Efficient 3-Party Distributed ORAM*

Paul Bunn[1], Jonathan Katz[2], Eyal Kushilevitz[3], and Rafail Ostrovsky[4]

[1] Stealth Software Technologies, Inc.
[2] Department of Computer Science, University of Maryland
[3] Computer Science Department, Technion
[4] Department of Computer Science and Department of Mathematics, University of California Los Angeles

Abstract. *Distributed Oblivious RAM (DORAM)* protocols—in which parties obliviously access a shared location in a shared array—are a fundamental component of secure-computation protocols in the RAM model. We show here an efficient, 3-party DORAM protocol with semi-honest security for a single corrupted party. To the best of our knowledge, ours is the first protocol for this setting that runs in constant rounds, requires sublinear communication and linear work, and makes only black-box use of cryptographic primitives. We believe our protocol is also concretely more efficient than existing solutions.

As a building block of independent interest, we construct a 3-server distributed point function with security against *two* colluding servers that is simpler and has better concrete efficiency than prior work.

1 Introduction

A fundamental problem in the context of privacy-preserving protocols for large data is ensuring efficient oblivious read/write access to memory. Research in this area originated with the classical work on oblivious RAM (ORAM) [10], which can be viewed as allowing a stateful client to store an (encrypted) array on a server, and then obliviously read/write data from/to specific addresses of that array with sublinear client-server communication. Roughly, *obliviousness* here means that for each memory access the server learns nothing about which address is being accessed, the specific data being read or written, and even whether a read or a write is being performed. A long line of work [23, 11, 25, 31, 17, 26, 8, 20, 30, 24, 32, 1] has shown both asymptotic and concrete improvements to ORAM protocols. More recently [19, 1, 5, 27, 16], the idea of ORAM was extended to a *multi-server* setting in which a client stores data on two or more servers and obliviousness must hold with respect to each of them.

In all the aforementioned work, there is a fundamental distinction between the client and the server(s): the client knows the address being accessed and, in the case of writes, the data being written; following a read, the client learns

* This material is supported in part by DARPA through SPAWAR contract N66001-15-C-4065. The views expressed are those of the authors and do not reflect the official policy or position of the DoD or the U.S. Government.

the data that was read. That is, there are no privacy/obliviousness requirements with respect to the client.

One of the primary applications of ORAM protocols is in the realm of secure computation in the random-access machine (RAM) model of computation [22, 12, 8, 18, 30, 2, 21, 6, 29, 7, 13, 32, 5, 15]. Here, parties may store an array in a distributed fashion (such that none of them know its contents), and may need to read from or write to the array during the course of executing some algorithm. Here, memory accesses must be oblivious to *all* the parties; there is, in general, no one party who can act as a "client" and who is allowed to learn information about, e.g., the positions in memory being accessed. There is thus a need for a new primitive, which we refer to as *distributed ORAM* (DORAM), that allows the parties to collectively maintain an array and to perform reads/writes on that array. (We refer to Section 5 for a more formal definition.)

An n-party DORAM protocol can be constructed from any n'-party ORAM scheme ($n' \leq n$) using generic secure computation. The main idea is for n' of the parties to act as the servers in the underlying ORAM scheme; a memory access for an address that is secret-shared among the n parties is carried out by having those parties run a secure-computation protocol to evaluate the client algorithm of the underlying ORAM scheme. This approach (with various optimizations) was followed in some prior work on RAM-model secure computation, and motivated efforts to design ORAM schemes in which the client algorithm can be implemented by a low-complexity circuit [30, 28]. In addition to the constructions of DORAM that are implied by prior work on ORAM, or that are implicit in previous work on RAM-based secure computation, dedicated DORAM schemes have been given in the 2-party [5] and 3-party [6, 14] settings.

1.1 Our Contribution

We show here a novel 3-party DORAM protocol, secure against semi-honest corruption of one of the parties. To the best of our knowledge, it is the first such protocol that simultaneously runs in constant rounds, requires sublinear communication and linear work, and makes only black-box use of cryptographic primitives. (The last property, in particular, rules out constructions that apply generic secure computation to known ORAM schemes.) We believe our protocol is also concretely more efficient than existing solutions.

As a building block of independent interest, we show a new construction of a 3-server distributed point function (see Section 2) that is secure against any *two* colluding servers. Our construction has communication complexity $O(\sqrt{N})$, where N is the size of the domain. This matches the asymptotic communication complexity of the only previous construction [3], but our scheme is both simpler and has better concrete efficiency.

1.2 Outline of the Paper

We describe a construction of a 3-server distributed point function (DPF), with privacy against two semi-honest corruptions, in Section 2. In Section 3 we review known constructions of multi-server schemes for oblivious reading (PIR) or oblivious writing (PIW) based on DPFs. We then show in Section 4 how to combine our 3-server DPF with any 2-server PIR scheme to obtain a 3-server ORAM scheme, secure against semi-honest corruption of one server. Finally, in Section 5 we discuss how to extend our ORAM scheme to obtain a 3-party *distributed* ORAM protocol, secure against one semi-honest corruption. Relevant definitions are given in each of the corresponding sections.

2 A 2-Private, 3-Server Distributed Point Function

Distributed point functions were introduced by Gilboa and Ishai [9], and further generalized and improved by Boyle et al. [3, 4].

2.1 Definitions

Fix some parameters N and B. For $y \in \{1, \ldots, N\}$ and $v \in \{0,1\}^B$, define the *point function* $F_{y,v} : \{1, \ldots, N\} \to \{0,1\}^B$ as follows:

$$F_{y,v}(x) = \begin{cases} v & \text{if } x = y \\ 0^B & \text{otherwise.} \end{cases}$$

A distributed point function provides a way for a client to "secret share" a point function among a set of servers. We define it for the special case of three servers, with privacy against any set of two colluding servers. The definitions can be extended in the natural way for other cases.

Definition 1. *A* 3-server distributed point function *consists of a pair of algorithms* $(\mathsf{Gen}, \mathsf{Eval})$ *with the following functionality:*

- Gen *takes as input the security parameter* 1^κ*, an index* $y \in \{1, \ldots, N\}$*, and a value* $v \in \{0,1\}^B$*. It outputs keys* K^1, K^2, K^3*.*
- Eval *is a deterministic algorithm that takes as input a key* K *and an index* $x \in \{1, \ldots, N\}$*, and outputs a string* $\tilde{v} \in \{0,1\}^B$*.*

Correctness requires that for any κ*, any* $(y, v) \in \{1, \ldots, N\} \times \{0,1\}^B$*, any* K^1, K^2, K^3 *output by* $\mathsf{Gen}(1^\kappa, y, v)$*, and any* $x \in \{1, \ldots, N\}$*, we have*

$$\mathsf{Eval}(K^1, x) \oplus \mathsf{Eval}(K^2, x) \oplus \mathsf{Eval}(K^3, x) = F_{y,v}(x).$$

Definition 2. *A 3-server DPF is* 2-private *if for any* $i_1, i_2 \in \{1, 2, 3\}$ *and any* PPT *adversary* A*, the following is negligible in* κ*:*

$$\left| \Pr\left[\begin{array}{c} (y_0, v_0, y_1, v_1) \leftarrow A(1^\kappa); b \leftarrow \{0,1\}; \\ (K^1, K^2, K^3) \leftarrow \mathsf{Gen}(1^\kappa, y_b, v_b) \end{array} : A(K^{i_1}, K^{i_2}) = b \right] - \frac{1}{2} \right|.$$

2.2 Our Construction

Let $G : \{0,1\}^\kappa \to \left(\{0,1\}^B\right)^{\sqrt{N}}$ be a pseudorandom generator. We now describe our construction of a 2-private, 3-server DPF:

$\underline{\mathsf{Gen}(1^\kappa, y, v)}$: View $y \in \{1, \ldots, N\}$ as a pair (i, j) with $i, j \in \sqrt{N}$. Then:

1. For $k = 1, \ldots, \sqrt{N}$ do:
   (a) Choose seeds $s_k^1, s_k^2, s_k^3, s_k^4 \leftarrow \{0,1\}^\kappa$ and let a_k, b_k, c_k, d_k be a random permutation of $1, 2, 3, 4$.
   (b) If $k \neq i$, then define

$$S_k^1 = \{(a_k, s_k^{a_k}), (b_k, s_k^{b_k})\}, \quad S_k^2 = \{(b_k, s_k^{b_k}), (c_k, s_k^{c_k})\},$$
$$\text{and } S_k^3 = \{(a_k, s_k^{a_k}), (c_k, s_k^{c_k})\}.$$

   Note that in this case, one of the seeds is not used, and the other three seeds are each in exactly two of the above sets.

   If $k = i$, then define

$$S_k^1 = \{(a_k, s_k^{a_k}), (b_k, s_k^{b_k})\}, \quad S_k^2 = \{(a_k, s_k^{a_k}), (c_k, s_k^{c_k})\},$$
$$\text{and } S_k^3 = \{(a_k, s_k^{a_k}), (d_k, s_k^{d_k})\}.$$

   Note that in this case, one seed is in all three of the above sets, and the other three seeds are each in exactly one set.
   We stress that the S_i^j are all *unordered* sets.
2. Choose uniform $V^1, V^2, V^3, V^4 \in \left(\{0,1\}^B\right)^{\sqrt{N}}$ with

$$V^1 \oplus V^2 \oplus V^3 \oplus V^4 = (\underbrace{\overbrace{0^B, \ldots, 0^B, v}^{j}, 0^B, \ldots, 0^B}_{\sqrt{N}}).$$

3. Compute

$$\delta^1 = V^1 \oplus G(s_i^1), \quad \delta^2 = V^2 \oplus G(s_i^2),$$
$$\delta^3 = V^3 \oplus G(s_i^3), \quad \delta^4 = V^4 \oplus G(s_i^4).$$

4. The keys that are output are

$$K^1 = (S_1^1, \ldots, S_{\sqrt{N}}^1, \ \delta^1, \delta^2, \delta^3, \delta^4),$$
$$K^2 = (S_1^2, \ldots, S_{\sqrt{N}}^2, \ \delta^1, \delta^2, \delta^3, \delta^4),$$
$$K^3 = (S_1^3, \ldots, S_{\sqrt{N}}^3, \ \delta^1, \delta^2, \delta^3, \delta^4).$$

The length of each key is $O((\kappa + B) \cdot \sqrt{N})$.

$\underline{\mathsf{Eval}(K, x)}$. View $x \in \{1, \ldots, N\}$ as a pair (i', j') with $i', j' \in \sqrt{N}$. Let

$$K = (S_1, \ldots, S_{\sqrt{N}}, \delta^1, \delta^2, \delta^3, \delta^4),$$

where $S_k = \{(\alpha_k, s_k^{\alpha_k}), (\beta_k, s_k^{\beta_k})\}$ for $k = 1, \ldots, \sqrt{N}$ with $\alpha_k, \beta_k \in \{1, 2, 3, 4\}$. Compute the vector

$$\tilde{V} = G(s_{i'}^{\alpha_{i'}}) \oplus G(s_{i'}^{\beta_{i'}}) \oplus \delta^{\alpha_{i'}} \oplus \delta^{\beta_{i'}} \in \left(\{0,1\}^B\right)^{\sqrt{N}},$$

and output the B-bit string in position j' of $\tilde{V}$.

Correctness. Let $y = (i, j)$ and say K^1, K^2, K^3 are output by $\mathsf{Gen}(1^\kappa, y, v)$. Let $x = (i', j')$ and consider the outputs $\tilde{v}^1 = \mathsf{Eval}(K^1, x)$, $\tilde{v}^2 = \mathsf{Eval}(K^2, x)$, and $\tilde{v}^3 = \mathsf{Eval}(K^3, x)$. Let $\tilde{V}^1, \tilde{V}^2, \tilde{V}^3$ denote the intermediate vectors computed by these three executions of Eval. We consider two cases:

1. Say $i' \neq i$. Then
$$\begin{aligned}&\tilde{V}^1 \oplus \tilde{V}^2 \oplus \tilde{V}^3 \\ &= \left(G(s^a_{i'}) \oplus G(s^b_{i'}) \oplus \delta^a \oplus \delta^b\right) \oplus \left(G(s^b_{i'}) \oplus G(s^c_{i'}) \oplus \delta^b \oplus \delta^c\right) \\ &\quad \oplus \left(G(s^a_{i'}) \oplus G(s^c_{i'}) \oplus \delta^a \oplus \delta^c\right) \;=\; 0^{B\cdot\sqrt{N}}.\end{aligned}$$
Hence $\tilde{v}^1 \oplus \tilde{v}^2 \oplus \tilde{v}^3 = 0^B$ for any j'.
2. Say $i' = i$. Then
$$\begin{aligned}&\tilde{V}^1 \oplus \tilde{V}^2 \oplus \tilde{V}^3 \\ &= \left(G(s^a_i) \oplus G(s^b_i) \oplus \delta^a \oplus \delta^b\right) \oplus \left(G(s^a_i) \oplus G(s^c_i) \oplus \delta^a \oplus \delta^c\right) \\ &\quad \oplus \left(G(s^a_i) \oplus G(s^d_i) \oplus \delta^a \oplus \delta^d\right) \\ &= G(s^a_i) \oplus G(s^b_i) \oplus G(s^c_i) \oplus G(s^d_i) \oplus \delta^a \oplus \delta^b \oplus \delta^c \oplus \delta^d \\ &= V^1 \oplus V^2 \oplus V^3 \oplus V^4 = (\underbrace{\overbrace{0^B, \ldots, 0^B, v}^{j}, 0^B, \ldots, 0^B}_{\sqrt{N}}).\end{aligned}$$
Hence $\tilde{v}^1 \oplus \tilde{v}^2 \oplus \tilde{v}^3$ is equal to 0^B if $j' \neq j$, and is equal to v if $j' = j$.

Theorem 1. *The above scheme is 2-private.*

Proof. By symmetry we may assume without loss of generality that servers 1 and 2 are corrupted. Fix a PPT algorithm A and let Expt_0 denote the experiment as in Definition 2. Let ϵ_0 denote the probability with which A correctly outputs b in that experiment, i.e.,

$$\epsilon_0 = \Pr\left[\begin{array}{c}(y_0, v_0, y_1, v_1) \leftarrow A(1^\kappa); b \leftarrow \{0,1\}; \\ (K^1, K^2, K^3) \leftarrow \mathsf{Gen}(1^\kappa, y_b, v_b)\end{array} : A(K^1, K^2) = b\right].$$

Now consider an experiment Expt_1 in which Gen is modified as follows:

1. Compute S^1_k and S^2_k as before. Note that S^3_k need not be defined, since we only care about the keys K^1, K^2 that are provided to A.
2. As before.
3. Compute $\delta^{a_i}, \delta^{b_i}$, and δ^{c_i} as before, but choose uniform $\delta^{d_i} \in \left(\{0,1\}^B\right)^{\sqrt{N}}$.
4. Keys K^1, K^2 are then computed as before.

Observe that seed $s^{d_i}_i$ is never used. It follows from pseudorandomness of G that the view of A in Expt_1 is computationally indistinguishable from its view in Expt_0; hence if we let ϵ_1 denote the probability that A correctly outputs b in Expt_1 we must have $|\epsilon_1 - \epsilon_0| \leq \mathsf{negl}(\kappa)$.

We next define another experiment Expt_2 in which Gen works as follows:

1. For $k = 1, \ldots, \sqrt{N}$, let a_k, b_k, c_k, d_k be a random permutation of 1, 2, 3, 4. Choose $s_k^{a_k}, s_k^{b_k}, s_k^{c_k} \leftarrow \{0,1\}^\kappa$ and set
$$S_k^1 = \{(a_k, s_k^{a_k}), (b_k, s_i^{b_k})\} \text{ and } S_k^2 = \{(b_k, s_k^{b_k}), (c_k, s_k^{c_k})\}.$$
2. Do not define V^1, V^2, V^3, V^4 at all.
3. Choose $\delta^1, \delta^2, \delta^3, \delta^4 \leftarrow \left(\{0,1\}^B\right)^{\sqrt{N}}$.
4. Keys K^1, K^2 are then computed as before.

The joint distribution of K^1, K^2 above is identical to their joint distribution in Expt_1. Thus, if we let ϵ_2 be the probability that A correctly outputs b in Expt_2, we have $\epsilon_2 = \epsilon_1$.

Finally, observe that in Expt_2 the view of A does not depend on the inputs y, v provided to Gen at all, and so $\epsilon_2 = 1/2$. This completes the proof.

3 Oblivious Reading and Writing

We describe here n-server private information retrieval (PIR) protocols for oblivious reading and private information writing (PIW) protocols for oblivious writing, based on any n-server DPF [9]. In the context, as in the case of ORAM, we have a client interacting with these servers, and their is no obliviousness requirement with respect to the client. If the DPF is t-private, these protocols are t-private as well. (Formal definitions are given by Gilboa and Ishai [9].)

PIR. Let $D \in (\{0,1\}^B)^N$ be an encrypted data array. Let $(\mathsf{Gen}, \mathsf{Eval})$ be an n-server DPF for point functions with 1-bit output. Each of the n servers is given a copy of D. To retrieve the data $D[y]$ stored at address y, the client computes $\mathsf{Gen}(1^\kappa, y, 1)$ to obtain keys $K^1, \ldots, K^n$, and sends K^i to the ith server. The ith server computes $c_x^i = \mathsf{Eval}(K^i, x)$ for $x \in \{1, \ldots, N\}$, and sends
$$r^i = \bigoplus_{x \in \{1,\ldots,N\}} c_x^i \cdot D[x]$$
to the client. Finally, the client computes the result $\bigoplus_{i=1}^n r^i$. Correctness holds since
$$\begin{aligned} \bigoplus_{i=1}^n r^i &= \bigoplus_{x \in \{1,\ldots,N\}} \bigoplus_{i=1}^n c_x^i \cdot D[x] \\ &= \bigoplus_{x \in \{1,\ldots,N\}} F_{y,1}(x) \cdot D[x] \;=\; D[y]. \end{aligned}$$
Privacy follows immediately from privacy of the DPF.

PIW. Let $D \in (\{0,1\}^B)^N$ be a data array. Let $(\mathsf{Gen}, \mathsf{Eval})$ be an n-server DPF for point functions with B-bit output. Now, each of the servers is given an *additive share* D^i of D, where $\bigoplus D^i = D$. When the client wants to write the value v to address y, we require the client to know the current value v_{old} stored at that address. (Here, we simply assume the client knows this value; in applications of PIW we will need to provide a way for the client to learn it.) The client computes $\mathsf{Gen}(1^\kappa, y, v \oplus v_{\mathsf{old}})$ to obtain keys $K^1, \ldots, K^n$, and sends K^i to the ith server. The ith server computes $\mathsf{Eval}(K^i, x)$ for $x = 1, \ldots, N$ to obtain a sequence of B-bit values $\tilde{V}^i = (\tilde{v}^i_1, \ldots, \tilde{v}^i_N)$, and then updates its share D^i to $\tilde{D}^i = D^i \oplus \tilde{V}^i$. Note that if we define $\tilde{D} = \bigoplus \tilde{D}^i$, then $\tilde{D}$ is equal to D everywhere except at address y, where the value at that address has been "shifted" by $v \oplus v_{\mathsf{old}}$ so that the new value stored there is v.

4 3-Server ORAM

In this section we describe a 3-server ORAM scheme secure against a *single* semi-honest server. The scheme can be built from any 2-private, 3-server DPF in conjunction with any 2-server PIR protocol. (As discussed in the previous section, a 2-server PIR protocol can be constructed from any 1-private, 2-server DPF; efficient constructions of the latter are known [9, 4].)

A 4-server ORAM scheme. As a warm-up, we sketch a 4-server ORAM protocol (secure against a single semi-honest server), inspired by ideas of [22], based on 2-server PIR and PIW schemes constructed as in the previous section. Let $D \in (\{0,1\}^B)^N$ be the client's (encrypted) data, and let D^1, D^2 be shares so that $D^1 \oplus D^2 = D$. Servers 1 and 2 store D^1, and servers 3 and 4 store D^2. The client can then obliviously read from and write to D as follows: to read the value at address y, the client runs a 2-server PIR protocol with servers 1 and 2 to obtain $D^1[y]$ and with servers 3 and 4 to obtain $D^2[y]$. It then computes $D[y] = D^1[y] \oplus D^2[y]$.

To write the value v to address y, the client first performs an oblivious read (as above) to learn the value v_{old} currently stored at that address. It then runs a 2-server PIW protocol with servers 1 and 3 to store v at address y in the array shared by those servers. Next, it sends the *same* PIW messages to servers 2 and 4, respectively. (The client does *not* run a fresh invocation of the PIW scheme; rather, it sends server 2 the same message it sent to server 1 and sends server 4 the same message it sent to server 3.) This ensures that (1) servers 1 and 2 hold the same updated data $\tilde{D}^1$; (2) servers 3 and 4 hold the same updated data $\tilde{D}^2$; and (3) the updated array $\tilde{D} = \tilde{D}^1 \oplus \tilde{D}^2$ is identical to the previously stored array except at position y (where the value stored is now v).

A 3-server ORAM scheme. We now show how to adapt the above ideas to the 3-server case, using a 2-server PIR scheme and a 2-private, 3-server DPF. The data D of the client is again viewed as an N-element array of B-bit entries. The invariant of the ORAM scheme is that at all times there will exist three shares D^1, D^2, D^3 with $D^1 \oplus D^2 \oplus D^3 = D$; server 1 will hold D^1, D^2, server 2 will hold D^2, D^3, and server 3 will hold D^3, D^1.

Before describing how read and write are performed, we define two subroutines $\mathsf{GetValue}$ and $\mathsf{ShiftValue}$.

$\mathsf{GetValue}$. To learn the entry at address y, the client uses three independent executions of a 2-server PIR scheme. Specifically, it uses an execution of the PIR protocol with servers 1 and 2 to learn $D^2[y]$; an execution of the PIR protocol with servers 2 and 3 to learn $D^3[y]$; and an execution of the PIR protocol with servers 1 and 3 to learn $D^1[y]$. Finally, it XORs the three values just obtained to obtain $D[y] = D^1[y] \oplus D^2[y] \oplus D^3[y]$.

$\mathsf{ShiftValue}$. Let $(\mathsf{Gen}, \mathsf{Eval})$ be a 2-private, 3-server DPF scheme with B-bit output. This subroutine allows the client to shift the value stored at position y by $\Delta \in \{0,1\}^B$, i.e., to change D to $\tilde{D}$ where $\tilde{D}[x] = D[x]$ for $x \neq y$ and $\tilde{D}[y] = D[y] \oplus \Delta$. To do so, the client computes $K^1, K^2, K^3 \leftarrow \mathsf{Gen}(y, \Delta)$ and sends K^1 to server 1, K^2 to server 2, and K^3 to server 3. Each server s respectively computes $\mathsf{Eval}(K^s, x)$ for $x = 1, \ldots, N$ to obtain a sequence of B-bit values $\tilde{V}^s = (\tilde{v}_1^s, \ldots, \tilde{v}_N^s)$, and then updates its share D^s to $\tilde{D}^s = D^s \oplus \tilde{V}^s$. Note that if $\tilde{D}$ denotes the updated version of the array, then $\tilde{D}^1 \oplus \tilde{D}^2 \oplus \tilde{D}^3 = \tilde{D}$.

After the above, server 1 holds $\tilde{D}^1, D^2$, server 2 holds $\tilde{D}^2, D^3$, and server 3 holds $\tilde{D}^3, D^1$, and so the desired invariant does not hold. To fix this, the client also sends K^1 to server 3, K^2 to server 1, and K^3 to server 2. (We stress that the *same* keys used before are being used here, i.e., the client does not run a fresh execution of the DPF.) This allows each server to update its "other" share and hence restore the invariant.

With these in place, we may now define our read and write protocols.

Read. To read the entry at index y, the client runs $\mathsf{GetValue}(y)$ followed by $\mathsf{ShiftValue}(y, 0^B)$.

Write. To write a value v to index y, the client first runs $\mathsf{GetValue}(y)$ to learn the current value v_{old} stored at index y. It then runs $\mathsf{ShiftValue}(y, v \oplus v_{\mathsf{old}})$.

Correctness of the construction is immediate. Security against a single semi-honest server follows from security of the $\mathsf{GetValue}$ and $\mathsf{ShiftValue}$ subroutines, which in turn follow from security of the primitives used: $\mathsf{GetValue}$ is secure because the PIR scheme hides y from any single corrupted server; $\mathsf{ShiftValue}$ is secure against any single corrupted server—even though that server sees *two* keys from the DPF—by virtue of the fact that the DPF is 2-private.

5 3-party Distributed ORAM

5.1 Definition

In the previous section we considered the client/server setting where a single client outsources its data to three servers, and can perform reads and writes on that data. In that setting, the client knows the index y when reading and knows the index y and value v when writing. Here, in contrast, we consider a setting where three parties P_1, P_2, P_3 distributively implement the client (as well as the servers), and none of them should learn the input(s) or output of read/write requests—in fact, they should not even learn whether a read or a write was performed. Instead, all inputs/outputs are additively shared among the three parties, and should remain hidden from any single (semi-honest) party.

More formally, we may define an ideal, reactive functionality $\mathcal{F}_{\mathsf{mem}}$ corresponding to distributed storage of an array with support for memory accesses. For simplicity we leave initialization implicit, and so assume the functionality always stores an array $D \in (\{0,1\}^B)^N$. The functionality works as follows:

1. On input additive shares of (op, y, v) from the three parties, do:
   (a) If $\mathsf{op} = \mathsf{read}$ then set $o = D[y]$.
   (b) If $\mathsf{op} = \mathsf{write}$ then set $D[y] = v$ and $o = 0^B$.
2. Let o^1, o^2, o^3 be random, additive shares of o. Return o^s to party s.

We then define a *1-private, 3-party distributed ORAM* (DORAM) protocol to be a 3-party protocol that realizes the above ideal functionality in the presence of a single (semi-honest) corrupted party.

5.2 Our Construction

The data is shared as in the 3-server ORAM scheme from the previous section, namely, at all times there are three shares D^1, D^2, D^3 with $D^1 \oplus D^2 \oplus D^3 = D$; party 1 will hold D^1, D^2, party 2 will hold D^2, D^3, and party 3 will hold D^3, D^1.

As in the previous section, we begin by constructing subroutines $\mathsf{GetValue}$ and $\mathsf{ShiftValue}$.

$\mathsf{GetValue}$. Here the parties hold y^1, y^2, y^3, respectively, with $y = y^1 \oplus y^2 \oplus y^3$; after running this protocol the parties should hold additive shares v^1, v^2, v^3 of the value $D[y]$. This is accomplished as follows:

1. P_2 chooses uniform r^2 and sends $y^2 \oplus r^2$ to P_3 and r^2 to P_1. Party P_3 chooses uniform r^3 and sends $y^3 \oplus r^3$ to P_2 and r^3 to P_1. Then P_2 and P_3 each compute $\omega = y^2 \oplus r^2 \oplus y^3 \oplus r^3$, and P_1 computes
$$y^1 \oplus r^2 \oplus r^3 = y \oplus \omega.$$
2. P_1 runs the client algorithm in the 2-server PIR protocol using the "shifted index" $y \oplus \omega$. Parties P_2 and P_3 will play the roles of the servers using the "shifted database" that results by shifting the position of every entry in D^3 by ω. Rather than sending their responses to P_1, however, P_2 and P_3 simply record those values locally. Note that this results in P_2 and P_3 holding additive shares of $D^3[y]$.

Repeating the above with P_2 acting as client (reading from D^1) and P_3 acting as client (reading from D^2)—and then having the parties locally XOR their shares together—results in the three parties holding additive shares of $D[y]$.

$\mathsf{ShiftValue}$. Here we assume the parties have shares i^1, i^2, i^3 and j^1, j^2, j^3 such that, if $i = i^1 \oplus i^2 \oplus i^3$ and $j = j^1 \oplus j^2 \oplus j^3$, the shared index is $y = (i, j)$. The parties also have shares v^1, v^2, v^3 with $v^1 \oplus v^2 \oplus v^3 = v$. At the end of this protocol, the parties should hold shares of the updated data $\tilde{D}$ where all entries are the same as in the original data D except that $\tilde{D}[y] = D[y] \oplus v$.

We show how to implement a distributed version of the Gen algorithm in our 3-server DPF. A distributed version of ShiftValue can then be implemented following the ideas from the previous section.

Intuitively, we define a particular pseudorandom generator G' to use in our DPF construction, namely,

$$G'(s_1, s_2, s_3) = G(s_1) \oplus G(s_2) \oplus G(s_3),$$

where G is a pseudorandom generator. Note that this has the property that the output of G' remains pseudorandom as long as at least one of the seeds is unknown. The high-level idea is that every seed in the original DPF will now be split into three seeds, with each seed known to two parties.

The protocol proceeds as follows:

1. For $k = 1, \ldots, \sqrt{N}$ do:
   (a) P_1 chooses values $s^1_{k,1}, s^2_{k,1}, s^3_{k,1}, s^4_{k,1} \leftarrow \{0,1\}^\kappa$ and shares them with P_3. Similarly, P_2 chooses $s^1_{k,2}, s^2_{k,2}, s^3_{k,2}, s^4_{k,2}$ and shares them with P_1, and P_3 chooses $s^1_{k,3}, s^2_{k,3}, s^3_{k,3}, s^4_{k,3}$ and shares them with P_2.
   (b) The parties run a secure multi-party computation implementing the following functionality:

      Choose a random permutation a, b, c, d of 1, 2, 3, 4.

      If $k \neq i$ then give $\{(a, s^a_{k,1}, s^a_{k,2}, s^a_{k,3}), (b, s^b_{k,1}, s^b_{k,2}, s^b_{k,3})\}$ to P_1; give $\{(b, s^b_{k,1}, s^b_{k,2}, s^b_{k,3}), (c, s^c_{k,1}, s^c_{k,2}, s^c_{k,3})\}$ to P_2; and finally give $\{(a, s^a_{k,1}, s^a_{k,2}, s^a_{k,3}), (c, s^c_{k,1}, s^c_{k,2}, s^c_{k,3})\}$ to P_3.

      If $k = i$ then give $\{(a, s^a_{i,1}, s^a_{i,2}, s^a_{i,3}), (b, s^b_{i,1}, s^b_{i,2}, s^b_{i,3})\}$ to P_1; give $\{(a, s^a_{i,1}, s^a_{i,2}, s^a_{i,3}), (c, s^c_{i,1}, s^c_{i,2}, s^c_{i,3})\}$ to P_2; and give values $\{(a, s^a_{i,1}, s^a_{i,2}, s^a_{i,3}), (d, s^d_{i,1}, s^d_{i,2}, s^d_{i,3})\}$ to P_3.

   Note that the above computation is quite simple (in particular, it can be implemented by an NC^0 circuit), and does no cryptographic computation. In ongoing work, we are designing a dedicated protocol implementing the above without relying on generic secure computation.)
2. For $\ell = 1, \ldots, 4$, party P_1 computes $G^\ell_{k,1} = G(s^\ell_{k,1})$ and $G^\ell_{k,2} = G(s^\ell_{k,2})$; party P_2 computes $G^\ell_{k,2} = G(s^\ell_{k,2})$ and $G^\ell_{k,3} = G(s^\ell_{k,3})$; and party P_3 computes $G^\ell_{k,3} = G(s^\ell_{k,3})$ and $G^\ell_{k,1} = G(s^\ell_{k,1})$.
3. For $k = 1, \ldots, \sqrt{N}$ and $\ell = 1, \ldots, 4$, define $G^\ell[k] = G^\ell_{k,1} \oplus G^\ell_{k,2} \oplus G^\ell_{k,3}$. Note that the parties share G^ℓ in the same manner as required for the GetValue protocol described earlier, and so can distributively compute shares $\hat{\delta}^\ell_1, \hat{\delta}^\ell_2, \hat{\delta}^\ell_3$ (with P_1 holding $\hat{\delta}^\ell_1$, P_2 holding $\hat{\delta}^\ell_2$, and P_3 holding $\hat{\delta}^\ell_3$) such that

$$G^\ell[i] = \hat{\delta}^\ell_1 \oplus \hat{\delta}^\ell_2 \oplus \hat{\delta}^\ell_3.$$

4. The parties run a protocol (see below) to generate shares $\{V^1_s, V^2_s, V^3_s, V^4_s\}$ (where the share with subscript s is held by party P_s) such that, if we define

$V^\ell = V_1^\ell \oplus V_2^\ell \oplus V_3^\ell$ (for $\ell = 1, \ldots, 4$) then

$$V^1 \oplus V^2 \oplus V^3 \oplus V^4 = (\underbrace{\overbrace{0^\ell, \ldots, 0^\ell, v}^{j}, 0^\ell, \ldots, 0^\ell}_{\sqrt{N}}).$$

This is done as follows:

(a) As earlier, the parties exchange values so that P_1 holds $j \oplus \omega$ and P_2 and P_3 hold ω for a uniform shift ω.

(b) P_1 runs the client algorithm for a 2-server DPF with index $j \oplus \omega$ and value v^1. Parties P_2 and P_3 shift the local outputs they get by ω. As a result, P_2 and P_3 now have shares a_2 and a_3 such that

$$a_2 \oplus a_3 = (\underbrace{\overbrace{0^\ell, \ldots, 0^\ell, v^1}^{j}, 0^\ell, \ldots, 0^\ell}_{\sqrt{N}}).$$

(c) Symmetrically, the parties compute shares b_1 and b_2 (held by P_1 and P_2, respectively) such that

$$b_1 \oplus b_2 = (\underbrace{\overbrace{0^\ell, \ldots, 0^\ell, v^3}^{j}, 0^\ell, \ldots, 0^\ell}_{\sqrt{N}}),$$

and shares c_1, c_3 (held by P_1 and P_3, respectively) such that

$$c_1 \oplus c_3 = (\underbrace{\overbrace{0^\ell, \ldots, 0^\ell, v^2}^{j}, 0^\ell, \ldots, 0^\ell}_{\sqrt{N}}).$$

Each party P_s can then locally compute four random shares $V_s^1, V_s^2, V_s^3, V_s^4$ whose XOR is equal to the XOR of the two shares they just learned.

5. Each party P_s locally computes $\delta_p^\ell = V_p^\ell \oplus \hat{\delta}_p^\ell$ for $\ell = 1, \ldots, 4$. The parties then all send their shares δ_p^ℓ to each other, so each party can compute

$$\delta^1 = \delta_1^1 \oplus \delta_2^1 \oplus \delta_3^1, \quad \delta^2 = \delta_1^2 \oplus \delta_2^2 \oplus \delta_3^2,$$
$$\delta^3 = \delta_1^3 \oplus \delta_2^3 \oplus \delta_3^3, \quad \delta^4 = \delta_1^4 \oplus \delta_2^4 \oplus \delta_3^4.$$

Note that after the above, each party P_s has a key K^s corresponding to the output of the Gen algorithm for the 3-server DPF.

Memory access. We can now handle a memory access by suitably modifying the approach from the previous section. The parties begin holding additive shares of a memory-access instruction (op, y, v) and data D, and proceed as follows:

1. The parties run the $\mathsf{GetValue}$ protocol using their shares of y. This results in the parties holding shares v^1, v^2, v^3 such that $v^1 \oplus v^2 \oplus v^3 = D[y] = v_{\mathsf{old}}$.

2. The parties run a secure multi-party computation implementing the following functionality:

   If $\mathsf{op} = \mathsf{read}$ then set $w = 0^B$ and $o = v_{\mathsf{old}}$. Otherwise, set $w = v \oplus v_{\mathsf{old}}$ and $o = 0^B$. Output random additive shares w^1, w^2, w^3 of w and random additive shares o^1, o^2, o^3 of o to the parties.
3. The parties run the $\mathsf{ShiftValue}$ protocol using their shares of y and their shares of w. The parties locally output their shares of o.

A proof of the following is tedious, but straightforward.

Theorem 2. *The above is a 1-private, 3-party DORAM protocol in which each memory access requires constant rounds and $O(\sqrt{N})$ communication.*

References

1. Ittai Abraham, Christopher W. Fletcher, Kartik Nayak, Benny Pinkas, and Ling Ren. Asymptotically tight bounds for composing ORAM with PIR. In *17th Intl. Conference on Theory and Practice of Public Key Cryptography—PKC 2017, Part I*, volume 10174 of *LNCS*, pages 91–120. Springer, 2017.
2. A. Afshar, Z. Hu, P. Mohassel, and M. Rosulek. How to efficiently evaluate RAM programs with malicious security. In *Advances in Cryptology—Eurocrypt 2015, Part I*, volume 9056 of *LNCS*, pages 702–729. Springer, 2015.
3. E. Boyle, N. Gilboa, and Y. Ishai. Function secret sharing. In *Advances in Cryptology—Eurocrypt 2015, Part II*, volume 9057 of *LNCS*, pages 337–367. Springer, 2015.
4. Elette Boyle, Niv Gilboa, and Yuval Ishai. Function secret sharing: Improvements and extensions. In *23rd ACM Conf. on Computer and Communications Security (CCS)*, pages 1292–1303. ACM Press, 2016.
5. Jack Doerner and Abhi Shelat. Scaling ORAM for secure computation. In *24th ACM Conf. on Computer and Communications Security (CCS)*, pages 523–535. ACM Press, 2017.
6. Sky Faber, Stanislaw Jarecki, Sotirios Kentros, and Boyang Wei. Three-party ORAM for secure computation. In *Advances in Cryptology—Asiacrypt 2015, Part I*, volume 9452 of *LNCS*, pages 360–385. Springer, 2015.
7. Sanjam Garg, Divya Gupta, Peihan Miao, and Omkant Pandey. Secure multiparty RAM computation in constant rounds. In *Theory of Cryptography Conference—TCC, Part I*, volume 9985 of *LNCS*, pages 491–520. Springer, 2016.
8. Craig Gentry, Kenny Goldman, Shai Halevi, Charanjit Jutla, Mariana Raykova, and Daniel Wichs. Optimizing ORAM and using it efficiently for secure computation. In *Privacy Enhancing Technologies (PETS)*, volume 7981 of *LNCS*, pages 1–18. Springer, 2013.
9. N. Gilboa and Y. Ishai. Distributed point functions and their applications. In *Advances in Cryptology—Eurocrypt 2014*, volume 8441 of *LNCS*, pages 640–658. Springer, 2014.
10. O. Goldreich and R. Ostrovsky. Software protection and simulation on oblivious RAMs. *Journal of the ACM*, 43(3):431–473, 1996.
11. Michael T. Goodrich and Michael Mitzenmacher. Privacy-preserving access of outsourced data via oblivious RAM simulation. In *38th Intl. Colloquium on Automata, Languages, and Programming (ICALP), Part II*, volume 6756 of *LNCS*, pages 576–587. Springer, 2011.

12. S. Dov Gordon, Jonathan Katz, Vladimir Kolesnikov, Fernando Krell, Tal Malkin, Mariana Raykova, and Yevgeniy Vahlis. Secure two-party computation in sublinear (amortized) time. In *19th ACM Conf. on Computer and Communications Security (CCS)*, pages 513–524. ACM Press, 2012.
13. Carmit Hazay and Avishay Yanai. Constant-round maliciously secure two-party computation in the RAM model. In *Theory of Cryptography Conference—TCC, Part I*, volume 9985 of *LNCS*, pages 521–553. Springer, 2016.
14. S. Jarecki and B. Wei. 3PC ORAM with low latency, low bandwidth, and fast batch retrieval. Available at `https://eprint.iacr.org/2018/347`, 2018.
15. M. Keller and A. Yanai. Efficient maliciously secure multiparty computation for RAM. In *Advances in Cryptology—Eurocrypt 2018, Part III*, volume 10822 of *LNCS*, pages 91–124. Springer, 2018.
16. E. Kushilevitz and T. Mour. Sub-logarithmic distributed oblivious RAM with small block size. Available at `https://arxiv.org/pdf/1802.05145.pdf`, 2018.
17. Eyal Kushilevitz, Steve Lu, and Rafail Ostrovsky. On the (in)security of hash-based oblivious RAM and a new balancing scheme. In *23rd SODA*, pages 143–156. ACM-SIAM, 2012.
18. Chang Liu, Yan Huang, Elaine Shi, Jonathan Katz, and Michael W. Hicks. Automating efficient RAM-model secure computation. In *IEEE Symposium on Security and Privacy*, pages 623–638. IEEE, 2014.
19. Steve Lu and Rafail Ostrovsky. Distributed oblivious RAM for secure two-party computation. In *9th Theory of Cryptography Conference—TCC 2013*, LNCS, pages 377–396. Springer, 2013.
20. T. Mayberry, E.-O. Blass, and A.H. Chan. Efficient private file retrieval by combining ORAM and PIR. In *NDSS 2014*. The Internet Society, 2014.
21. Kartik Nayak, Xiao Shaun Wang, Stratis Ioannidis, Udi Weinsberg, Nina Taft, and Elaine Shi. GraphSC: Parallel secure computation made easy. In *IEEE Symposium on Security and Privacy*, pages 377–394. IEEE, 2015.
22. R. Ostrovsky and V. Shoup. Private information storage. In *29th Annual ACM Symposium on Theory of Computing (STOC)*, pages 294–303. ACM Press, 1997.
23. Benny Pinkas and Tzachy Reinman. Oblivious RAM revisited. In *Advances in Cryptology—Crypto 2010*, volume 6223 of *LNCS*, pages 502–519. Springer, 2010.
24. L. Ren, C. Fletcher, A. Kwon, E. Stefanov, E. Shi, M. van Dijk, and S. Devadas. Constants count: Practical improvements to oblivious RAM. In *USENIX Security Symposium*, pages 415–430. USENIX Association, 2015.
25. Elaine Shi, T.-H. Hubert Chan, Emil Stefanov, and Mingfei Li. Oblivious RAM with $O((\log N)^3)$ worst-case cost. In *Advances in Cryptology—Asiacrypt 2011*, volume 7073 of *LNCS*, pages 197–214. Springer, 2011.
26. Emil Stefanov, Marten van Dijk, Elaine Shi, Christopher W. Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. Path ORAM: An extremely simple oblivious RAM protocol. In *20th ACM Conf. on Computer and Communications Security (CCS)*, pages 299–310. ACM Press, 2013.
27. X. Wang, D. Gordon, and J. Katz. Simple and efficient two-server ORAM. Available at `https://eprint.iacr.org/2018/005`, 2018.
28. Xiao Wang, T.-H. Hubert Chan, and Elaine Shi. Circuit ORAM: On tightness of the Goldreich-Ostrovsky lower bound. In *22nd ACM Conf. on Computer and Communications Security (CCS)*, pages 850–861. ACM Press, 2015.
29. Xiao Shaun Wang, S. Dov Gordon, Allen McIntosh, and Jonathan Katz. Secure computation of MIPS machine code. In *ESORICS 2016, Part II*, volume 9879 of *LNCS*, pages 99–117. Springer, 2016.

30. Xiao Shaun Wang, Yan Huang, T.-H. Hubert Chan, Abhi Shelat, and Elaine Shi. SCORAM: Oblivious RAM for secure computation. In *21st ACM Conf. on Computer and Communications Security (CCS)*, pages 191–202. ACM Press, 2014.
31. Peter Williams and Radu Sion. Single round access privacy on outsourced storage. In *19th ACM Conf. on Computer and Communications Security (CCS)*, pages 293–304. ACM Press, 2012.
32. S. Zahur, X. S. Wang, M. Raykova, A. Gascón, J. Doerner, D. Evans, and J. Katz. Revisiting square-root ORAM: Efficient random access in multi-party computation. In *IEEE Symposium on Security and Privacy*, pages 218–234. IEEE, 2016.