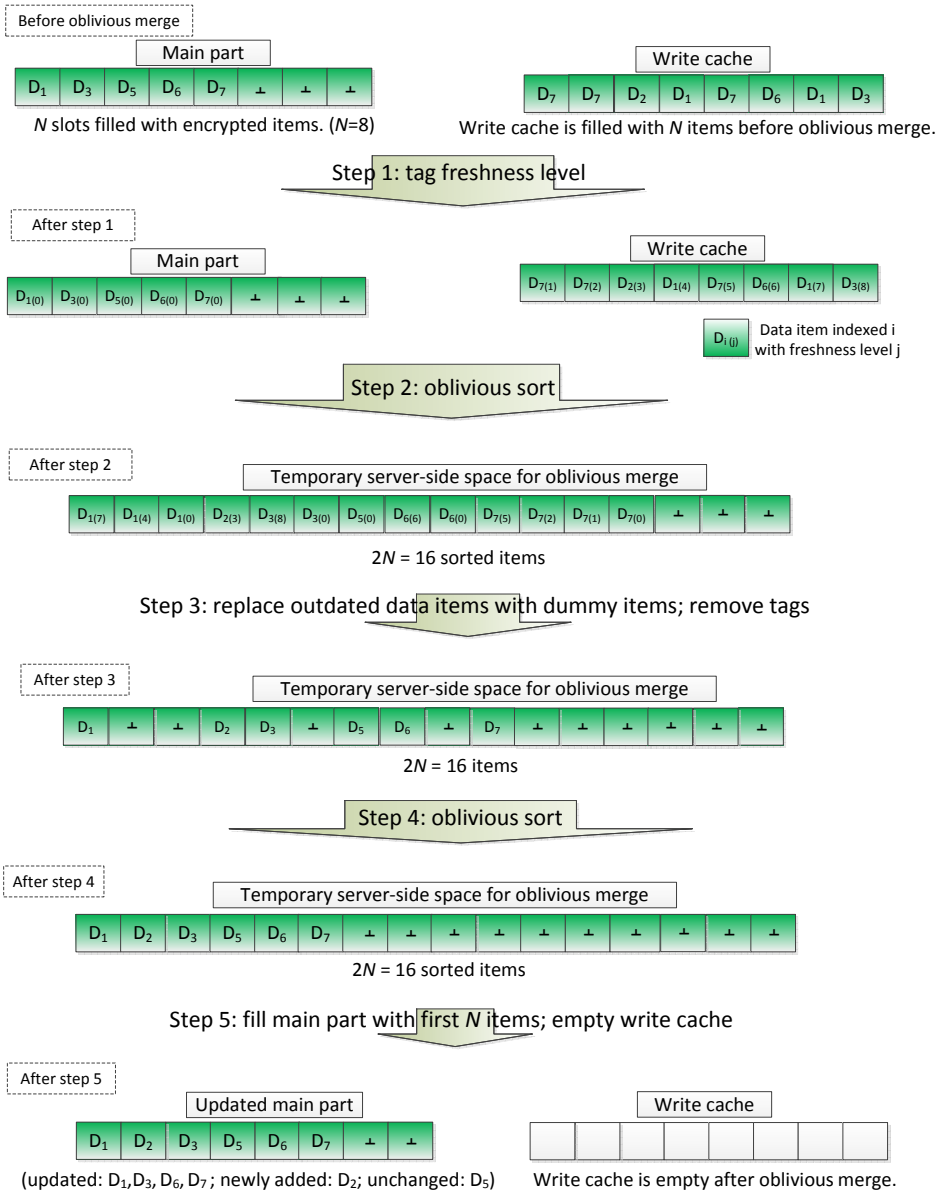

Figure 2. A toy example of oblivious merge based on oblivious sort

the data owner or a data consumer. A data reader may retrieve data items using PIR, or download all data items directly. Concurrent PIR reads will use PIR to retrieve data items from the old main part, old write cache and new write cache. A concurrent read of all data items will download all items in the old main part, old write cache and new write cache directly. When the oblivious merge is done, items in the old main part and old write cache will be deleted.

D. Key Sharing for Data Consumers and Multiple ORAM Clients

A symmetric encryption scheme like AES can be used in ORAM. In the outsourced data sharing scenario, cryptographic keys should be shared to data consumers so that they can decrypt retrieved data items. The data owner may have multiple ORAM clients (e.g. the data owner is a company and multiple employees access the data stored in the ORAM server). A client need to decrypt data/dummy items encrypted by other clients during an oblivious merge. Then keys should be

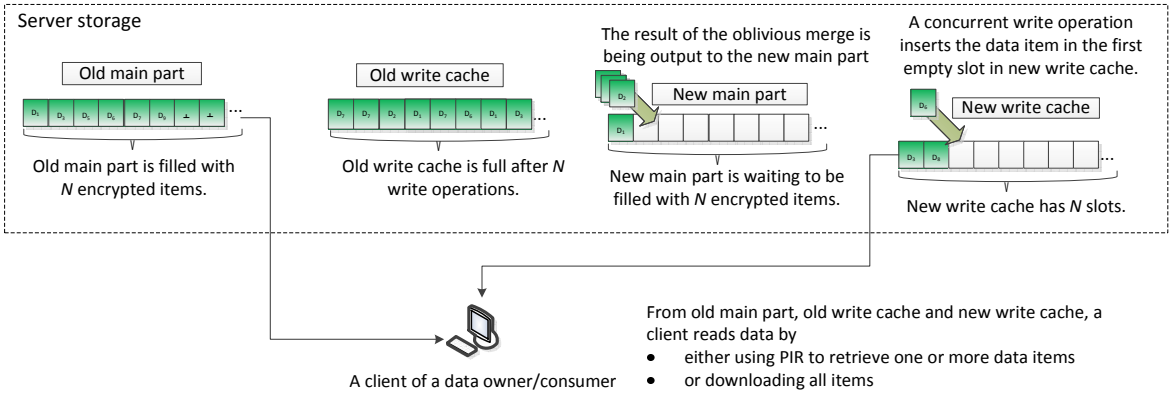

Figure 3. Concurrent read and write during an obvious merge

shared among ORAM clients too. The simplest way of managing and sharing keys is using the same key for all items like most ORAMs and pre-sharing the key like [16]. Alternative, [15]'s way can be used. [15] uses different keys for different items, and considers access control when sharing keys. By control the access to keys, the data owner can control which data items are accessible to a specific data consumer. Please refer to [15] for the details.

E. Analysis

During either an oblivious write operation or an oblivious merge, from the perspective of the server, the ORAM client's behaviors are independent of the data item being written and the data items been written before. The server does not know the index of any data item being written or having been written. The server cannot find the linkage between two write operations writing on the same data item. The server does not know if a given data item is updated more frequently than other items.

This write-only ORAM uses $O(N \times l)$ server-side storage. The communication and computational costs of an oblivious write are both $O(l)$. If the client-side storage usage is $O(l)$, the communication/compuatational cost of an oblivious merge is $O(N \times \log N \times l)$. If the client-side storage usage is $O(N^{1/r} \times l)$, the communication/compuatational cost of an oblivious merge is $O(N \times l)$. An oblivious merge is performed after every N oblivious writes. So the amortized communication and computational costs of an oblivious write are both: $O(l)$ when client-side storage is $O(N^{1/r} \times l)$; $O(\log N \times l)$ when client-side storage is $O(l)$. Here, $r \geq 2$ and r is a small constant.

F. Supporting PIR Clients

1) Maintaining Data Item Location for PIR Clients

For a client using PIR to retrieve a data item, as introduced in Section II, it needs to locate the data item's location in the server before using PIR. A table containing mappings from data item indices to their locations should be maintained. The table is called slot mapping table, which has N entries, and the i-th entry stores the i-th data item's location. If the i-th data item does not exist, the i-th entry stores a special value (e.g. -1) indicating the non-existence of the i-th data item. Entry size is $O(\log N)$ bits, and the table size is $O(N \times \log N)$ bits. To share the table among the clients of the data owner and data consumers, the mapping table is stored in the ORAM server. Each entry is encrypted with a semantically secure probabilistic encryption scheme. ORAM clients need to update the table obliviously, and PIR clients need to read the table obliviously. [15] has proposed an efficient method to read and update such a mapping table obliviously. We reuse its

method after two major modifications for reducing client-side storage usage, and summarize the modified method as below.

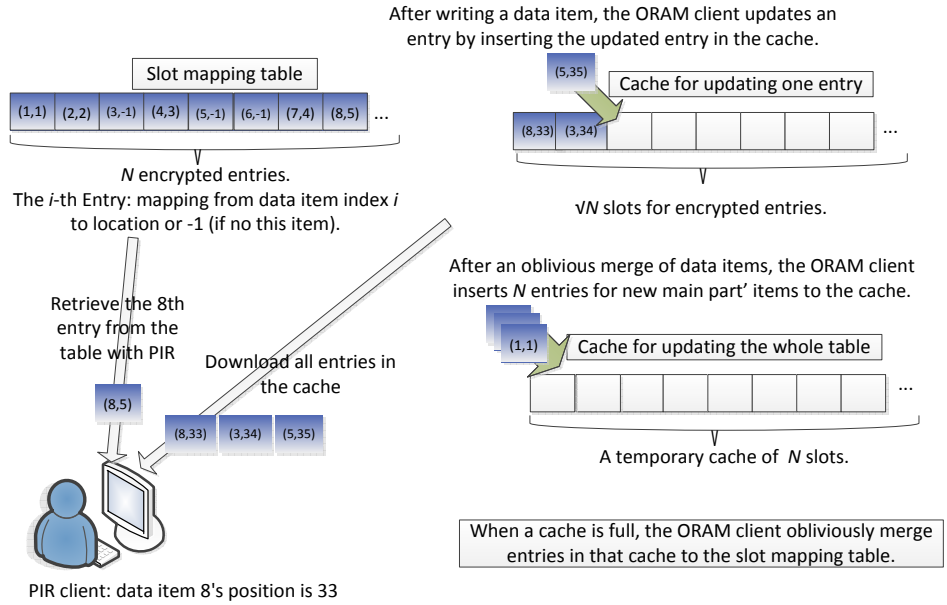

Figure 4. Slot mapping table for supporting PIR clients

As shown in Figure 4, the slot mapping table and two caches are maintained in the ORAM server. The original design of [15] has only one cache: the cache of $\sqrt{N}$ slots for updating one entry. We add a cache of N slots for updating the whole table. After an oblivious write of a data item, the ORAM client inserts the updated entry of the data item in the cache for updating one entry. To lookup a data item's location, the PIR client not only retrieves the item's entry in the table using PIR, but also downloads all entries in the cache for updating one entry. Then the amortized communication cost of a PIR read on a data item is $O(l + \sqrt{N} \times \log N)$. If the cache is full after inserting $\sqrt{N}$ entries, the ORAM client obliviously merges the entries in the cache and slot mapping table. In [15]'s original design, the ORAM client also stores a local slot mapping table with updated entries, and uploads the entries of the local table to replace the entries in the remote table. We make a modification to reduce client-side storage usage here: the client doesn't store a local slot mapping table; the client does an oblivious merge based on oblivious sort, which is the same as the oblivious merge algorithm for data items in Section IV-C. To enable concurrent read/write during the oblivious merge, an additional empty cache can be used for concurrent writes, and concurrent reads can retrieve entries from the slot mapping table, the full cache and the additional cache. This is very similar to the way of enabling concurrent read/write of data items. Another modification we make is the update of slot mapping table during an oblivious merge of data items. The server uses a cache of N entries for the update. During an oblivious merge of data items, N data/dummy items are put in the new main part. The client inserts the entries of these N items in this cache. If an item is a dummy item, a dummy entry for the dummy item is inserted. After the cache is full, the client does an oblivious merge to update the slot mapping table with entries in the cache. Again, this oblivious merge is based on oblivious sort, which is the same as the oblivious merge algorithm for data items in Section IV-C. Then communication and computational costs of an oblivious merge of entries are both: $O(N \times r \times \log N) = O(N \times \log N)$ if using $O(N^{1/r} \times \log N)$ client-side storage; $O(N \times (\log N)^2)$ if using $O(\log N)$ client-side storage. $r \geq 2$ but r is a small constant, which can be omitted in $O(N \times r \times \log N)$.

An oblivious merge of slot mapping entries is performed after every $\sqrt{N}$ oblivious ORAM writes and during every oblivious merge of data items. Recall that the amortized communication cost of an oblivious write without sharing slot mapping table is: $O(l)$ when client-side storage is $O(N^{1/r} \times l)$; $O(\log N \times l)$ when client-side storage is $O(l)$. Normally, $N^{1/r} \times l > N^{1/r} \times \log N$. Then, if sharing the slot mapping table, the amortized communication cost of an oblivious write is increased to: $O(l+\sqrt{N}\times\log N)$ when client-side storage is $O(N^{1/r}\times l)$; $O(\log N\times(l+\sqrt{N}))$ when client-side storage is $O(l + N^{1/r} \times \log N)$; $O(\log N \times l + \sqrt{N} \times (\log N)^2) = O(\sqrt{N} \times (\log N)^2)$ when client-side storage is $O(l)$ and l's order of magnitude is lower than $N^{1/r} \times \log N$'s. $r \geq 2$ and r is a small constant here. The computational cost of retrieving a data item and a slot mapping table entry with PIR are $O(N \times l)$ and $O(N \times \log N)$ respectively. Normally, $N \times l \geq N \times \log N$. So the amortized computational cost of retrieving a data item with PIR is still $O(N \times l)$.

2) Multi-server PIR Support

Multi-server PIR requires that each storage has the exact replication of the database. In another word, at any given position of the database, every server stores the same item. To use ORAM and multi-server PIR together, every server needs to store the identical encrypted item at the same position. The ORAM client can run the single-server write-only ORAM in one server, and synchronizes the changes of the database to every other server. Then, to use with a k-server PIR, the communication cost of ORAM is increased at most k times. k is regarded as a small constant. The communication cost is proportionally increased, but the order of magnitude thus stays the same.

V. Advanced Single-server Write-only ORAM

A. Construction

Using $O(N^{1/r} \times l)$ bits client-side storage, basic write-only ORAM can achieve an amortized communication cost of $O(l)$ bits. Advanced write-only ORAM can also achieve $O(l)$ bits amortized communication cost by employing basic write-only ORAM as a building block. Advanced write-only ORAM requires less client-side storage usage to achieve this cost when $O(N \times \log N) < O(N^{1/r} \times l)$. Here, $r \geq 2$ and r is a small constant.

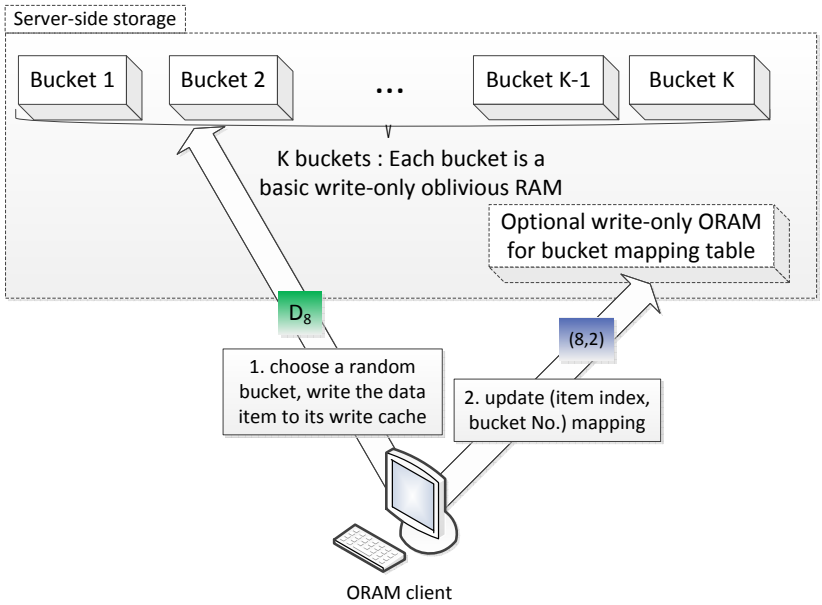

Figure 5. Advanced write-only ORAM and an oblivious write example

As shown in Figure 5, the server-side storage contains K buckets ($K < N$), and each bucket is a basic write-only ORAM that can store at most B data items. So the bucket size is $O(B \times l)$ bits. The values of K and B depend on N and l, which will be discussed later. A table named bucket mapping table containing the mappings from item indices to bucket numbers is maintained at the client-side or server-side. The table has N entries, and the i-th entry stores the i-th data item's bucket number. If the i-th data item does not exist, the i-th entry stores a special number (e.g. -1) indicating the non-existence of the data item. If there is only one ORAM client, the table is maintained at the client-side. Otherwise, as shown in Figure 5, all table entries are maintained at the server-side in an additional basic write-only ORAM. An entry contains a data item index and a bucket number. Then the entry size is $O(\log N + \log K) = O(\log N)$ bits, and the ORAM size (server-side storage for the ORAM) is $O(N \times \log N)$ bits.

In advanced write-only ORAM, an ORAM client obliviously writes a data item to the outsourced storage by following steps. First, it encrypts the data item locally with a semantically secure probabilistic encryption scheme. Second, it chooses a bucket uniformly at random, and oblivious writes the data item to the bucket ORAM. Third, it updates the data item's bucket number in a table named bucket mapping table, which stores the mappings from all data items' indices to their bucket numbers.

If the bucket mapping table is maintained in a basic write-only ORAM at the server side, an ORAM client can utilize $O(N \times \log N)$ client-side storage do a trivial oblivious merge of this basic write-only ORAM locally. Then the amortized communication cost of updating bucket mapping table is $O(\log N)$, which is less than $O(l)$ normally. So updating the data item's bucket number won't increase the amortized communication cost of writing data item no matter the bucket mapping table is maintained at client-side or server-side.

The client does a bucket's oblivious merge based on oblivious sort when the bucket's write cache is full. If a basic write-only ORAM have B data items, $O(B^{1/r} \times l)$ client-side storage is required to do an oblivious merge and achieve $O(l)$ amortized communication cost. However, a bucket is not just a basic write-only ORAM. To do an oblivious merge of a bucket, additional client-side storage and operations are also needed to detect and remove outdated data items in the bucket. The bucket mapping table is maintained for detecting outdated items. The client can lookup the bucket mapping table to see if all versions of the i-th data items in a bucket are outdated. If the i-th data item's bucket is not the current bucket, all versions are outdated. Otherwise, one of these versions is the most updated, which is determined by their freshness levels. If there is only one ORAM client, the bucket mapping table can be maintained at the client-side storage. Otherwise, the bucket mapping table should be shared via the ORAM server, and the client downloads the table before doing an oblivious merge. The bucket mapping table's size is $O(N \times \log N)$ bits. So the client-side storage usage is $O(N \times \log N + B^{1/r} \times l)$. An oblivious merge of a bucket is performed after every B writes to the bucket. Then downloading the bucket mapping table increases the amortized communication cost of data item writing by $O(N/B \times \log N)$. Therefore, the amortized communication cost of writing a data item is: still $O(l)$ if the bucket mapping table is maintained at the client side; $O(l + N/B \times \log N)$ otherwise. In the case that a bucket's main part will be overflowed after an oblivious merge, the client concurrently inserts each overflowed data item to a random chosen bucket by doing an oblivious write. As discussed below, B is set to a proper value to make overflow rare.

B. Parameter Optimization

Now we discuss how to choose the values of K and B. Our goal is to minimize client-side storage usage $O(N \times \log N + B^{1/r} \times l)$ while keeping amortized communication cost of an oblivious write as $O(l)$ and server-side storage usage as $O(N \times l)$. To keep server-side storage usage as $O(N \times l)$, we need to set B as $O(N/K)$. Based on the standard balls in bins analysis [33], the bucket with

most data items has $O(N/K)$ data items with high probability if $N \geq K \times \log K$. So we must choose K's value to make $N \geq K \times \log K$. As analyzed above, if the bucket mapping table is maintained at the client side, $O(l)$ amortized communication cost can be achieved. However, if the bucket mapping table is shared at the server for multiple clients, the amortized communication cost is $O(l + N/B \times \log N)$. In that case, we must make $O(l) \geq O(N/B \times \log N) = O(K \times \log N)$ to achieve $O(l)$ amortized communication cost. Therefore, to meet our goal, following conditions must be met: 1) $N \geq K \times \log K$; 2) $B = O(N/K)$; 3) $O(l) \geq O(K \times \log N)$ if there are multiple ORAM clients. Below, we show how to choose K's value to meet these conditions.

- If there are multiple ORAM clients and $N \geq l$, we choose K's value as the maximal integer that can make $l \geq K \times \log N$. As $N \geq l \geq K \times \log N > K \times \log K$, B's value can be set to $O(N/K)$. Therefore, the client-side storage usage is $O(N \times \log N + (N/K)^{1/r} \times l)$. As $O(l) = O(K \times \log N)$, we got $O((N/K)^{1/r} \times l) = O((N/(l/logN))^{1/r} \times l) = O(N^{1/r} \times l^{(r-1)/r} \times (\log N)^{1/r}) < O(N \times \log N)$. Then the client-side storage usage is $O(N \times \log N)$. As $O(l) \geq O(K \times \log N)$, the amortized communication cost of an oblivious write is $O(l)$.
- Otherwise, if there is only one ORAM client or $N < l$, we choose K's value as the maximal integer that can make $N \geq K \times \log N$. So B's value can be set to $O(N/K) = O(\log N)$. Therefore, the client-side storage usage is $O(N \times \log N + (\log N)^{1/r} \times l)$. Normally, $(\log N)^{1/r}$ can be viewed as a small constant. Then the client-side storage usage is $O(N \times \log N + l)$. Because $O(l) \geq O(N) \geq O(K \times \log N)$, the amortized communication cost of an oblivious write is $O(l)$ no matter there is only one ORAM client or not.

The communication cost and storage usage after optimization are as follows. The amortized communication cost of an oblivious write is $O(l)$. The server-side storage usage is $O(N \times l)$. If there are multiple ORAM clients and $N \geq l$, the client-side storage usage is $O(N \times \log N)$. Otherwise, the client-side storage usage is $O(N \times \log N + l)$.

C. Security Analysis

All the operations in advanced write-only ORAM are oblivious. Each bucket is a basic write-only ORAM. If the bucket mapping table is maintained at the server side, table entries are stored in a basic write-only ORAM. Operations on any bucket or the bucket mapping table at the server side are oblivious. The types of performed operations (write or oblivious merge) and the operated buckets' numbers are independent of the sequence of written data items' indices. Any two write operation sequences with the same length are indistinguishable to the server.

D. Supporting PIR Clients

Supporting PIR clients in basic write-only ORAM and advanced write-only ORAM are the same. The ways of maintaining data item location and supporting multi-server PIR are the same. In basic write-only ORAM, a mapping table called slot mapping table storing data item locations is maintained. Please note that, in advanced write-only ORAM, we use only one slot mapping table for all buckets instead of maintaining a separate table for each bucket. This table stores the mapping from data item indices to their exact locations in advanced write-only ORAM. The size of the slot mapping table is still $O(N \times \log N)$.

As discussed in Section IV-F, the amortized communication cost of a PIR read is $O(l + \sqrt{N} \times \log N)$. Section IV-F also shows that the amortized communication cost of an oblivious write is increased by $O(\sqrt{N} \times \log N)$ if using $O(N^{1/r} \times \log N)$ client-side storage to support PIR clients. $r \geq 2$ and r is a small constant. So, to support PIR, the amortized communication cost and client-side storage usage are increased by $O(\sqrt{N} \times \log N)$ and $O(N^{1/r} \times \log N)$ respectively. However, the increased client-side storage usage is negligible compared with the usage without PIR support. Therefore, the amortized communication cost of an oblivious write is increased to

$O(l + \sqrt{N} \times \log N)$, and client-side storage usage's order of magnitude is not increased. The computational costs of retrieving a data item and a data item's location with PIR are $O(N \times l)$ and $O(N \times \log N)$ respectively. Normally, $N \times l \geq N \times \log N$. So the amortized computational cost of retrieving a data item with PIR is $O(N \times l)$.

VI. Multi-server Write-only ORAM

In the outsourced data sharing scenario, as proposed by [15], t-private k-server itPIR can be used with ORAM to hide data access pattern, where $k > t \geq 1$. Recall that the use of t-private k-server itPIR requires k outsourced storages, e.g. storages provided by k cloud providers, and the number of colluding storages are no more than t. t-private k-server itPIR also requires that each storage has the exact replication of the database. In another word, at any given position/address of the database, every server stores the same item. As discussed in Section II-B and IV-F, single-server ORAM scheme can be used with multi-server PIR scheme: the ORAM client runs a single-server write-only ORAM scheme in one server, and synchronizes the changes of the database to every other server. Alternatively, giving the assumption that the number of colluding storages are no more than t, we can design a t-private k-server write-only ORAM scheme, and use it with t-private k-server itPIR.

In this section, we design a multi-server write-only ORAM scheme based on the basic single-server write-only ORAM. Compared with basic write-only ORAM, this multi-server write-only ORAM has lower client-side storage usage when $O(l + N) < O(N^{1/r} \times l)$. This ORAM's client-side storage usage is also lower than advanced write-only ORAM's. In addition to being used with multi-server PIR, multi-server write-only ORAM may also be used in the case that data consumers download all data items trivially.

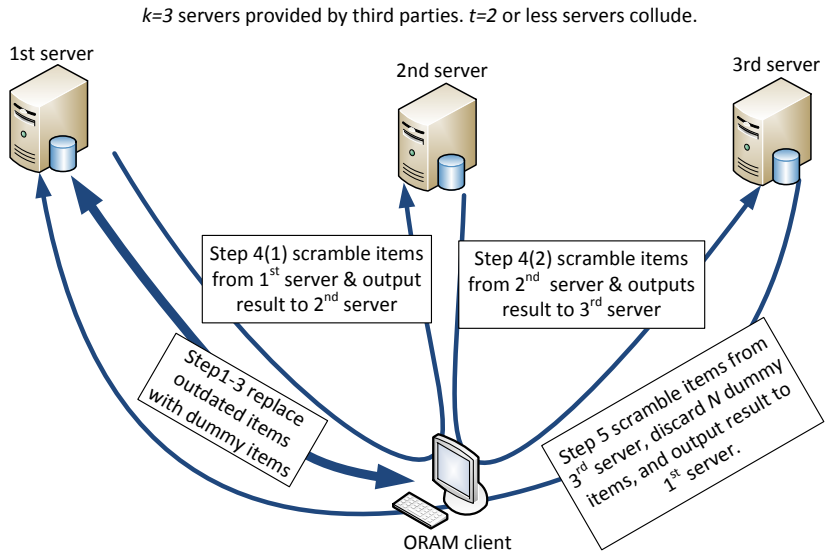

Figure 6. Oblivious merge in multi-server write-only oblivious RAM

The only difference between multi-server write-only ORAM and basic write-only ORAM is oblivious merge. Suppose there are k servers, and the number of colluding servers are no more than t. Multi-server write-only ORAM scheme utilizes $t+1$ servers to do an oblivious merge. The same as applying single-server write-only ORAM to multiple servers, the client runs the write-only ORAM in one server, and synchronizes the changes of the database to every other server. The synchronization is required by the use of multi-server PIR, but is optional for data consumers downloading all data items trivially. When the ORAM's write cache is full, the client utilizes $t+1$

servers to do an oblivious merge. Suppose the server running ORAM is indexed 1. The client chooses other t servers randomly. We index these servers from 2 to $t+1$. As illustrated in Figure 6, the ORAM client does the oblivious merge based on oblivious remove and oblivious scramble by following steps. In these steps, items are encrypted with a semantically secure probabilistic encryption scheme.

- First, the client creates a bitmap of N bits. The bitmap is initialized as all zeros. The bitmap is used to detect outdated data items. Let's call this bitmap detective bitmap.
- Second, the client scans the write cache of the first server from tail to head, replaces outdated data items with dummy items. If data consumers use PIR, to keep every server's database identical and allow concurrent PIR read, the client operates on a copy of the write cache in a temporary space of the server instead of the write cache. Because the cache is scanned from tail to head, more recently written data items in the write cache are scanned before less recently written data items. During the scan, if a data item indexed i is found, the client checks the i-th bit of the detective bitmap. If the i-th bit is zero, which means the data item is the most updated version of the i-th data item, the client sets the i-th bit as 1. Then client re-encrypts the data item, and puts it back to its slot in the write cache. If the i-th bit of the detective bitmap is 1, which means a more updated i-th data item exists, the client puts a dummy item back to the data item's slot in the write cache.
- Third, the client scans the main part of the first server, replaces outdated data items with dummy items. If data consumers use PIR, to allow concurrent PIR read, the client operates on a copy of the main part in the server instead of the main part. The client detects and replaces outdated data items by utilizing the detective map, which is very similar to that in the second step. During the scan, if a data item indexed i is found, the client checks the i-th bit of the detective bitmap. If the i-th bit is zero, the client sets the i-th bit as 1. Then the client re-encrypts the data item, and puts it back to its slot in the main part. If the i-th bit of the detective bitmap is 1, the client puts a dummy item to the data item's slot in the main part.
- Fourth, the client scrambles all items t times utilizing $t+1$ servers. The client downloads the items in the first server's main part and write cache one by one in a random order. After the second and third steps, all data items in the downloaded items are the most updated. During the download, the client re-encrypts the downloaded items, uploads them to the second server, and removes uploaded items from local storage. Then only $O(l)$ client-side storage is required in this step. The uploaded items are stored in a temporary space in the second server. After all items being uploaded to the second server, the first scramble is finished. Using the same way, the client scrambles the items uploaded to the second server, and uploads the result of the second scramble to the third server. Using the same way, the client scrambles the items $t-2$ times more, and eventually the result is stored in a temporary space in the $(t+1)$-th server.
- Fifth, the client scrambles all items in the $(t+1)$-th server, and outputs data items to the first server. The client downloads the items in the $(t+1)$-th server, which have been scrambled t times, in a random order. During the download, the client re-encrypts downloaded data items, uploads them to the first server, and removes uploaded items and dummy items from local storage. The uploaded items are put in the new main part of the first server. If data item count is less than N, the client also uploads one or more new dummy items to fill the new main part during the upload of data items. The data item count can be measured during the second and third step, and stored with a counter of $\lceil \log N \rceil$ bits. If data consumers use PIR, the change of the ORAM is also synchronized to other servers by the client.

Analysis After first three steps, outdated data items are replaced with dummy items, and updated

data items are kept in their original positions before the oblivious merge. No information about the write pattern leaks in the first three steps because the ORAM client's behaviors are independent of the write pattern. In the fourth and fifth step, items in the database are scrambled $t+1$ times. After $t+1$ scrambles, updated data items and maybe some dummy items are outputted to the first server as the result of the oblivious merge, and N dummy items of the $(t+1)$-th scramble's result are discarded. By observing the upload speed of outputting the oblivious merge's result, the first server learns information about positions of updated data items and dummy items in the $(t+1)$-th scramble's result. If the first server can correlate the items before and after the $t+1$ scrambles by colluding with some servers, it knows information about updated data items' positions before oblivious merge and write pattern. However, if no more than t servers collude, no server can learn the information. The i-th server can correlate the items before and after the i-th scramble only. If any t of the k servers collude, they can correlate at most t scrambles' input items and output items, but can't correlate the items before and after $t+1$ scrambles. Therefore, if colluding server count is no more than t, the oblivious merge is secure.

The client-side and server-side storage usages are $O(N+l)$ and $O(N \times l)$ bits respectively. t and k are small integers, and viewed as constants in this paper. Then the communication cost of an oblivious merge is $O(N \times l)$. An oblivious merge is performed after every N oblivious writes. So the amortized communication cost of oblivious write is $O(l)$.

Supporting PIR clients in multi-server write-only ORAM is the same as that in single-server write-only ORAM. To support PIR, the amortized communication cost is increased by $O(\sqrt{N} \times \log N)$ using additional $N^{1/r} \times \log N$ client-side storage usage. This is the same as the case in the basic write-only ORAM. So, to support PIR, the amortized communication cost of oblivious write is increased to $O(l + \sqrt{N} \times \log N)$ and client-side storage is $O(N + l + N^{1/r} \times \log N) = O(N + l)$. The same as the basic write-only ORAM, in multi-server write-only ORAM, the amortized communication and computational costs of PIR read are $O(l + \sqrt{N} \times \log N)$ and $O(N \times l)$ respectively.

VII. Full Functional ORAM using Write-only ORAM as a Building Block

[4] and [5] proposed ORAMs containing multiple buckets. [4]'s bucket is a full functional ORAM storing at most $O(\log N)$ data items, while [5]'s bucket is a full functional ORAM storing $O(\sqrt{N})$ data items. We can view their designs as frameworks using other full functional ORAM schemes as building blocks. Most traditional full functional ORAM schemes and our write-only ORAM schemes (together with PIR) can be used to implement these buckets with slight modifications. Using our write-only ORAM and PIR as a build block in [4] and [5]'s ORAMs, the communication cost can be reduced, and the increased computational cost could be acceptable or even negligible for many reasonable values of N and l.

[4] and [5]'s bucket ORAM is a bit different from normal full functional ORAM. [4] and [5] require a bucket ORAM to provide oblivious read-and-remove primitive and add primitive instead of read primitive and write primitive. The oblivious read-and-remove primitive can read and remove a real data item or a dummy data item from a bucket, while the oblivious add primitive can add a real data item or a dummy data item to a bucket. It is not required to hide an primitive's type. Building [4]'s bucket ORAM and [5]'s bucket ORAM based on write-only ORAM and PIR are similar but not exactly the same. We make slight modifications to [4]'s ORAM framework and our write-only ORAM. Next, we describe how to build bucket ORAM based on write-only ORAM for [5] and [4] separately.

A. Using Write-only ORAM in [5]'s ORAM Framework

In [5], there is only one client, and it uses $O(\sqrt{N} \times l + N \times \log N)$ bits local storage. The client maintains $\sqrt{N}$ queues locally. The i-th queue stores the data items to be added to the i-th bucket

later. Locally, the client also maintains a data location table, which contains each data item's exact location. A read/write operation of a data item indexed i is simulated by the following steps. First, the client looks up the data location table locally to get the i-th data item's location. Second, if the i-th data item is in a local queue, the client removes the data item from the queue. Third, the client performs an oblivious read-and-remove operation on a bucket. The client performs a read-and-remove operation on the i-th data item's bucket to get the data item if it exists in the server. Otherwise, the client performs a read-and-remove operation on a random bucket to read a dummy item. Fourth, if the client has obtained the most updated data value of the i-th data item, the client chooses a random bucket for the i-th data item, and puts the i-th data item with its value in the bucket's queue. If the simulated operation is a write operation, the data value is given by the operation. Otherwise, the data value is read from a local queue in the second step or a bucket in the third step. Fifth, the client updates the i-th data item's location in the data location table unless the simulated operation is a read operation and the i-th data item is not found.

To prevent queues becoming full, the client carries out a background eviction process continuously. An eviction process may remove a data item from a bucket's queue, and add the item to the bucket using the oblivious add primitive. An eviction process may add a dummy item obliviously to a bucket so as to hide that the bucket's queue is empty. To hide access pattern, [5] designed several eviction algorithms to schedule eviction processes.

We choose to use our basic write-only ORAM to build the bucket ORAM in [5]'s ORAM framework. As the client has enough storage space to store a bucket's items locally, the client can download all items in a bucket, and do the oblivious merge locally. There is no need to use the more complicated advanced write-only ORAM here. In the bucket ORAM based on basic write-only ORAM and PIR, the ORAM client is both a write-only ORAM client and a PIR client. Because the client already maintains each data item's location locally in [5]'s design, there is no need to maintain write-only ORAM's slot mapping table. To read-and-remove a data item from a bucket ORAM, the client first looks up the data item's location inside the bucket locally, and then retrieves the data item from the bucket using PIR. For our write-only ORAM, there is no need to do an explicit remove because our ORAM can detect outdated data items by looking up the data location table. To read-and-remove a dummy item from a bucket ORAM, the client simply retrieves a random item (data item or dummy item) using PIR. To add a data item or dummy item to a bucket ORAM, the client simply does an oblivious write by putting the item in the bucket's write cache. The amortized communication and computational costs of a read-and-remove operation on a bucket are both $O(l)$, while the amortized communication and computational costs of an add operation on a bucket are $O(l)$ and $O(\sqrt{N} \times l)$ respectively.

During an oblivious read/write, a read-and-remove operation and an add operation are performed. Using write-only ORAM and PIR to build the buckets in [5]'s ORAM framework, the amortized communication cost of an oblivious read/write is $O(l)$, and the amortized computational cost is $O(\sqrt{N} \times l)$. If using traditional full functional ORAM to build buckets, the best amortized communication and computational costs are both $O(\log N \times l)$. We can see that using write-only ORAM makes a tradeoff of computational cost for communication cost. What values of N and l can make this tradeoff worthy needs further work to find out.

B. Using Write-only ORAM in [4]'s ORAM Framework

We omit [4]'s the design here, and only introduce some of [4]'s properties that affects our building of bucket ORAM. Please refer to [4] for the design details. [4]'s design contains a basic construction and a recursive construction based on the basic construction. We can use write-only ORAM to build the buckets of both the basic and recursive constructions, and reduce both constructions' communication costs. However, compared with the best known traditional ORAM schemes, the

performance of the recursive construction using write-only ORAM is not better. So we introduce only the basic construction, and show how to build buckets for the basic construction only.

In [4]'s original design, there are $O(N)$ buckets organized in a binary tree, and each node in the tree is a bucket storing at most $O(\log N)$ items. The server-side storage usage is $O(N \times \log N \times l)$, and the tree has $h = O(\log N)$ levels. These parameters can be optimized to reduce server-side storage usage [45]. After optimization, there are $O(N/\log N)$ buckets, and each bucket stores at most $O(\log N)$ items. Then, the server-side storage usage is reduced to $O(N \times l)$ and the tree has $h = O(\log(N/\log N)) = O(\log N)$ levels. The simulation of a read/write operation contains following work: the client chooses $O(h)$ buckets, and performs a read-and-remove operation to get a data item or dummy item from each of these buckets; the client chooses $O(h)$ buckets, and adds a data or dummy item to each of the chosen buckets using the add primitive. A data item being added to a bucket is an item not existed in any bucket currently. The item is either never stored in any bucket before, or just removed from a bucket using the read-and-remove primitive.

In [4]'s basic construction, the client locally maintains a data structure, called index structure by [4], containing entries for all data items. A data item's entry contains $O(\log N)$ bits data. So the client storage usage is $O(N \times \log N)$. The client needs the index structure to decide the choices of buckets. A data item's entry is read and updated once during an oblivious read/write. To reduce client-side storage usage, we make a slight modification here. Instead storing the index structure at the client side, we can use an additional full functional ORAM store all entries of the index structure. For example, we can use an ORAM from [3, 16, 17], and item size of this ORAM is $\log N$. Then, according to Table I, the client-side storage and amortized communication cost for this ORAM are $O(N^{1/r} \times \log N)$ and $O(\log N \times \log N)$ respectively.

We can build a bucket ORAM for [4]'s basic construction based on our write-only ORAM and PIR. If the client uses $O((\log N)^{1/r} \times l)$ local storage space, which is acceptable in many situations, a bucket can be built based on of basic write-only ORAM. So we prefer to choose the simpler basic write-only ORAM to build buckets. In [4]'s design, the client doesn't have a data location table contains each data item's location. But the client of our bucket ORAM needs to locate data item location. Instead of using the slot mapping table, we use a table called property table to locate data item location inside a bucket ORAM. The property table is also used to detect outdated data items during oblivious merges. The server additionally stores a property table for each bucket containing the properties of all items in the bucket. The i-th entry of a bucket's property table stores the properties of the item in the i-th slot inside the bucket. The item properties include item type (data item or dummy item), freshness (outdated or updated), and item index. If the item in i-th slot is a dummy item, freshness value and index value can be set to some random values. Each property table is encrypted with a semantically secure probabilistic encryption scheme. The size of a property table is $O((\log N)^2)$ bits. Usually, $O((\log N)^2)$ is not bigger than $O(l)$. When full functional ORAM is used in outsourced storage, it is usually used as a block storage system or the underlying layer of a file system [8, 35]. Usually, the item size is at least several KB. Suppose l is 8192 bits (1KB). N has to be bigger than 2^{90} to make $(\log N)^2$ bigger than l, which is unlikely.

To read-and-remove a data item from a bucket ORAM, the client does following steps. First, the client downloads the bucket's property table, decrypts it, and looks up the data item's location inside the bucket locally. Second, the client retrieves the data item from the bucket using PIR. Third, the client updates the property table by setting the item's freshness property as outdated, and uploads the table after re-encrypting it. Instead of removing the data item directly, we virtually remove it by setting its freshness property as outdated here. To read-and-remove a dummy item from a bucket ORAM, the client does following steps. First, the client downloads the bucket's property table, and decrypts it. Second, the client retrieves a random item (data item or dummy item) from the bucket using PIR. Third, the client uploads the property table after re-encrypting it. To add a data item

or dummy item to a bucket ORAM, the client does following steps. First, the client downloads the bucket's property table, and decrypts it. Second, the client obliviously write the item to the bucket. Third, the client updates the property table by adding the item's properties, and uploads the table after re-encrypting it. The amortized communication and computational costs of an add operation on a bucket are both $O(l)$, while the amortized communication and computational costs of a read-and-remove operation on a bucket are $O(l)$ and $O(\log N \times l)$ respectively.

In [4]'s basic construction, an original data read/write operation is simulated with $O(\log N)$ read-and-remove and add operations on buckets. One read access and one write access to an entry of the index structure are also required in the simulation of an original data operation. Recall that we modify the basic construction by adding an additional ORAM for storing its index structure. The client-side storage and amortized communication cost for this additional ORAM are $O(N^{1/r} \times \log N)$ and $O((\log N)^2)$ respectively. Using write-only ORAM and PIR to build the buckets in [4]'s basic construction, the amortized communication cost of an oblivious read/write is $O(\log N \times l + (\log N)^2) = O(\log N \times l)$, and the amortized computational cost is $O((\log N)^2 \times l)$. The client-side storage usage is $O((\log N)^{1/r} \times l + N^{1/r} \times \log N)$. $(\log N)^{1/r}$ can be viewed as a small constant in practice. Then the client-side storage usage is $O(l + N^{1/r} \times \log N)$. The server-side storage usage is $O(N \times l)$ after optimization [45].

[4] uses two kinds of traditional ORAM-based buckets in its basic ORAM construction. The best one is based on [1]'s Square-Root ORAM. If building buckets based on [1]'s Square-Root ORAM, the amortized communication and computational costs are both $O(\log N \times \sqrt{\log N} \times \log\log N \times l)$. If improving [4]'s basic construction as introduced in this section (adding an additional ORAM for storing its index structure and optimizing server-side storage usage), the server-side and client-side storage usages are the same as our result. Therefore, using write-only ORAM makes a tradeoff of computational cost for communication cost.

VIII. Concluding Remarks

In this paper, two single-server write-only ORAM schemes and one multi-server write-only ORAM scheme have been proposed to hide the write pattern on honest but curious outsourced storages. This paper has discussed write-only ORAM's usages in two scenarios: the data owner sharing data to data consumers via outsourced storage; outsourced storage for the data owner's own usage. To hide both read and write patterns, PIR is used together with write-only ORAM in first scenario, and full functional ORAM based on write-only ORAM and PIR is used in the second scenario. Write-only ORAM may be used alone in some cases of the first scenario where only write pattern need to be hidden. In addition to proposing three write-only ORAM schemes, this paper have studied supporting PIR in ORAM as well as building full functional ORAM based on write-only ORAM and PIR. The stroage/communication/computational costs of write-only ORAM schemes have been estimated and compared with traditional ORAM's costs in above scenarios. Compared with using traditional ORAM, using write-only ORAM in the above scenarios has lower communication cost or much less client-side storage usage. However, using write-only ORAM in the second scenario has higher computational cost. Real-life experiments are still needed to evaluate how these costs affect data operation throughput and data access latency under different settings of data item count and data item length. We leave this as a future work.

References

[1] O. Goldreich. "Towards a Theory of Software Protection and Simulation by Oblivious RAMs." In STOC 1987.

[2] B. Pinkas and T. Reinman. "Oblivious RAM Revisited." In CRYPTO 2010.

[3] M. T. Goodrich and M. Mitzenmacher. "Privacy-preserving Access of Outsourced Data via Oblivious RAM Simulation." In ICALP 2011.

[4] E. Shi, T. H. Chan, E. Stefanov, and M. Li. “Oblivious RAM with $O((\log N)^3)$ Worst-case Cost.” In ASIACRYPT 2011.
[5] E. Stefanov, E. Shi, and D. Song. “Towards Practical Oblivious RAM.” In NDSS 2012.
[6] D. Boneh, D. Mazieres, and R. A. Popa. “Remote Oblivious Storage: Making Oblivious RAM Practical.” Technical Report MIT-CSAIL-TR-2011-018, 2011. http://dspace.mit.edu/bitstream/handle/1721.1/62006/MIT-CSAIL-TR-2011-018.pdf
[7] M. Franz, P. Williams, B. Carbunar, S. Katzenbeisser, A. Peter, R. Sion, and M. Sotakova. “Oblivious Outsourced Storage with Delegation.” In FC 2012.
[8] P. Williams, R. Sion, and A. Tomescu. “PrivateFS: a Parallel Oblivious File System.” In CCS 2012.
[9] B. Chor, O. Goldreich, E. Kushilevitz, and M. Sudan. “Private Information Retrieval.” In FOCS 1995.
[10] D. Asonov. “Private Information Retrieval: An Overview and Current Trends.” In the ECDPvA Workshop, 2001.
[11] L. Sassaman, B. Cohen, and N. Mathewson. “The Pynchon Gate: A Secure Method of Pseudonymous Mail Retrieval.” In WPES 2005.
[12] S. B. Mane, S. T. Sawant, and P. K. Sinha. “Using Private Information Retrieval Protocol for an E-commerce Application.” In CUBE Intl. Inf. Tech. Conf., 2012.
[13] A. M. Miceli, B. J. Sample, C. E. Ioup, and D. M. Abdelguerfi. “Private Information Retrieval in an Anonymous Peer-to-Peer Environment.” In SAM 2011.
[14] S. Wang, D. Agrawal, and A. E. Abbadi. “Generalizing PIR for Practical Private Retrieval of Public Data.” In DBSec 2010.
[15] Y. Huang and I. Goldberg. “Outsourced Private Information Retrieval with Pricing and Access Control.” Technical Report CACR 2013-11. http://cacr.uwaterloo.ca/techreports/2013/cacr2013-11.pdf
[16] M. T. Goodrich, M. Mitzenmacher, O. Ohrimenko, and R. Tamassia. “Privacy-preserving Group Data Access via Stateless Oblivious RAM Simulation.” In SODA 2012.
[17] M.T. Goodrich, M. Mitzenmacher, O. Ohrimenko, and R. Tamassia. “Oblivious RAM Simulation with Efficient Worst-case Access Overhead.” In CCSW 2011.
[18] E. Stefanov, M. v. Dijk, E. Shi, C. Fletcher, L. Ren, X. Yu, and S. Devadas. “Path O-RAM: an Extremely Simple Oblivious RAM Protocol.” Technical Report arXiv:1202.5150v2, http://arxiv.org/pdf/1202.5150v2
[19] E. Kushilevitz, S. Lu, and R. Ostrovsky. “On the (In)security of Hash-based Oblivious RAM and a New Balancing Scheme.” In SODA 2012.
[20] S. Lu and R. Ostrovsky. “Distributed Oblivious RAM for Secure Two-party Computation.” In TCC 2013.
[21] A. Beimel, Y. Ishai, and E. Kushilevitz. “General Constructions for Information-theoretic Private Information Retrieval.” In Journal of Computer and System Sciences 71.2 (2005).
[22] S. Yekhanin. “Towards 3-query Locally Decodable Codes of Subexponential Length.” In STOC 2007.
[23] C. Gentry and Z. Ramzan. “Single-Database Private Information Retrieval with Constant Communication Rate.” In ICALP 2005.
[24] J. Trostle and A. Parrish. “Efficient Computationally Private Information Retrieval from Anonymity or Trapdoor Groups.” In ISC 2011.
[25] A. Beimel, Y. Ishai, E. Kushilevitz, I. Orlov. “Share Conversion and Private Information Retrieval.” In CCC 2012
[26] A. Iliev and S. Smith. “Private Information Storage with Logarithmic-space Secure Hardware.” In Information Security Management, Education and Privacy, 2004.

[27] A. Iliev and S. Smith. “Protecting Client Privacy with Trusted Computing at the Server.” In S&P 2005.
[28] S. Wang, X. Ding, R. H. Deng, and F. Bao. “Private Information Retrieval using Trusted Hardware.” In ESORICS 2006.
[29] P. Williams and S. Radu. “Usable PIR.” In NDSS 2008.
[30] T. Mayberry, E. O. Blass, A. Chan. “Efficient Private File Retrieval by Combining ORAM and PIR.” In Cryptology ePrint Archive: Report 2013/086, 2013.
[31] H. Lipmaa, and B. Zhang. “Two New Efficient PIR-Writing Protocols.” In ACNS 2010.
[32] M. T. Goodrich. “Randomized Shellsort: a Simple Oblivious Sorting Algorithm.” In SODA 2010.
[33] M. Raab and A. Steger. ““Balls into Bins” - a Simple and Tight Analysis.” In RANDOM 1998.
[34] I. Damgard, S. Meldgaard, and J. B. Nielsen. “Perfectly Secure Oblivious RAM without Random Oracles.” In TCC 2011.
[35] E. Stefanov and E. Shi. “ObliviStore: High Performance Oblivious Cloud Storage.” In S&P 2013.
[36] F. Olumofin, I. Goldberg. “Revisiting the Computational Practicality of Private Information Retrieval.” In FC 2011.
[37] R. Sion, and B. Carbunar. “On the Computational Practicality of Private Information Retrieval.” In NDSS 2007.
[38] C. A. Melchor and P. Gaborit. “a Lattice-Based Computationally-Efficient Private Information Retrieval Protocol.” In WEWORC 2007.
[39] C. A. Melchor, B. Crespin, P. Gaborit, V. Jolivet, and P. Rousseau. “High-Speed Private Information Retrieval Computation on GPU.” In SECURWARE 2008.
[40] C. Devet. “Evaluating Private Information Retrieval on the Cloud.” Technical Report CACR 2013-05, 2013. http://cacr.uwaterloo.ca/techreports/2013/cacr2013-05.pdf.
[41] T. Mayberry, E. O. Blass, and A. H. Chan. “Pirmap: Efficient Private Information Retrieval for Mapreduce.” In FC 2013.
[42] I. Goldberg. “Improving the Robustness of Private Information Retrieval.” In S&P 2007.
[43] Goldwasser, Shafi, and Silvio Micali. “Probabilistic Encryption.” In Journal of Computer and System Sciences 28.2 (1984).
[44] M. T. Goodrich. “Data-oblivious External-memory Algorithms for the Compaction, Selection, and Sorting of Outsourced Data.” In SPAA 2011.
[45] C. Gentry, K. A. Goldman, S. Halevi, C. Julta, M. Raykova, and D. Wichs. “Optimizing ORAM and Using it Efficiently for Secure Computation.” In PETS 2013.

APPENDIX A
FULL FUNCTIONAL ORAM DIRECTLY BASED ON WRITE-ONLY ORAM AND PIR

A. Full Functional ORAM directly based on Basic Single-server Write-only ORAM and PIR

We can build a full functional ORAM based on the basic write-only ORAM and PIR directly. In this full functional ORAM, an ORAM client is both a write-only ORAM client and a PIR client.

A read operation on the i-th data item is simulated by following steps. First, the client locates the i-th data item's slot by obviously looking up the slot mapping table. Second, the client retrieves an item from the server using PIR. If the most updated version of the i-th data item exists in the server, the client retrieves the data item. Otherwise, the client retrieves a random item (data item or dummy item) from the server. Third, the client writes a dummy item to the write cache. Fourth, if the slot mapping table is stored in the server, the client pretends to update the mapping table by writing a dummy entry to the table's cache.

Given the new data value of the i-th data item, a write operation on the i-th data item is simulated by following steps. First, if the slot mapping table is stored in the server, the client locates the i-th data item's slot by obviously looking up the slot mapping table. Second, the client retrieves an random item (data item or dummy item) from the server using PIR. Third, the client writes the i-th data item with its new data value to the write cache. Fourth, the client updates the i-th data item's slot location in the mapping table obliviously.

Based on the analysis in Section IV-F, we can learn that the amortized communication cost of an oblivious read/write is: $O(l+\sqrt{N}\times\log N)$ when client-side storage is $O(N^{1/r}\times l)$; $O(\log N\times(l+\sqrt{N}))$ when client-side storage is $O(l+N^{1/r}\times\log N)$; $O(\sqrt{N}\times(\log N)^2)$ when client-side storage is $O(l)$ and $O(l)<O(N^{1/r}\times\log N)$. $r\geq 2$ and r is a small constant here. The amortized computational cost of an oblivious read/write is $O(N\times l)$. The server-side storage usage is $O(N\times l)$.

If there is only one ORAM client and the client has enough local storage for the slot mapping table, the table can be stored in the client side as well. In that case, the amortized communication cost can be reduced to: $O(l)$ when client-side storage is $O(N^{1/r}\times l+N\times\log N)$; $O(\log N\times l)$ when client-side storage is $O(l+N\times\log N)$.

B. Full Functional ORAM directly based on Advanced Single-server Write-only ORAM and PIR

We can build a full functional ORAM based on the advanced write-only ORAM and PIR directly. In this full functional ORAM, an ORAM client is both a write-only ORAM client and a PIR client.

A read operation on the i-th data item is simulated by following steps. First, the client locates the i-th data item's slot by obviously looking up the slot mapping table. Second, the client retrieves an item from the server using PIR. If the most updated version of the i-th data item exists in the server, the client retrieves it using PIR. Otherwise, the client retrieves a random item (data item or dummy item) from the server using PIR. Third, the client randomly chooses a bucket, and writes a dummy item to the bucket's write cache. Fourth, if the slot mapping table is stored in the server, the client pretends to update the slot mapping table by writing a dummy entry to the table's cache.

Given the new data value of the i-th data item, a write operation on the i-th data item is simulated by following steps. First, if the slot mapping table is stored in the server, the client locates the i-th data item's location by obliviously looking up the slot mapping table. Second, the client retrieves an random item (data item or dummy item) from the server using PIR. The first step and second step are performed to hide the type of the simulated operation. Third, the client randomly chooses a bucket, and writes the i-th data item with its new data value to the bucket's write cache. Fourth, the client updates the i-th data item's bucket number and slot location in the bucket mapping table and the slot mapping table obliviously.

The amortized computational and communication costs of an oblivious read/write are $O(N\times l)$ and $O(l+\sqrt{N}\times\log N)$ respectively. The server-side and client-side storage usages are $O(N\times l)$ and $O(N\times\log N+l)$ respectively.

If there is only one ORAM client, the bucket mapping table and slot mapping table can be stored in the client side as well. In that case, the amortized communication cost of an oblivious read/write is reduced to $O(l)$. The computational cost, server-side and client-side storage usages are still the same.