Summer 8-8-2015

GP-ORAM: A Generalized Partition ORAM

Zhang Jinsheng
Department of Computer Science, alexzjs@iastate.edu

Zhang Wensheng
Iowa State University

Daji Qiao
Iowa State University

GP-ORAM: A Generalized Partition ORAM

Jinsheng Zhang[1], Wensheng Zhang[1], and Daji Qiao[2]

[1] Department of Computer Science
[2] Department of Electric and Computer Engineering
Iowa State University
Ames, Iowa, 50010, USA
alexzjs, wzhang, daji@iastate.edu

Abstract. Oblivious RAM (ORAM) is a provable technique to protect a user's access pattern to outsourced data. Recently, many ORAM constructions have been proposed, but most of them are impractical due to high communication and user-side storage costs. Motivated by Partition ORAM (P-ORAM) [16], a state-of-the-art communication-efficient ORAM construction, this paper proposes GP-ORAM (Generalized Partition ORAM) as a new framework to assemble multiple ORAM partitions together while overcoming the limitations of the P-ORAM construction. GP-ORAM allows smaller and adjustable number of partitions, fully utilizes the available user-side storage to reduce communication cost, and can efficiently export the index table to the server. As a result, GP-ORAM incurs low bandwidth cost (i.e., $O(\log N)$ data blocks per query in practice) and has significantly less user-side storage cost than P-ORAM. We demonstrate the security and practicality of GP-ORAM through extensive performance analysis.

1 Introduction

Oblivious RAM (ORAM) [4], which was originally proposed by Goldreich and Ostrovsky, has been a provable approach to preserving a user's access pattern to data outsourced to a remote storage server. The past decades have witnessed numerous ORAM constructions [3, 5–9, 12–16, 18, 19] developed for various purposes. Although many neat asymptotical results have been reported, the practicality of these constructions is still not satisfactory. Particularly, the designs either demand for large user-side storage or incur high communication cost.

Partition ORAM (P-ORAM) [16] is one recent effort in developing practical ORAMs. The P-ORAM construction was designed to achieve a low and thus practically acceptable communication cost. Specifically, the server-side storage of P-ORAM is organized as $\sqrt{N}$ partitions, assuming N is the number of exported data blocks, and each partition is an ORAM. The user-side storage includes an index table recording the location of each block, a shuffling buffer that can store and shuffle all data blocks of any ORAM partition, and $\sqrt{N}$ stash slots. With such a storage arrangement, it has been shown that the communication cost for data query and shuffling is as low as $\log N$ data blocks per query. Compared to other state-of-the-art ORAM constructions [10, 17, 20], P-ORAM achieves higher communication efficiency.

However, P-ORAM design has its limitations. First of all, it requires a large and fixed local storage to store the index table and facilitate shuffling. For example, when

$N = 2^{32}$ and block size is 64 KB, 31 GB local storage is needed. Second, the index table cannot be efficiently exported to the server. According to our evaluation, if the index structure is exported to the server, in order to query just a single block, more than 1000 data blocks on average have to be retrieved. In addition, the user's accesses to data blocks have to be entirely sequential in order to compress the index table.

To address the above limitations of P-ORAM, while inheriting its nice feature of low communication cost, this paper proposes a generalized version of P-ORAM, called GP-ORAM. There are a few key improvements of GP-ORAM over P-ORAM. First, the number of partitions is adjustable in GP-ORAM. This way, even with a smaller local storage than what P-ORAM requires, GP-ORAM may still achieve a low communication overhead via properly adjusting the number of partitions. Second, each ORAM partition in GP-ORAM is redesigned (different from that in P-ORAM) to enable efficient query and shuffling. Finally, the index structure in GP-ORAM is also redesigned to enable efficient exportation of it and accommodate the above changes.

Rigorous security analysis has been conducted to prove that the proposed GP-ORAM construction can preserve a user's access pattern and the construction fails with only a probability of $O(N^{-\log\log N})$. Extensive cost analysis has also been conducted to show that GP-ORAM is a more practical construction than P-ORAM. Particularly, the local storage demanded by the recursive version of our proposed GP-ORAM scheme is only 2.5%$\sim$0.14% of that by the non-recursive version of the P-ORAM scheme (note: as shown in Section 6, the recursive version of the P-ORAM scheme is impractical due to its extremely high communication cost, and therefore is not considered), while GP-ORAM only yields 1 to 3 times higher communication cost than P-ORAM.

In the rest of the paper, Section 2 formalizes the problem. Section 3 describes the intuitions behind the proposed GP-ORAM design, and Section 4 elaborates the details of the GP-ORAM construction. Sections 5 and 6 present the security and cost analyses of GP-ORAM, respectively. Section 7 briefly reviews the existing ORAM constructions. Finally, Section 8 concludes the work.

2 Problem Statement

We consider a system composed of a user and a remote storage server. The user exports a large set of data to the server, and wishes to access these data without exposing the access pattern to the storage server. Data is assumed to be stored and accessed in the unit of block, and typically a block is no less than 64 KB [16]. Let N and B denote the total number of blocks exported and the size of a block (in bits), respectively.

Server and user may have different storage capabilities. The cloud server could hold terabytes to petabytes of data in its storage cluster. The user may use thin devices such as tablets and smartphones, and thus may have only gigabytes of RAM and local storage available. Moreover, in practice, bandwidth is usually more expensive than computation and storage. Thus, we aim to design an ORAM scheme that can utilize the given user-side storage efficiently so that the bandwidth cost can be minimized.

In an ORAM system, each data request from the user, which the user wishes to keep private, can be one of the following types: *(1) read a data block D_i of unique ID i from the storage, denoted as a 3-tuple $(read, i, D_i)$; (2) write/modify a data block D_i*

of unique ID i to the storage, denoted as a 3-tuple $(write, i, D_i)$. To accomplish a data request, the user may need to access the remote storage multiple times. Each access to the remote storage, which is observable by the server, can be one of the following types: *(1) retrieve (read) a data block D_i from a location loc at the remote storage, denoted as $(read, loc, D_i)$; (2) upload (write) a data block D_i to a location loc at the remote storage, denoted as $(write, loc, D_i)$.*

Security Definition We assume that the server is honest but curious. That is, it behaves faithfully according to the ORAM design to store data and serve users' read or write requests, but it may attempt to figure out the user's data access pattern. The network connection between the user and the server is assumed to be secure; in practice, this can be achieved using well-known techniques such as SSL [2].

We inherit the standard security definition of ORAM in [16] to define the security of our proposed ORAM. Intuitively, an ORAM system is considered secure if the server learns nothing about user's access pattern. More precisely, it is defined as follows:

Definition 1. *Let $\boldsymbol{x} = \langle\ (op_1, i_1, D_1),\ (op_2, i_2, D_2),\ \cdots\ \rangle$ denote a private sequence of user's intended data requests, where each op is either a $read$ or $write$ operation, and $A(\boldsymbol{x}) = \langle\ (op'_1, loc_1, D'_1),\ (op'_2, loc_2, D'_2),\ \cdots\ \rangle$ denote the sequence of user's accesses to the remote storage (observable by the server) to accomplish the intended data requests. An ORAM system is said to be secure if (i) for any two equal-length private sequences $\boldsymbol{x}$ and $\boldsymbol{y}$ of intended data requests, their corresponding observable access sequences $A(\boldsymbol{x})$ and $A(\boldsymbol{y})$ are computationally indistinguishable; and (ii) the probability that the ORAM system fails to operate is small, i.e., $O(N^{-\log\log N})$.*

3 Intuition

As GP-ORAM is generalized from P-ORAM, we first review the key ideas and limitations of P-ORAM. As shown in Figure 1, the server-side storage of P-ORAM is

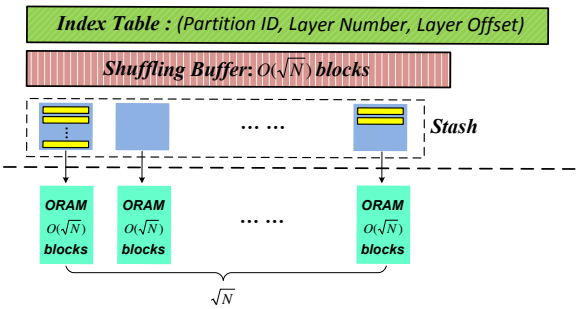


Fig. 1. P-ORAM Storage Organization.

organized as $\sqrt{N}$ ORAM partitions, while the user-side storage includes an index table

recording the location (i.e., partition ID, layer number and layer offset) of each block, a shuffling buffer that can store and shuffle $O(\sqrt{N})$ data blocks and $\sqrt{N}$ stash slots each corresponding to one partition. To query one data block, it needs to retrieve one data block from each layer of an ORAM partition on the server, which results in $O(\log N)$ data blocks of communication cost, and the query target block is relocated to a randomly selected stash slot. Each query is followed by a background eviction, in which some data blocks are evicted from stash slots into their corresponding ORAM partitions; the evictions cause the ORAM partitions to be gradually reshuffled, and shuffling causes $O(\log N)$ data blocks of communication cost per query, on average. To summarize, as bandwidth is usually more expensive than storage, P-ORAM was designed to achieve a low communication overhead at the cost of increased local storage.

However, P-ORAM has the following limitations. First, P-ORAM requires a large local storage ($O(\sqrt{N}B)$ bits), due to $\sqrt{N}$ stash slots and a shuffling buffer with a capacity of $O(\sqrt{N})$ blocks. This limits P-ORAM's practical applicability as it is impossible to implement P-ORAM if the user has less local storage than required. Second, the index table cannot be efficiently outsourced to the server. Each entry of the table has three fields: partition ID, layer number, and layer offset. The layer number and layer offset need to be updated during both query and shuffling processes. If the index table is outsourced to the server, the query and shuffling processes need to frequently query and update the index table, which leads to impractically high communication cost. Third, the user's data accesses have to be entirely sequential in order to compress the index table.

Motivated by P-ORAM and also to overcome its limitations, we present GP-ORAM as a new framework to assemble multiple ORAM partitions together. It has the following key ideas. First, the number of partitions is not fixed so that the user can adjust the number of partitions according to the available local storage. Second, the index table is re-designed so that it can be outsourced to the server efficiently. Third, to make full use the available local storage, each ORAM partition is based on a revised S-ORAM [20] construction. As a result, GP-ORAM inherits the security property and the communication efficiency of P-ORAM while being able to work with and fully utilize a wide range of available local storage.

4 The Proposed GP-ORAM Construction

We elaborate the design of GP-ORAM in terms of storage organization, system initialization, query process, and background eviction process. To simplify the presentation, we assume the user stores index entries of all outsourced data blocks locally. In practice, to save the user's local storage, the index entries can be recursively exported to the storage server, following the same ideas used in tree ORAM [14] and Path-ORAM [17]. Detailed description of the recursive version of the GP-ORAM construction can be found in Appendix I

4.1 Storage Organization

GP-ORAM stores both real blocks (i.e., user's N actual data blocks outsourced to the server) and dummy blocks (i.e., faked data blocks with random padding). When a block

is in plain-text, it can be split into *pieces* and the size of each piece is $b = \log N$ bits. For each real block, the block ID i is contained in its first piece, denoted as $d_{i,1}$, while the first piece of each dummy block is set to -1. The remaining pieces store the content of that block, denoted as $d_{i,2}, d_{i,3}, \cdots, d_{i,\eta-1}$.

Before being exported to the remote storage server, the plain-text block is encrypted using CTR encryption mode (counter encryption mode) [11] piece by piece with a secret key k. Specifically, the ciphertext of each block D_i contains η pieces, denoted as $c_{i,0}, \cdots, c_{i,\eta-1}$, where

$$\begin{aligned} &c_{i,0} = E_k(ctr), \text{ where } ctr \text{ is a nounce generated by a pseudo-random function;} \\ &c_{i,1} = E_k(ctr+1) \oplus d_{i,1}; \\ &\cdots; \\ &c_{i,\eta-1} = E_k(ctr+\eta-1) \oplus d_{i,\eta-1}. \end{aligned} \tag{1}$$

Thus, the encrypted block (denoted as D_i) is $D_i = (c_{i,0}, c_{i,1}, c_{i,2}, \cdots, c_{i,\eta-1})$.

Server Storage The server-side storage is divided into P smaller fully-functional ORAM partitions, where P is a system parameter. Each partition can hold $1.1N/P$ real blocks. As shown in Lemma 1 (Section 5), given that $\log N \log\log N \leq P \leq \sqrt{N}$, the number of real blocks in each partition is upper bounded by $1.1N/P$ with a probability of $1 - O(N^{-\log\log N})$.

In GP-ORAM, each ORAM partition is a revised version of the S-ORAM [20] construction. Specifically, each partition is organized as a pyramidical structure shown in Figure 2, where the total number of layers is denoted as $L_2 = \lceil \log(N/P) \rceil$. The top layer, i.e., layer 1, is an array containing up to four blocks. Each of the rest layers is organized as one or multiple segments. These layers are further divided into single-segment layers (i.e., T1-layers, including layers 2 to $L_1 = \lfloor \log(3\log^2 N) \rfloor - 1$) and multi-segment layers (i.e., T2-layers, including layers $L_1 + 1$ to L_2).

Each T1-layer l has a single segment. The segment stores 2^{l+1} blocks, at most half of which are real blocks, and one encrypted index block I_l with 2^{l+1} entries. Each entry of I_l corresponds to a block in the segment and consists of three fields: *ID of the block*, *location of the block in the segment*, and *access bit* indicating whether the block has been accessed since it was placed to the segment.

For each T2-layer $l < L_2$, it is composed of $W_l = \lceil 2^l / \log^2 N \rceil$ segments, while the bottom layer (i.e., layer L_2) contains $W_{L_2} = \lceil 1.1 * 2^{L_2} / \log^2 N \rceil$ segments. The bottom layer has slightly more segments, because it should be able to accommodate $1.1N/P$ real data blocks. A T2-layer segment has the same format as a T1-layer segment except that it needs to contain exactly $3\log^2 N$ data blocks. Having $3\log^2 N$ data blocks per segment is to ensure the security property of the design and it has been proved in [20].

Inside each segment, there is an index block with at most $3\log^2 N$ entries and each entry contains three fields: *ID of the block* (needing $\log N$ bits), *location of the block in the segment* (needing $\log(3\log^2(1.1N/P))$ bits), and *access bit* (needing 1 bit). Thus, an index block needs at most $3\log^2 N[\log N + \log(3\log^2(1.1N/P)) + 1]$ bits. In practice, with $N \leq 2^{32}$ which is considered large enough to accommodate most practical

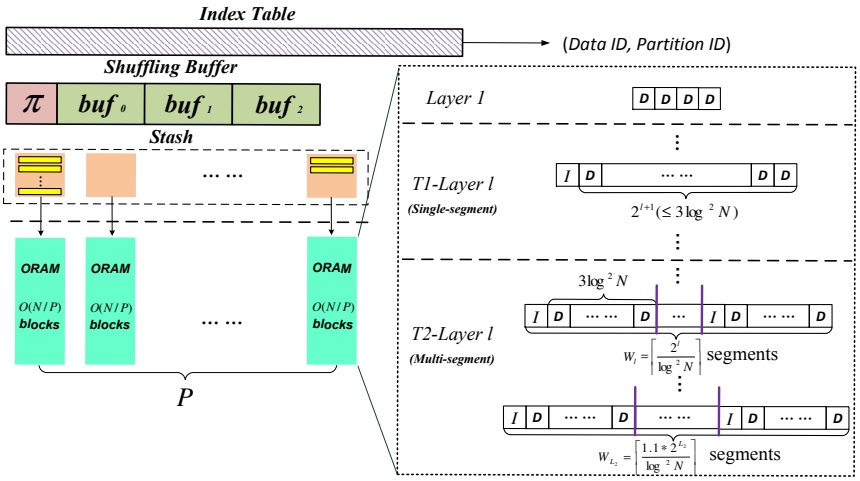


Fig. 2. Organization of the server-side storage.

applications, the size of an index block is less than 32 KB, which can fit into a typical block assumed in P-ORAM [16].

In addition, each ORAM partition p maintains a counter C_p to keep track of the times that the partition has been queried.

User Storage The user-side storage consists of the following components. (i) **Stash with P slots**: each stash slot corresponds to one of the ORAM partitions; that is, it buffers the blocks that should be written to the corresponding partition later. (ii) **Shuffling buffer**: the shuffling buffer (with the capacity of S blocks) is used for data shuffling process. (iii) **Index table**: the index table records the information of each block. Specifically, it has N entries and each entry (p_i, l_i) has two fields; the block is in partition p_i and the block is latest stored on layer l_i. (iv) **Secret storage**: it stores all secrets including cryptographic keys for encryption and authentication, and its size is negligible compared to the other components.

4.2 System Initialization

To initialize, the user first selects a data encryption key, denoted as k. Then, each real block is encrypted and randomly assigned to one of the P partitions; the local index table is initialized to reflect the assignment.

After the above assignment, the user initializes each partition p_i as follows. For each of the real blocks D_j assigned to partition p_i, the user selects a secure hash function, denoted as $H_{p_i,L_2}(*)$, for the bottom layer L_2, and assign D_j to segment $H_{p_i,L_2}(j)$. Then, the user adds dummies to ensure each segment contains exactly $3\log^2 N$ blocks. For each segment, the user randomly permutes all blocks inside it and builds an encrypted index block for it. Finally, the index and data blocks are uploaded to the server.

4.3 Data Query

To query a data block D_t, the user first searches the index table to get partition ID p_t and layer number l_t for D_t. Then, the user searches the stash slot of p_t. If D_t is not found, the user will launch a query for D_t in partition p_t; otherwise, a dummy query to p_t will be launched.

Algorithm 1 $Query(D_t, p_t)$

1: $\mathcal{L} \leftarrow$ the set of non-empty layers of partition p_t
2: Retrieve C_p from partition p_t
3: **if** (D_t is a dummy block) **then**
4: $\mathcal{S} \leftarrow \{seg_l | \forall\, l \in \mathcal{L}, seg_l$ is a randomly-selected segment of layer $l\}$
5: Retrieve the index block of each segment in $\mathcal{S}$
6: From each segment in $\mathcal{S}$, retrieve a dummy block that has not been accessed
7: Update, re-encrypt & upload the retrieved index block
8: **else**
9: Find layer $\hat{l}_t$ where D_t is located; $seg_{\hat{l}_t} \leftarrow H_{p_t,\hat{l}_t}(t)$
10: //Secure hash function $H_{p_t,\hat{l}_t}(t)$ decides which segment of layer $\hat{l}_t$ in partition p_t stores D_t
11: $\mathcal{S} \leftarrow \{seg_l | \forall\, l \in \mathcal{L} \setminus \{\hat{l}_t\}, seg_l$ is a randomly-selected segment of layer $l\}$
12: Retrieve the index blocks of segments in $\mathcal{S} \cup \{seg_{\hat{l}_t}\}$
13: From each segment $s \in \mathcal{S} \cup \{seg_{\hat{l}_t}\}$, retrieve a dummy block that has not been accessed if $s \in \mathcal{S}$, or D_t otherwise
14: Update, re-encrypt & upload the retrieved index block
15: **end if**

The algorithm for querying D_t in partition p_t, i.e., $Query(D_t, p_t)$, is revised from the query algorithm in S-ORAM [20] and formally presented in Algorithm 1. In the algorithm, the layer $\hat{l}_t$ where D_t is located is found as follows: First, based on the query counter C_{p_t}, the most recently shuffled layer l' can be inferred. Then, $\hat{l}_t \leftarrow l'$ if $l' \geq l_t$ because D_t must have been shuffled to l' during the most recent shuffling process; otherwise, $\hat{l}_t \leftarrow l_t$.

4.4 Background Eviction

After each data query, a background eviction process as described in Algorithm 2 should be launched to avoid stash overflowing. Similar to P-ORAM, this process could be sequential or random. For simplicity, we adopt the sequential approach. Suppose ψ records the last evicted stash slot and λ denotes the eviction rate (i.e., the number of stash slots that should be evicted after each data query). The eviction operation essentially pushes one data block from its stash slot to layer 1 of its corresponding partition. As the capacity of layer 1 is limited, every four eviction operations performed on a partition could result in layer 1 overflow and thus should trigger a data shuffling of that partition.

Different from P-ORAM, GP-ORAM shuffles data in *pieces* instead of *blocks*, as in S-ORAM [20]. To shuffle a certain x number of blocks in the unit of piece, only

Algorithm 2 Sequential Background Eviction (λ)

for k = 1 to λ **do**
  $\psi \leftarrow (\psi + 1) \mod P$
  if (stash slot[ψ] does not contain real block) **then** write a dummy to layer 1 of p_ψ
  elseremove a real block from stash slot[ψ] and write it to layer 1 of p_ψ
  end if
  $C_{p_\psi} \leftarrow C_{p_\psi} + 1$
  if ($C_{p_\psi} \bmod 4 = 0$) **then**
    Shuffle partition p_ψ
  end if
end for

bx bits of local storage is needed, while Bx bits of local storage would be needed if shuffling these blocks in the unit of block. Hence, GP-ORAM can utilize the shuffling buffer more efficiently than P-ORAM. To facilitate fine-grained shuffling, the shuffling buffer is split into the following two components (as shown in Figure 2): (i) π, which is a buffer to store a *permutation* of up to $2m^2$ inputs and thus needs $2m^2 \log(2m^2)$ bits, where m is a system parameter; (ii) buf_0, which is used to temporarily store up to $2m^2$ data pieces. Recall that each data piece has b bits and the capacity of the shuffling buffer is S bits. In GP-ORAM, we set the shuffling buffer size to

$$S = 4.4 \cdot \frac{N}{P} \cdot (\log(4.4 \cdot \frac{N}{P}) + b). \tag{2}$$

The purpose is to ensure that, for any layer of each partition, each block is downloaded and uploaded for only once during a shuffling process. The shuffling process is the same as in S-ORAM [20], and thus is skipped here due to space limitation.

5 Security Analysis

To show that GP-ORAM is secure according to Definition 1 in Section 2, we develop a proof in two parts: (1) GP-ORAM generates a random access pattern independent of user's actual access pattern, and (2) GP-ORAM fails with only a negligible probability. For the second part, there are three aspects to be proved in detail: (i) the stash overflows with a negligible probability of $O(N^{-\log\log N})$, (ii) any partition overflows with a negligible probability of $O(N^{-\log\log N})$, and (iii) any layer of any partition overflows during data shuffling with a negligible probability of $O(N^{-\log N})$.

Lemma 1. *Given that $P \geq \log N \log\log N$, the total number of real blocks in the stash at any time during data queries is upper bounded by $2P(1 - 2/P)$ with a probability of $1 - O(N^{-\log\log N})$.*

Lemma 2. *Given that $\log N \log\log N \leq P \leq \sqrt{N}$, the total number of real blocks for any partition at any time during data queries is upper bounded by $\Phi = 1.1N/P$ with a probability of $1 - O(N^{-\log\log N})$.*

Theorem 1. *GP-ORAM is secure under the security definition in Section 2.*

Due to space limitation, please refer to Appendices II, III, and IV for the proofs of the above lemmas and theorem.

6 Cost Analysis

In this section, we analyze the costs of non-recursive and recursive GP-ORAM constructions, and compare them to P-ORAM [16], Path-ORAM [17] and S-ORAM [20], which are the most communication-efficient state-of-the-art ORAM constructions.

Cost Analysis for Non-recursive GP-ORAM The communication cost includes query and background eviction costs. Each data query retrieves two blocks (i.e., one index block and one data block) from and uploads only the index block to each non-empty layer of the server. As there are $L_2 = \lceil \log(N/P) \rceil$ layers, query cost on average is: $C_{\text{query}} < 1.5 \cdot \log(\frac{N}{P}) \cdot B$.

As for the background eviction cost, after each query, λ blocks are written to λ consecutive partitions at the server. Thus, P/λ queries result in all P partitions being accessed once. Therefore, for each partition, layer l ($1 < l < L_2$) is involved in a shuffling process every $2 \cdot 2^l \cdot P/\lambda$ queries, while layer L_2 is shuffled every $2^{L_2} \cdot P/\lambda$ queries. Recall that shuffling a T1-layer l involves $2 \cdot 2^l$ blocks, shuffling a T2-layer l involves $4 \cdot 2^l$ blocks, and shuffling layer L_2 involves $5.3 \cdot 2^{L_2}$ blocks. Hence, the amortized shuffling cost is $C_{\text{shuffle}} = (\sum_{l=2}^{L_1} \frac{2 \cdot 2^l \cdot P}{2 \cdot 2^l \cdot P/\lambda} + \sum_{l=L_1+1}^{L_2-1} \frac{4 \cdot 2^l \cdot P}{2 \cdot 2^l \cdot P/\lambda} + \frac{4.4 \cdot 2^{L_2} \cdot P}{2^{L_2} \cdot P/\lambda})B$, Therefore, the communication cost for non-recursive GP-ORAM is $C_{\text{GP-ORAM(NR)}} = C_{\text{query}} + C_{\text{shuffle}} = (1.5 + 2\lambda) \log \frac{N}{P} \cdot B - \lambda(\log\log N - 2.8) \cdot B$.

For storage cost, as stated in Lemmas 1 and 2, the user needs to maintain the following amount of storage space: $2P(1 - \frac{2}{P})B + S + N \cdot (\log N + \log\log \frac{1.1N}{P})$, where $P \geq \log N \log\log N$. The size of the stash is $2P(1 - 2/P)B$, the size of the shuffling buffer is S, and the size of the index table is $N \cdot (\log N + \log\log \frac{1.1N}{P})$, respectively. Note that, the shuffling buffer storage is temporary, while the stash and index table spaces are permanently needed. For server storage, each partition contains at most $5.3N/P$ blocks. Thus, the server storage is less than $5.3NB$.

Cost Analysis for Recursive GP-ORAM Suppose there are ϕ levels of recursion in the recursive construction, and the i^{th} level of recursion is implemented by GP-ORAM$_i$. Thus, GP-ORAM$_1$, which is used to store the user's data blocks, requires a stash of size $2P(1 - 2/P)B$ and a shuffling buffer of size S in the user's local storage, while the index table is exported to the server as GP-ORAM$_2$. The compression rate for GP-ORAM$_2$ can be smaller than 2^{-13} (i.e., the size of GP-ORAM$_2$ can be less than $\frac{1}{2^{13}}$ of that of GP-ORAM$_1$) when $N \leq 2^{44}$ and $B \geq 64\ KB$, which covers the practical scenarios considered in [16]. Therefore, parameter ϕ is no more than 4; that is, no more than 4 levels of recursion are needed in practice.

Since GP-ORAM$_1$ has much larger capacity than other GP-ORAMs, the extra communication cost introduced by recursion can be computed as $O(\sum_{i=1}^{\phi} \log(\alpha^{-i}N) \cdot B)$ in practice. For the extra local storage cost, it mainly comes from the stashes for extra GP-ORAMs (note that the shuffling buffer for GP-ORAM$_1$ can be reused for other smaller GP-ORAMs), and the total size of these stashes is much less than that for GP-ORAM$_1$. Specifically, a stash of size $3P(1 - 2/P)B$ is enough for recursive constructions. At last, the extra cost on server storage is $O(\sum_{i=1}^{\phi} \alpha^{-i}N \cdot B)$.

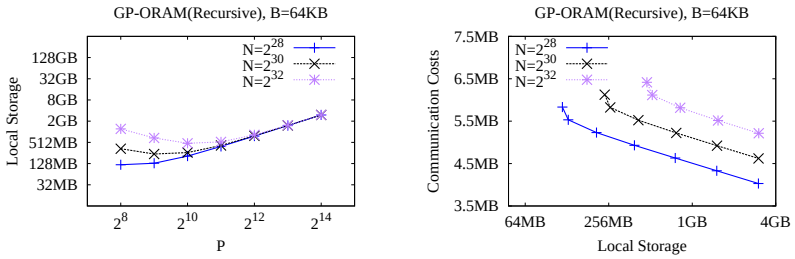
Tradeoff between Local Storage Capacity and Communication Cost in GP-ORAM Suppose a user exports N data blocks each of B bits, and the local storage capacity is $\mathcal{S}_l$. The user could find an optimal P (i.e., number of partitions) for GP-ORAM to minimize the communication cost.

According to $C_{\text{GP-ORAM(NR)}}$ in the non-recursive GP-ORAM cost analysis, the larger is P, the smaller is the communication cost. Hence, the optimal P should be the largest P without incurring a local storage cost higher than $\mathcal{S}_l$. Formally:

$$\begin{aligned}&\text{Maximize } P,\\ &\text{subject to } 2P(1-\frac{2}{P})B + S + \frac{NB}{\alpha} \leq \mathcal{S}_l \text{ for non-recursive GP-ORAM},\\ &\text{subject to } 3P(1-\frac{2}{P})B + S \leq \mathcal{S}_l \text{ for recursive GP-ORAM}.\end{aligned}$$

The example plotted in Figure 3(a) shows the relation between P and local storage consumption in the recursive GP-ORAM. Recall that, the local storage includes shuffling buffer and stash. As we can see from Figure 3(a), when P is small, local storage consumption decreases as P increases; when P becomes large, local storage consumption increases as P increases. This phenomenon can be explained as follows.

- When P is small, the size of each partition is large; hence, the shuffling buffer dominates the local storage. As P increases, shuffling buffer decreases which causes the local storage to decrease as well.
- When P is large, the number of partitions gets large and so the stashes dominates the local storage. As P increases, the size of stashes increases which causes the local storage to increase too.



(a) P vs. Local storage consumption (b) Local storage capacity vs. Minimal communication cost

Fig. 3. Examples illustrating the relation between P, local storage, and minimal communication cost.

Based on the relation plotted in Figure 3(a), the user can find a range of P, with which the required local storage does not exceed $\mathcal{S}_l$. Because the communication cost decreases as P increases, the maximum P within the range becomes the optimal P that minimizes the communication cost. This way, for any given $\mathcal{S}_l$, the communication cost corresponding to the optimal P can be found. Figure 3(b) plots an example to

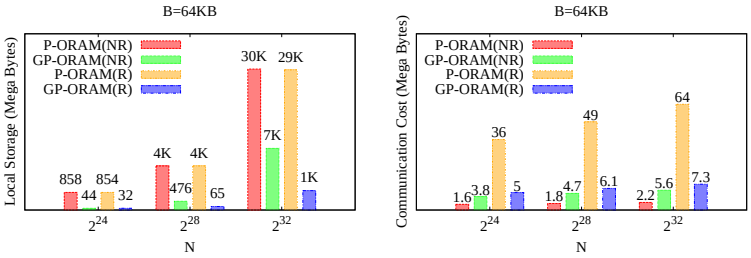
illustrate the relation between local storage capacity and minimal communication cost in the recursive GP-ORAM.

GP-ORAM VS. P-ORAM Table 1 compares GP-ORAM with P-ORAM in terms of asymptotical performance. From the table, we have the following observations: (i) When P is set to N^c ($c < 0.5$) and S is set as in Equation (2), the communication costs for both non-recursive and recursive GP-ORAM can be re-written as $O(\log N \cdot B)$, which is comparable to the cost for non-recursive P-ORAM and much lower than that for recursive P-ORAM. (ii) The local storage costs for non-recursive P-ORAM and GP-ORAM are both $O(NB)$, as the costs are dominated by the index table. The local storage cost for recursive GP-ORAM is $O(PB + S)$, which is asymptotically smaller than $O(\sqrt{N}B)$ as $P < \sqrt{N}$.

Table 1. Asymptotical Performance Comparison.

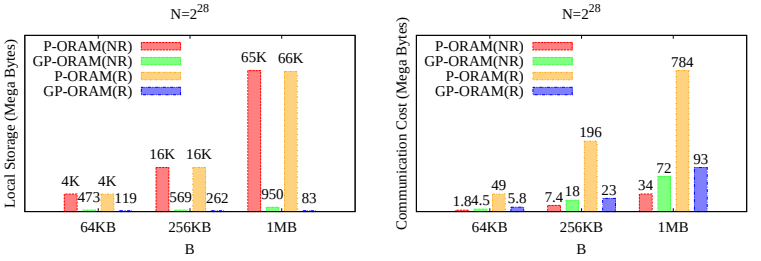
Scheme	Bandwidth Cost	User Storage	Server Storage	Failure Prob.
P-ORAM (NR)	$O(\log N \cdot B)$	$O(NB)$	$< 4NB$	$O(\frac{1}{N^c})$
P-ORAM (R)	$O(\log^2 N \cdot B)$	$O(\sqrt{N}B)$	$< 8NB$	$O(\frac{1}{N^c})$
GP-ORAM (NR)	$O(\frac{\log^3(N/P)}{\log^2 S} \cdot B)$	$O(NB)$	$< 5.3NB$	$O(N^{-\log\log N})$
GP-ORAM (R)	$O(\frac{\log^3(N/P)}{\log^2 S} \cdot B)$	$O(PB + S)$	$< 5.3NB$	$O(N^{-\log\log N})$

Figures 4 and 5 compare the performance of GP-ORAM with P-ORAM under the practical system settings used in [16] (i.e., block size ranging from 64 KB to 1 MB; the number of blocks ranging from 2^{24} to 2^{32}). From the figures, we have the following observations: (i) The local storage demanded by recursive GP-ORAM is only 2.5%$\sim$0.14% of that by non-recursive P-ORAM, while GP-ORAM only yields about 1 to 3 times higher communication cost than P-ORAM. (ii) Recursive P-ORAM is impractical due to its extremely high communication cost.



(i) Local Storage Size under Different N (ii) Bandwidth Cost under Different N

Fig. 4. Comparing local storage and communication cost when $B = 64$ KB.



(i) Local Storage Size under Different Block Size (ii) Bandwidth Cost under Different Block Size

Fig. 5. Comparing local storage and communication cost when $N = 2^{28}$.

Comparing GP-ORAM, Path-ORAM and S-ORAM Table 2 shows the asymptotical performance comparisons between GP-ORAM, Path-ORAM and S-ORAM. Compared to S-ORAM and Path-ORAM, GP-ORAM introduces one adjustable system parameter P, which makes it more tunable.

Table 2. Asymptotical Performance Comparison.

Scheme	Bandwidth Cost	User Storage	Server Storage	Failure Prob.
S-ORAM	$O(\frac{\log^3 N}{\log^2 S} \cdot B)$	$O(S)$	$< 6NB$	$O(N^{-\log N})$
Path-ORAM (NR)	$O(\log N \cdot B)$	$O(NB)$	$10NB$	$N^{-\omega(1)}$
Path-ORAM (R)	$O(\log^2 N \cdot B)$	$O(\log N \cdot B) \cdot \omega(1)$	$> 10NB$	$N^{-\omega(1)}$
GP-ORAM (NR)	$O(\frac{\log^3 (N/P)}{\log^2 S} \cdot B)$	$O(NB)$	$< 5.3NB$	$O(N^{-\log \log N})$
GP-ORAM (R)	$O(\frac{\log^3 (N/P)}{\log^2 S} \cdot B)$	$O(PB + S)$	$< 5.3NB$	$O(N^{-\log \log N})$

The performance comparison between GP-ORAM and Path-ORAM under practical scenarios [16] is shown in Table 3. From the table, it can be seen that GP-ORAM can fully utilize the local storage to achieve better communication efficiency, and it incurs lower server-side storage cost.

Table 3. Practical Performance Comparison.

Scheme	Bandwidth Cost	User Storage	Server Storage
Path-ORAM (NR)	$10 \log N \cdot B$	$N \log N + \log N \cdot B \cdot \omega(1)$	$10NB$
Path-ORAM (R)	$10 \log^2 N \cdot B$	$\log N \cdot B \cdot \omega(1)$	$20NB$
GP-ORAM (NR)	$< 4 \log N \cdot B$	$N \log N + PB + S$	$< 5.3NB$
GP-ORAM (R)	$< 6 \log N \cdot B$	$PB + S$	$< 5.3NB$

Figure 6 shows the performance comparison between GP-ORAM and S-ORAM under practical scenarios [16]. From the figure, we can see that S-ORAM is not fully

tunable as local storage increases. Especially when the local storage is large enough, the communication cost cannot be further reduced. For example, when $N = 2^{32}$, $B = 64$KB and the local storage size has exceeded 1.2 GB, the communication remains the same regardless of the increase in local storage size, while GP-ORAM can achieve 50%-60% savings in communication cost as the local storage gets larger.

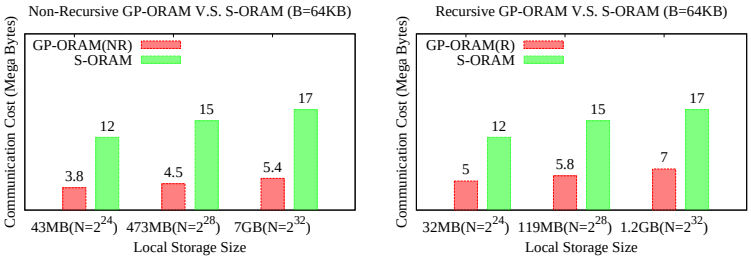


Fig. 6. GP-ORAM vs. S-ORAM with same given local storage.

7 Related Work

According to local storage assumptions, existing ORAM constructions can be roughly classified into the following categories.

ORAMs with $O(1)$ Local Storage [3–10,12,14]. These ORAMs only have little state information, such as secret keys and query counters, stored in local storage. Among them, Balanced ORAM (B-ORAM) [9] proposed by Kushilevitz et. al. incurs the lowest asymptotical communication cost $O(\frac{\log^2 N}{\log \log N})$. In general, these constructions are impractical as the hidden constants behind the big-O notation are quite large due to the heavy data shuffling and background eviction processes. Recently, S-ORAM [20] with constant local storage was proposed to incur $O(\log^2 N)$ communication cost but with practically small constants behind the big-O notation. It leverages the fact that block size is usually large and introduced segmentation-based design of query and shuffling. However, the local storage was not fully utilized as in GP-ORAM.

ORAMs with $O(\log^c N)$ Local Storage [5, 13, 17, 19]. Among these constructions, Path-ORAM [17] re-designed the tree structure ORAM [14] and reduced the bucket size by adding an additional stash to local storage, which resulted in only $O(\log^2 N)$ communication cost. PrivateFS [19] modified and improved a Bloom filter based ORAM solution [18] to approach practicality and concurrency, and resulted in $O(\log^2 N \log \log N)$ communication cost. These ORAMs still have high communication costs, needing to retrieve more than 1000 data blocks per query.

ORAMs with $O(N^c)$ Local Storage [4,5,16–18]. The first ORAM with square-root local storage appeared in [4]. Though the actual communication cost is higher than $\sqrt{N}$ data blocks per query, it is still an inspiring solution that opens the door for subsequent research. Since then, a novel Bloom filter ORAM [18] was proposed which integrates

a more efficient shuffling method to achieve better performance. ORAMs with $O(N^c)$ ($c > 0$) local storage were also studied in [5]. Recently, P-ORAM, with sublinear local storage [16] (square-root local storage in practice) and efficient implementation [17], has achieved much lower communication cost of $O(\log N)$. However, as discussed in Section 6, the user-side storage cost could be too high to be acceptable, especially when the number of outsourced data blocks is large.

8 Conclusion

This paper proposed a new ORAM construction, called Generalized Partition ORAM (GP-ORAM). GP-ORAM utilizes a new shuffling method, adjusts the number of partitions according to the available user-side local storage, and outsources the index table to the server. Through these techniques, it achieves low bandwidth cost ($O(\log N)$) and has significantly less user-side storage cost than P-ORAM. We demonstrate the effectiveness of GP-ORAM via extensive security and cost analysis.

References

1. D. Dubhashi and D. Ranjan. Balls and bins: a study in negative dependence. *Random Structures and Algorithms*, 13, 1996.
2. A. O. Freier, P. Karlton, and P. C. Kocher. The secure sockets layer (SSL) protocol version 3.0. In *RFC 6101*, 2011.
3. C. Gentry, K. Goldman, S. Halevi, C. Julta, M. Raykova, and D. Wichs. Optimizing ORAM and using it efficiently for secure computation. In *Proc. PETS*, 2013.
4. O. Goldreich and R. Ostrovsky. Software protection and simulation on Oblivious RAM. *Journal of the ACM*, 43(3), May 1996.
5. M. T. Goodrich and M. Mitzenmacher. Mapreduce parallel Cuckoo hashing and Oblivious RAM simulations. In *Proc. CoRR*, 2010.
6. M. T. Goodrich and M. Mitzenmacher. Privacy-preserving access of outsourced data via Oblivious RAM simulation. In *Proc. ICALP*, 2011.
7. M. T. Goodrich, M. Mitzenmacher, O. Ohrimenko, and R. Tamassia. Oblivious RAM simulation with efficient worst-case access overhead. In *Proc. CCSW*, 2011.
8. M. T. Goodrich, M. Mitzenmacher, O. Ohrimenko, and R. Tamassia. Privacy-preserving group data access via stateless Oblivious RAM simulation. In *Proc. SODA*, 2012.
9. E. Kushilevitz, S. Lu, and R. Ostrovsky. On the (in)security of hash-based Oblivious RAM and a new balancing scheme. In *Proc. SODA*, 2012.
10. T. Mayberry, E.-O. Blass, and A. H. Chan. Efficient private file retrieval by combining ORAM and PIR. In *Proc. NDSS*, 2014.
11. NIST. Block cipher modes. http://csrc.nist.gov/groups/ST/toolkit/BCM/index.html.
12. B. Pinkas and T. Reinman. Oblivious RAM revisited. In *Proc. CRYPTO*, 2010.
13. L. Ren, X. Yu, C. W. Fletcher, M. van Dijk, and S. Devadas. Design space exploration and optimization of path Oblivious RAM in secure processors. In *Proc. ISCA*, 2006.
14. E. Shi, T.-H. H. Chan, E. Stefanov, and M. Li. Oblivious RAM with $O((\log N)^3)$ worst-case cost. In *Proc. ASIACRYPT*, 2011.
15. E. Stefanov and E. Shi. ObliviStore: high performance oblivious cloud storage. In *Proc. S&P*, 2013.
16. E. Stefanov, E. Shi, and D. Song. Towards practical Oblivious RAM. In *Proc. NDSS*, 2011.

17. E. Stefanov, M. van Dijk, E. Shi, C. Fletcher, L. Ren, X. Yu, and S. Devadas. Path ORAM: an extremely simple Oblivious RAM protocol. In *Proc. CCS*, 2013.
18. P. Williams and R. Sion. Building castles out of mud: practical access pattern privacy and correctness on untrusted storage. In *Proc. CCS*, 2008.
19. P. Williams, R. Sion, and A. Tomescu. PrivateFS: a parallel oblivious file system. In *Proc. CCS*, 2012.
20. J. Zhang, W. Zhang, and D. Qiao. S-ORAM: A segmentation-based Oblivious RAM. In *Proc. ASIACCS*, 2014.

Appendix I: Recursive GP-ORAM

In the construction presented in Section 4, the user needs to maintain an index table in local storage. To reduce the cost, we can adopt recursive construction to outsource the index table to the server. Specifically, letting GP-ORAM$_1$ denote the original GP-ORAM used to store data blocks, a new GP-ORAM$_2$ can be introduce to store the index table of GP-ORAM$_2$; furthermore, another GP-ORAM called GP-ORM$_3$ may be introduced to store the index table of GP-ORAM$_2$, and so on and so forth. Suppose one block in GP-ORAM$_{i+1}$ can store up to α index entries of GP-ORAM$_i$ ($1/\alpha$, therefore, is the compression rate, which is the ratio of GP-ORAM$_{i+1}$'s capacity to GP-ORAM$_i$'s capacity). Then, in each block of GP-ORAM$_{i+1}$, $\log(\frac{N}{\alpha^i})$ bits are needed to represent a sequence of α blocks in GP-ORAM$_i$ and $\alpha \cdot \log P_i$ bits are needed to record the partitions which these blocks should be stored to, where P_i is the number of partitions in GP-ORAM$_i$. Therefore, the relation between α, N, B and P_i is as Equation (3):

$$\log(\frac{N}{\alpha^i}) + \alpha \cdot \log P_i \leq B. \tag{3}$$

With the recursive construction, the local storage can be greatly reduced while the extra communication cost is insignificant. This is analyzed in detail in Section 6.

Appendix II: Proof of Lemma 1.

The stash capacity can be computed by summing up the number of blocks in all P slots. Thus, we can focus on the analysis for a single slot. Note that, blocks are loaded to slots in a uniformly random fashion, and are evicted through the background eviction procedure with eviction rate λ (we use $\lambda = 2$ in the following analysis).

For any single slot, a real block enters this slot with probability $p = 1/P$ and leaves with probability $q = 2/P$. Then, the number of real blocks in any slot is a Discrete Time Markov Chain (DTMC) starting with state 0 (no blocks in the slot), each state i (i blocks in the slot) has forward probability $p_f = p(1-q)$ to state $i+1$ and backward probability $p_b = q(1-p)$ to state $i-1$. Since $\rho = p_f/p_b \leq 1/2$. The stationary distribution for each state i is $\pi_i = \rho^i(1-\rho)$ ($i = 0, 1, 2, \cdots$) and the expectation of the stationary distribution is $\rho/(1-\rho)$. Thus, the expected number of real blocks in the entire stash is

$$\chi = P \cdot \frac{\rho}{1-\rho} = P(1 - \frac{2}{P}). \tag{4}$$

Now, let's observe the upper bound on the stash capacity. Suppose Z_i denotes the number of real blocks in slot i ($1 \leq i \leq P$). Then, Z_i's are negatively associated [1] and Z_i's are geometric random variables with parameter ρ as mentioned before. Hence, the upper tail bound for $Z = \sum_{i=1}^{P} Z_i$ [1, 16] is:

$$\mathbf{Pr}[Z \geq \Psi] \leq e^{-\frac{P \cdot (\Psi/\chi - 1)^2}{4}} = N^{-\frac{P}{4 \ln N}} = O(N^{-\log\log N}), \tag{5}$$

where $\Psi = 2\chi$ and $P \geq \log N \log\log N$. Therefore, given $P \geq \log N \log\log N$, the stash size can be bounded by Ψ with overwhelming probability $1 - O(N^{-\log\log N})$.

Appendix III: Proof of Lemma 2.

For every individual partition, we consider the partition together with its corresponding stash slot as a bin. Thus, the partition capacity can be upper bounded by the bin capacity. Note that, since the snapshot of any moment for the whole system can be seen as a particular distribution of randomly throwing N blocks into P bins, the following results can be deduced on bin capacity:

Consider a particular bin, and define $X_1, \cdots, X_N$ as random variables such that

$$X_i = \begin{cases} 1, \text{ the } i^{\text{th}}\text{real block is mapped to the bin,} \\ 0, \text{ otherwise.} \end{cases} \tag{6}$$

Note that, $X_1, \cdots, X_N$ are independent of each other, and hence for each X_i, $\mathbf{Pr}[X_i = 1] = 1/P$. Let $X = \sum_{i=1}^{N} X_i$. The expectation of X is

$$E[X] = E\left[\sum_{i=1}^{N} X_i\right] = \sum_{i=1}^{N} E[X_i] = N \cdot \frac{1}{P} = \frac{N}{P}. \tag{7}$$

According to the multiplicative form of Chernoff bound, for any $\Phi \geq E[X] = N/P$, it holds that

$$\mathbf{Pr}[\text{a specific bin has more than } \Phi \text{ real blocks}] < e^{-\frac{N(\Phi P/N-1)^2}{3P}}. \tag{8}$$

By applying the union bound, we can obtain

$$\mathbf{Pr}[\ \exists \text{ any bin with more than } \Phi \text{ real blocks}] < P \cdot e^{-\frac{N(\Phi P/N-1)^2}{3P}} = O(N^{-\log\log N}), \tag{9}$$

where $\Phi = 1.1N/P$ and $\log N \log\log N \leq P \leq \sqrt{N}$. Note that any partition capacity is upper bounded by the bin capacity, it holds immediately that any partition capacity is also upper bounded by Φ with overwhelming probability $1 - O(N^{-\log\log N})$.

Appendix IV: Proof of Theorem 1.

According to Definition 1, we will first show that, given any two equal-length sequence $\boldsymbol{x}$ and $\boldsymbol{y}$ of private data requests, their corresponding observable access sequences $A(\boldsymbol{x})$ and $A(\boldsymbol{y})$ are computationally indistinguishable.

Note that, for the k^{th} access $x_k = (op_k, i_k, D_k)$, the observable sequence $A(x_k)$ consists of two parts: $(read, p, \boldsymbol{D})$ which is data query; $(write, p', \boldsymbol{D})$ which is background eviction.

- First, for data query, x_k (or y_k) introduces a read operation on a random partition p_x (or p_y). Then, the background eviction incurs a sequential of write operations on pre-defined partition p' for both x_k and y_k. Hence, $A(x_k)$ and $A(y_k)$ are computationally indistinguishable with each other, because their first parts follow a uniform random distribution and their second parts are the same to each other.
- Second, accesses to individual ORAM partition are oblivious.

- The read operation to a selected partition accesses locations from each non-empty layer (except layer 1) randomly and non-repeatedly;
- When a data block is evicted to this partition, it is re-encrypted and appended to the first layer of this partition.

Also, we have proved GP-ORAM fails with a probability of $O(N^{-\log\log N})$ based on Lemma 1 and Lemma 2. Therefore, it is proved that GP-ORAM construction is secure.