Path ORAM, Onion ORAM and Garbled RAM

Samee Zahur, Matthew Irvine, Kate Highnam

April 3, 2015

1 Introduction

Encrypting large amounts of data when storing it on an untrusted server may not always be enough to protect privacy. Although the data is encrypted, access patterns can also leak sensitive information. For instance, suppose a client has a database stored encrypted on a server, and is performing keyword searches on it. Islam et. Al. have shown that an inference attack can identify approximately 80% of the queries with minimal background knowledge just from the access patterns [4].

This is the main problem we address in this summary. We will discuss design and application of protocols meant to hide the access pattern for a client to an encrypted database. Consider a naive, but inefficient, approach: the client simply accesses every record unit or block in the entire database on the server each time any one of them needs to be accessed. Since the server sees the client accessing each record every time, it cannot infer which record is actually of interest. This is a very simple construction for what is called an Oblivious RAM, or ORAM. In this construction, we can even hide if the operation is a read or write by having the client overwrite each block with fresh encryptions each time.

Obviously, this is prohibitively expensive. In this summary we discuss more efficient methods of achieving the same effect. In the next section, we formally define the problem we are trying to solve. In section 3, we discuss a construction that achieves polylogarithmic overhead on each access. Later, we talk about Onion ORAM, which is a different construction that reduces the communication complexity (but not computation) by offloading computation to the server using additively homomorphic encryption. Finally in Section 5, we discuss an application of ORAM in another area of cryptography — secure multiparty computation.

2 Definition of ORAM

Let us consider a large payload of data that is stored encrypted on a server. For our purposes, we will assume that the data is divided up into *blocks*, and that there is a total of N blocks stored on the server. Each block has a unique number in the range $[0, N)$. We will also assume that they are only accessed by means of either read or write operations, which operate on a block granularity.

A sequence of such operations, $y = (op_1, \ldots, op_m)$, is the access pattern that we want to hide. An oblivious RAM protocol is one that translates each of these operations into multiple read/write operations in a way that only the client knows, producing a new access pattern ORAM(y). We say that an ORAM construction is secure if two accesses ORAM(y) and ORAM(y') are computationally indistinguishable to anyone but the client.

3 Path ORAM

Path ORAM is a type of what is called a tree-based ORAM construction, first developed by Shi et al. [8] for clients with small memory capacities. The general layout of data in both the client and the server is shown in Figure 1. The paper uses a standard notation in tree-structured ORAM, such as a block being a set of B bits and a "bucket" storing some fixed amount Z of blocks within a data structure on the server. A summary of this notation with specific definitions may be viewed in Table 1.

For this protocol, the server-side storage is treated as a binary tree, where each node is a "bucket" of Z blocks. Each block has been encrypted to hide them from the server. If a bucket has less than Z blocks, extra dummy blocks are stored to

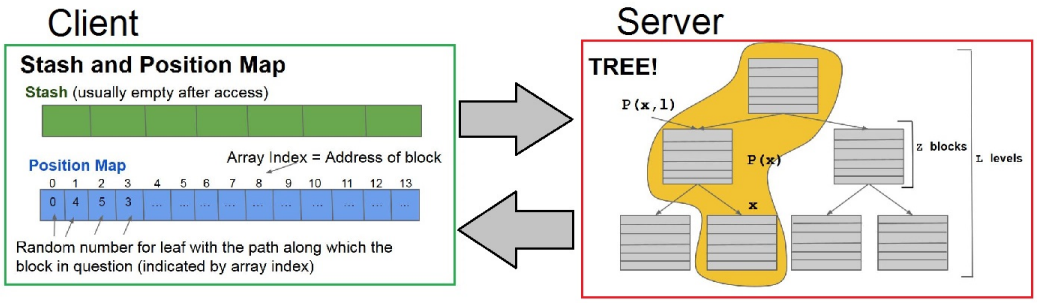

Figure 1: Here is a graphic with a more visual representation of all the notation and data structures involved in the algorithm

N	Total # blocks outsourced to server
L	Height of binary tree
B	Block size (in bits)
Z	Capacity of each bucket (in blocks)
$\mathcal{P}(x)$	path from leaf node x to the root
$\mathcal{P}(x, \ell)$	the bucket at level ℓ along the path $\mathcal{P}(x)$
S	client's local stash
position	client's local position map
$x :=$ position[a]	block a is currently associated with leaf node x, i.e., block a resides somewhere along $\mathcal{P}(x)$ or in the stash.

Table 1: Frequently used terms and notations

pad the bucket. If more than Z blocks are to be stored, they overflow into other buckets or the client's memory.

The client stores two data structures: the stash and the position map. During a read/write access, the stash stores the blocks currently being analyzed by the client. After each access, the stash may still hold blocks which overflowed the root bucket on the server's tree structure (happens rarely). The position map is a table which maps every block address to a leaf node index. If the a-th entry of the position map is x, it indicates that the a-th block is currently assigned to leaf x, and may be found anywhere on $\mathcal{P}(x)$.

Initially, the client stash is empty, the server buckets contain random encryptions of dummy blocks, and the client's position map is filled with independent random numbers corresponding to each leaf. Each block is placed in an appropriate bucket, encrypted, on the server with a write/update command, starting the protocol for Path ORAM.

3.1 Path ORAM Protocol

When the client interacts with the server, there are two possible actions it can take: read or write. In both these cases, the client does not trust the server and needs to ensure our goal of a secure interaction by hiding which block is being accessed.

Generally, this condition is satisfied by reading and decrypting into the stash all values from a path containing the desired information (whether reading or writing, the client needs to fetch it first), accessing the data as needed, and then creating new encryptions for all blocks before writing it all back to the server. Thus when the server sees the information, everything looks changed and some level of security is achieved.

Specifically, the protocol entails the following four steps in the general case of accessing block a:

1. Store the current position of block a in variable x before remapping this block to a new random position. Remapping just means the position map is updated.
2. Read all blocks along $\mathcal{P}(x)$ into the client's stash. Each block is decrypted along the way.
3. At this point, if the original instruction was to write in a new block, update the data stored within the clients stash
4. Write the blocks from the path back into the tree in the server from leaf to root with a fresh encryption scheme so that all blocks appear new to the server. While filling in each bucket, if a block a' is stored along the path, then push (aka. *evict*) it down to the lowest level bucket such that the bucket is along $\mathcal{P}(x)$ and $\mathcal{P}(position[a'])$. Any time a bucket is full, its parent is used. If the root bucket is full, some blocks remain in the client stash.

3.1.1 Computation

Overall, the run time is $O(\log(N))$ per access. In practice most of the time is spent encrypting and decrypting the blocks ($O(\log(N))$). The remote server only needs computation for when the client reads and writes requests.

3.1.2 Security Analysis

Throughout this process, the server sees sequences of requests for a block in a certain position. Once the client reveals a desired block through this access, the block is immediately remapped randomly, statistically independent of its previous position. Following Bayes rule, a quick computation shows that the access pattern using this protocol is computationally indistinguishable from a random sequence of bit strings. This follows the previously stated security conditions.

3.1.3 Recursive Version

The initial point of Path ORAM was for a functional ORAM algorithm with small client storage. The previously stated non-recursive function does, however, require relatively large storage for both the stash and the position map. One possible option for a client without much memory is instead of storing the position map on the client it stores it on the server. For this process to maintain security, a smaller ORAM protocol would be appropriate which would call and recurse through the position map for the requested information. This is described in [8] and [9].

4 Onion ORAM

In this section we describe OnionORAM by Devadas et al. [3]. They build upon the insight by [1] that the well-known $\Omega(\log n)$ lower bound on ORAM complexity bounds only computation, not bandwidth. Thus, they use this insight to construct an ORAM scheme with constant bandwidth overhead per access, using additively homomorphic encryption (in particular, the Damgard-Jurik cryptosystem [2]) to do server-side computation. They talk about why current fully homomorphic encryption schemes would not fit their purpose, citing abysmal performance and higher-than-constant ciphertext expansion as reasons. In this section we describe their algorithm, their protocol for semi-honest security, and the modifications necessary for security against a fully malicious adversary. Finally, we sketch out their performance bound proof.

The authors compare their work with prior work such as Path ORAM, Circuit ORAM [10] etc. Their comparisons show that they achieved much lower bandwidth requirement while having the same, or better, server-side storage and computation requirements.

4.1 Notations

We reuse many of the same notations as in Path ORAM such as B, N, Z, L (Table 1). Some of the other variables are:

- $2^{-\lambda}$: Bucket overflow probability, for some parameter λ. Recommended value is around $\lambda = 80$.
- A: Eviction frequency, where lower value means more frequent. The eviction procedure is invoked once every A ORAM access.

We use $\mathcal{P}(l)$ and $\mathcal{P}(l, i)$ to denote buckets in a tree using the same convention as in the Path ORAM section. We also introduce the notation δ_{ij} to mean a single bit which is 1 if $i = j$, 0 otherwise. For concreteness, the authors assume $B = \Theta(\log^2 N)$ and $A = Z = \Theta(\lambda)$.

4.2 ORAM Operations

Once again, like all other ORAM constructions, we have two basic operations: access (which combines read and write) and eviction. This subsection describes how they are achieved in Onion ORAM.

Just like Path ORAM in the previous section, this is a tree-based ORAM construction. This means it shares a lot of the same principles. We still have each block

mapped to a random leaf node in the tree storage structure, where the mapping is stored on the client side (or recursively, in a smaller ORAM). On each read, the entire path from root to leaf is accessed to select the desired block, after which the accessed node is written back to the root node, freshly encrypted and possibly modified. Each node in the tree stores a bucket of Z blocks. Later, there is an eviction process that happens once per A accesses, where blocks are propagated down a path.

4.2.1 Reducing Bandwidth

The primary bandwidth reduction using additive encryption shows up in how they read or write a block, both during access and eviction operations. Let's say the client wants to access a particular path with blocks B_0 through B_{n-1}, (for $n = ZL$) and the client is really interested in block B_i for some i. In other ORAM structures, the client would have to download them all just to hide B_i is the block he wants. But now, the client can send out encrypted selection bits $E(\delta_{ij})$ for all $j \in [0, n)$, and have the server return $\sum_{j=0}^{n-1} B_j E(\delta_{ij}) = E(B_i)$. Notice that we would normally have encrypted data $E(B_j)$ to start with instead of just B_j, but then we would just end up with $E(E(B_i)) = E^2(B_i)$ instead of $E(B_i)$. In general, each select adds an extra layer of encryption, and then the client can decrypt them all. This layered encryption scheme is the name Onion ORAM comes from.

Something we should note here is that only the actual data payload ever gets homomorphically operated on: the associated encrypted metadata (block number i and its assigned leaf label) is still processed separately by just sending them all to the client as usual. Usually, after each time B_i is read, the metadata for it would be invalidated by setting its address to $\perp$.

When we want to write new data into a bucket (say, writing back to the root node after a read), the client sends out freshly encrypted data as well as various encrypted selection bits. At this point the server can perform selection between the data received and the old block data for each block in the bucket.

4.2.2 Bounding Layers during Eviction

Eviction is the process that prevents the root node from overflowing, where blocks are propagated down the tree towards the leaves. This process is slightly different here compared to Path ORAM. We now run eviction only once every A accesses, as opposed to every access. Moreover, the path selected for eviction is now chosen in a predefined order, independently of the paths with accessed blocks. For eviction, the path to evict is selected in reverse lexicographic order. Meaning, for the i-th eviction, we select the path that goes from root to leaf number bit-reverse(i mod

2^{L-1}).

When a path gets evicted, every block along that path gets written either to the path leaf or one of the sibling nodes, depending on which leaf the path was assigned to. This way, blocks written to the root gets slowly propagated towards the leaves. More importantly, after this is done, the path is known to be completely empty (other than the leaf) to the server as well. At this point, the server can simply reset all the blocks to empty values, resetting the layers of encryption to 0. For the leaf, the client fetches them after eviction and peels of the layers and writes it back to achieve the same effect. This is how the Onion ORAM bounds the maximum layers of encryption. The complete proof is in the paper, but we summarize the key idea here. Because of the deterministic eviction order, there is a maximum number of accesses that can occur before a given block is propagated to the leaf (or accessed and rewritten to the root). This bounds the number of select operations it can be involved in before its layers get reset to 1.

4.3 Security against Malicious Adversary

The authors end the paper with a discussion on using error correction codes to achieve security against a fully malicious adversary. Each block is stored in some error-correcting form such that small, malicious, changes to the block can be reverted. To detect large changes, they keep copies of a few chunks of each block on the client side, which gets compared against the retrieved value on each fetch. If they mismatch, the client aborts. Here, the server does not know which chunks are stored redundantly on the client side. Thus, with all but negligible probability, they ensure that the server cannot change any block sufficiently to corrupt data.

5 How to Garble RAM Programs

This section describes an application of ORAM in the area of secure two-party computation (2PC), as described by Steve Lu and Rafail Ostrovsky [6]. The motivation for 2PC is as follows: let's say two parties have some private information, x and y, that they want to use in some joint computation, and obtain just the output $f(x, y)$. 2PC is a protocol that allows them to perform this without any trusted third party. One of the leading constant-round protocols for performing 2PC is Yao's protocol of *garbled circuits* [11, 5]. Lu and Ostrovsky's paper talks about a way to combine ORAM with garbled circuits to achieve better asymptotic performance while keeping the resulting protocol constant-round. Previous efforts [7] for combining these two techniques always resulted in increased rounds of interactivity.

5.1 Overview of Yao's Garbled Circuits

Consider 2PC as described above with two parties *Gen* and *Eval*, each with their own private inputs. The central concept in Yao's garbled circuits protocol is what is called a *garbling scheme*, and has three components: G, GI and GE. For any given circuit C and input x, the generating party *Gen* can perform $G(C) \to \Gamma$ and $GI(x) \to X$. Γ and X are called the *garbled circuit* and *garbled input* respectively. The evaluating party *Eval* can then receive Γ and X from *Gen* to evaluate the circuit as $GE(\Gamma, X) \to C(x)$ to obtain the output, without knowing anything about the inputs.

5.1.1 Construction of a Garbled Circuit

Here we describe one possible way for G to construct the garbled circuit Γ from C. To construct a garbled circuit, G creates a new garbled truth table for each gate, replacing plaintext inputs with encryption keys that map to encrypted outputs. As an example consider Table 2 below, which shows a garbling for a single gate that computes $z = OR(x, y)$. In this table, each wire x, y, z has two associated garbled keys K_w^0 and K_w^1, for plaintext 0 and 1 respectively. These keys are used as encryption keys for the output of subsequent gates.

Input	**Input**	**Output**	**Encrypted Output**
K_x^0	K_y^0	K_z^0	$Enc_{K_x^0}(Enc_{K_y^0}(K_z^0))$
K_x^0	K_y^1	K_z^1	$Enc_{K_x^0}(Enc_{K_y^1}(K_z^1))$
K_x^1	K_y^0	K_z^1	$Enc_{K_x^1}(Enc_{K_y^0}(K_z^1))$
K_x^1	K_y^1	K_z^1	$Enc_{K_x^1}(Enc_{K_y^1}(K_z^1))$

Table 2: Truth Table for Garbled Circuit $z = OR(x, y)$

These tables of encrypted outputs form Γ, while the appropriate input keys for a given value of x forms the garbled input X. Given these garbled inputs and tables for each gate in the circuit, *Eval* can now evaluate the entire circuit gate by gate to obtain the output $C(x)$, and nothing else.

5.2 Motivating the Protocol Combination

Yao's garbled circuits operates by accepting a circuit C for the function f being evaluated. However, most conventional programs are written for a RAM model, that uses arrays and pointers. While all RAM programs can be converted into a circuit, the overhead is rather large. The paper claims that an $O(t)$ algorithm can become $O(t^3 \log t)$ in the worst case, when converted into circuits. This can be large

in practice, although polynomial. Combining ORAM into the mix allows us to avoid this blowup, and keep the overall 2PC protocol $O(t)$.

5.2.1 Yao's Garbled Circuits and Garbled RAM

Since we are attempting to achieve similar goals to Yao's garbled circuits, most notably non-interactivity, the high level overviews of the schemes have some parallels. Each scheme is an algorithm that produces three programs G, GI, and GE as described in Table 3. In this table C is a circuit, Γ is a garbled circuit, π_t is a program that runs in time t, Π_t is the output garbled RAM program, x is the plaintext input and X is the garbled input.

Garbled Circuits	**Garbled RAM**	**Purpose**
$G(C) \rightarrow (\Gamma)$	$G(\pi_t) \rightarrow \Pi_t$	Garble circuit C or program π_t
$GI(x) \rightarrow X$	$GI(x) \rightarrow X$	Garble input x
$GE(\Gamma, X) \rightarrow C(x)$	$GE(\Pi_t, X) \rightarrow \pi_t(x)$	Evaluate garbled circuit/program on X

Table 3: Components of a garbling scheme, for circuits and RAM

5.3 Constructing Garbled RAM

As indicated earlier, combining ORAM with garbled circuits allows us to achieve a lower bounded runtime than when we do 2PC by garbling the program directly. The main idea is to create a virtual machine and a CPU that executes instructions for this virtual machine. This CPU will be emulated as a circuit. The CPU can request RAM read/writes, so we create an ORAM client circuit that executes these read/write operations. During evaluation we alternate between execution of the garbled CPU circuit and the garbled ORAM client circuit. By garbling the ORAM client we ensure that neither party can gain knowledge of the other party's inputs as part of the RAM read/write process. Going forward we refer to the ORAM client circuit as C_{ORAM} and the CPU circuit as C_{CPU}.

5.3.1 Algorithm G for Garbling a Program

Given a program running in some fixed number of steps, G generates garbled circuits $GC(C_{ORAM})$ and $GC(C_{CPU})$ for each step. The inputs and outputs of these circuits are described in more detail in subsequent secions, but G must ensure that their inputs and outputs use consistent garbled key encodings when these inputs and outputs must be chained together.

5.3.2 Algorithm GI for Garbling Input

The algorithm for GI is straightforward. Since the encodings were generated previously, simply map the input to the relevant time-labeled encodings.

5.3.3 Algorithm GE for Evaluation

Given a garbled program Π_t that originally ran in t steps and a garbled input X, store the beginning program state Σ_1, inputs, and first read/write query V_1 in memory. Then for each time step $i = 1...t$ evaluate the corresponding garbled circuit $GC(C_{ORAM})$ on V_i to obtain a circuit GC_{ORAM} that, when evaluated, executes the actual ORAM read/write query. Execute this query to obtain a garbled output X_i and store it locally. Finally, evaluate $GC(C_{CPU})$ on input X_i and state Σ_i to get the new read/write query V_{i+1} and the new state Σ_{i+1} for the next time step. After the last step, output the final garbled output X_{t+1}.

5.3.4 Constructing C_{CPU}

The circuit C_{CPU} takes the current CPU state Σ and the last memory contents read X as inputs. When executed, this circuit outputs the new CPU state Σ' and the next read/write query V'. It is implemented by creating a circuit that computes the result of all possible CPU operations and then uses a multiplexer to select the desired one.

5.3.5 Constructing C_{ORAM}

The circuit C_{ORAM} takes an ORAM read/write query V as input and outputs a garbled circuit GC_{ORAM} that is used to execute the ORAM query. The GC_{ORAM} output encodings for a given time step are constructed to match the input encodings for the garbled circuit C_{CPU} at that time step.

5.4 Results

Through the construction presented here, oblivious RAM is used in a different way than in the Path ORAM and Onion ORAM papers. The model of ORAM presented in those papers involves a separate client and server where the client makes a single request to the server for each read/write operation as shown in Figure 1 of the Path ORAM section. Because such a request must be made for each RAM read/write operation, algorithms making use of ORAM generally require interactivity. In this case we have two parties, one of which is the generator and one of which is the evaluator. The generator creates garbled circuits and sends them to the evaluator for execution. When executing these circuits the evaluating party acts as both the

ORAM client and server, and the construction ensures that this party is not able to discover information about the other party's input. This is because the ORAM client has already been garbled. Since requests are not being sent between a distinct client and server, no interactivity is required for this protocol.

References

[1] D. Apon, J. Katz, E. Shi, and A. Thiruvengadam. Verifiable Oblivious Storage. In *Public-Key Cryptography–PKC 2014*, pages 131–148. Springer, 2014.

[2] Ivan Damgård and Mads Jurik. A Generalisation, a Simplification and Some Applications of Paillier's Probabilistic Public-Key system. In *Public Key Cryptography*, pages 119–136. Springer, 2001.

[3] S. Devadas, M. van Dijk, C. W. Fletcher, L. Ren, E. Shi, and D. Wichs. Onion ORAM: A Constant Bandwidth ORAM using Additively Homomorphic Encryption. Cryptology ePrint Archive, Report 2015/005, 2015. `http://eprint.iacr.org/`.

[4] M. Islam, M. Kuzu, and M. Kantarcioglu. Access pattern disclosure on searchable encryption: Ramification, attack and mitigation, 2012.

[5] Yehuda Lindell and Benny Pinkas. A Proof of Security of Yaos Protocol for Two-Party Computation. *Journal of Cryptology*, 22(2):161–188, 2009.

[6] S. Lu and R. Ostrovsky. How to Garble RAM Programs. Cryptology ePrint Archive, Report 2012/601, 2012.

[7] R. Ostrovsky and V. Shoup. Private information storage (extended abstract). In *STOC*, pages 294–303, 1997.

[8] E. Shi, T.-H. H. Chan, E. Stefanov, and M. Li. Oblivious RAM with O((logN)3) worst-case cost, 2011.

[9] E. Stefanov and E. Shi. ObliviStore: High performance oblivious cloud storage, 2013.

[10] Xiao Shaun Wang, TH Hubert Chan, and Elaine Shi. Circuit oram: On tightness of the goldreich-ostrovsky lower bound, 2014.

[11] A. C. Yao. Protocols for Secure Computations. In *Proceedings of the 23th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 160–164, 1982.