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 [11, 25, 30, 18, 26, 8, 21, 29, 24, 31, 1] has shown both asymptotic and concrete improvements to ORAM protocols. More recently [20, 1, 5, 15, 17], 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 [23, 12, 8, 19, 29, 2, 22, 6, 28, 7, 13, 31, 5, 16]. Here, parties store an array in a distributed fashion (such that none of them know its contents), and need to read from and write to the array during the course of executing some algorithm. In this context, 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'-server 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 ORAM scheme. This approach (with various optimizations) was followed in some prior work on RAM-model secure computation, and motivated efforts to design (single-server) ORAM schemes in which the client algorithm can be implemented by a low-complexity circuit [29, 27]. 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 new 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 re-

view 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 describe 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 $y \in [N] = \{1, \ldots, N\}$ 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 a 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. Choose uniform $I^1, I^2, I^3 \in \{0,1\}^{\sqrt{N}}$ with

$$I^1 \oplus I^2 \oplus I^3 = (\underbrace{0, \ldots, 0, \overbrace{1}^{i}, 0, \ldots, 0}_{\sqrt{N}}).$$

2. For $k = 1, \ldots, \sqrt{N}$ do:
   (a) If $k \neq i$, then choose seeds $a_k, b_k, c_k \leftarrow \{0,1\}^\kappa$ and define

$$S_k^1 = \{a_k, b_k\},$$
$$S_k^2 = \{b_k, c_k\},$$
$$S_k^3 = \{c_k, a_k\}.$$

   Note that each seed is in exactly two of the above sets.

   If $k = i$, then choose seeds $a_k, b_k, c_k, d_k \leftarrow \{0,1\}^\kappa$ and define

$$S_k^1 = \{a_k, d_k\},$$
$$S_k^2 = \{b_k, d_k\},$$
$$S_k^3 = \{c_k, d_k\}.$$

   Note that in this case, there is 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* pairs, e.g., each set is specified by writing its elements in lexicographic order.
3. Let $\mathbf{e}_{j,v} \in (\{0,1\}^B)^{\sqrt{N}}$ denote a "characteristic vector" that is zero everywhere except position j, where it has value v. That is,

$$\mathbf{e}_{j,v} = (\underbrace{0^B, \ldots, \overbrace{0^B}^{j}, v, 0^B, \ldots, 0^B}_{\sqrt{N}}).$$

   Compute the *correction word*

$$C = G(a_i) \oplus G(b_i) \oplus G(c_i) \oplus G(d_i) \oplus \mathbf{e}_{j,v}.$$

   (The four seeds input to G are from the iteration of Step 2 when $k = i$.)
4. The keys that are output are:

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

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

$$K = (S_1, \ldots, S_{\sqrt{N}}, I, C),$$

where $S_k = \{\alpha_k, \beta_k\}$ for $k = 1, \ldots, \sqrt{N}$. Compute the vector

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

where $I_{i'}$ denotes the i'-th bit of I. 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

$$\begin{aligned}
&\tilde{V}^1 \oplus \tilde{V}^2 \oplus \tilde{V}^3 \\
&= \left(G(a_{i'}) \oplus G(b_{i'}) \oplus (I^1_{i'} \cdot C)\right) \oplus \left(G(b_{i'}) \oplus G(c_{i'}) \oplus (I^2_{i'} \cdot C)\right) \oplus \\
&\quad \left(G(c_{i'}) \oplus G(a_{i'}) \oplus (I^3_{i'} \cdot C)\right) \\
&= (I^1_{i'} \oplus I^2_{i'} \oplus I^3_{i'}) \cdot C \;=\; 0^{B \cdot \sqrt{N}},
\end{aligned}$$

   where the final equality is because $I^1_{i'} \oplus I^2_{i'} \oplus I^3_{i'} = 0$ when $i' \neq i$. Hence $\tilde{v}^1 \oplus \tilde{v}^2 \oplus \tilde{v}^3 = 0^B$ for any j'.
2. Say $i' = i$. Then

$$\begin{aligned}
&\tilde{V}^1 \oplus \tilde{V}^2 \oplus \tilde{V}^3 \\
&= \left(G(a_{i'}) \oplus G(d_{i'}) \oplus (I^1_{i'} \cdot C)\right) \oplus \left(G(b_{i'}) \oplus G(d_{i'}) \oplus (I^2_{i'} \cdot C)\right) \oplus \\
&\quad \left(G(c_i) \oplus G(d_i) \oplus (I^3_i \cdot C)\right) \\
&= G(a_i) \oplus G(b_i) \oplus G(c_i) \oplus G(d_i) \oplus \left((I^1_i \oplus I^2_i \oplus I^3_i) \cdot C\right) \\
&= G(a_i) \oplus G(b_i) \oplus G(c_i) \oplus G(d_i) \oplus C \;=\; \mathbf{e}_{j,v}.
\end{aligned}$$

   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.,

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

Now consider an experiment Expt_1 in which Gen is modified as follows, where we let $y_b = (i, j)$.

1. For $k = 1, \ldots, \sqrt{N}$, compute S_k^1 and S_k^2 as before. (Note that S_k^3 need not be defined, since we only care about the keys K^1, K^2 that are provided to A. In particular, we never need to define the value of seed c_i.)
2. Set C to a uniform value in $(\{0,1\}^B)^{\sqrt{N}}$.
3. Set I^1, I^2 to uniform values in $\{0,1\}^{\sqrt{N}}$.
4. Keys K^1, K^2 are then defined as before.

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 $|\epsilon_1 - \epsilon_0| \leq \mathsf{negl}(\kappa)$.

It may be observed that the view of A in Expt_1 is independent of i and j. In particular, the joint distributions of S_i^1, S_i^2 and S_k^1, S_k^2 (for $k \neq i$) are identical (namely, the distribution defined by choosing three uniform seeds $a, b, c \in \{0,1\}^\kappa$ and letting the first set be $\{a, b\}$ and the second set be $\{b, c\}$). Thus, $\epsilon_1 = 1/2$, concluding the proof. □

3 Oblivious Reading and Writing

We describe here known n-server protocols [9] for private information retrieval (PIR) for oblivious reading, and private information writing (PIW) for oblivious writing, based on any n-server DPF. In the context, as in the case of ORAM, we have a client interacting with these servers, and there 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 with domain $[N]$ and range $\{0,1\}$. 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 domain $[N]$ and range $\{0,1\}^B$. 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 [23], 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 reads and writes are performed, we define two subroutines GetValue and ShiftValue.

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 $D[y] = D^1[y] \oplus D^2[y] \oplus D^3[y]$.

ShiftValue. This subroutine allows the client to shift the value 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 $\tilde{D}[y] = D[y] \oplus \Delta$. Let $(\mathsf{Gen}, \mathsf{Eval})$ be a 2-private, 3-server DPF scheme with domain $[N]$ and range $\{0,1\}^B$. 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 $\tilde{V}^s = (\tilde{v}^s_1, \ldots, \tilde{v}^s_N)$, 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 to the same value held by the corresponding other server, 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 GetValue and ShiftValue subroutines, which in turn follow from security of the primitives used: GetValue is secure because the PIR scheme hides y from any single corrupted server; 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 define in Figure 1 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 $D \in (\{0,1\}^B)^N$.) We then define a *1-private, 3-party distributed ORAM* (DORAM) protocol to be a 3-party protocol that realizes this ideal functionality in the presence of a single (semi-honest) corrupted party.

Functionality $\mathcal{F}_{\mathsf{mem}}$

The functionality is assumed to be initialized with an array $D \in (\{0,1\}^B)^N$.

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[a] $o = D[y]$ and $D[y] = v$.
2. Let o^1, o^2, o^3 be random, additive shares of o. Return o^s to party s.

[a] For the purposes of the functionality, o can be an arbitrary value when $\mathsf{op} = \mathsf{write}$. For concreteness, we set $o = D[y]$ in this case so as to slightly optimized our protocol.

Fig. 1. Functionality $\mathcal{F}_{\mathsf{mem}}$ for distributed memory access.

5.2 Our Construction

We give a construction of a 3-party DORAM protocol inspired by the 3-server ORAM scheme described in the previous section. Here, however, we rely on the specific 3-server DPF constructed in Section 2.

We maintain the same invariant as in 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^1, D^3. As in the previous section, we begin by constructing subroutines GetValue and ShiftValue.

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 r^2 to P_1 and $y^2 \oplus r^2$ to P_3. Party P_3 chooses uniform r^3 and sends r^3 to P_1 and $y^3 \oplus r^3$ to P_2. 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 as client (reading from D^1) and again with P_3 as client (reading from D^2)—and then having the parties locally XOR their shares together—results in the three parties holding additive shares $\hat{o}^1, \hat{o}^2, \hat{o}^3$ of $D[y]$. Finally, the parties re-randomize their shares. Namely, each party P_s chooses uniform Δ_s and sends it to P_{s+1}; it then sets its output share equal to $o^s = \hat{o}^s \oplus \Delta_s \oplus \Delta_{s-1}$.

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. Namely, the parties will run a protocol that results in party 1 holding K^1, party 2 holding K^2, and party 3 holding K^3, where K^1, K^2, K^3 are distributed as in an execution of $\mathsf{Gen}(1^\kappa, y, v)$. Given this primitive, a distributed version of ShiftValue can then be implemented following the ideas from the previous section.

We now describe the distributed version of the DPF. For clarity, we describe sub-protocols corresponding to each of the steps of the Gen algorithm from Section 2; these sub-protocols can be parallelized in the obvious way.

1. The sub-protocol for step 1 proceeds as follows:
   (a) P_2 chooses uniform r^2 and sends r^2 to P_1 and $i^2 \oplus r^2$ to P_3. Party P_3 chooses uniform r^3 and sends r^3 to P_1 and $i^3 \oplus r^3$ to P_2. Then P_2 and P_3 each compute $\omega = i^2 \oplus r^2 \oplus i^3 \oplus r^3$, and P_1 computes
   $$i^1 \oplus r^2 \oplus r^3 = i \oplus \omega.$$
   (b) P_1 runs $\mathsf{Gen}(1^\kappa, i \oplus \omega, 1)$, where Gen denotes the key-generation algorithm for a 2-server DPF with domain $[\sqrt{N}]$ and range $\{0,1\}$. This results in keys K^2, K^3 that are sent to P_2 and P_3, respectively.
   (c) For $k = 1, \ldots, \sqrt{N}$, party P_2 sets $\hat{I}^2[k] = \mathsf{Eval}(K^2, k \oplus \omega)$ to obtain a string $\hat{I}^2 \in \{0,1\}^{\sqrt{N}}$. Similarly, P_3 computes $\hat{I}^3 \in \{0,1\}^{\sqrt{N}}$. Finally, P_1 sets $\hat{I}^1 = 0^{\sqrt{N}}$. Note that
   $$\hat{I}^2[k] \oplus \hat{I}^3[k] = \begin{cases} 1 & \text{if } k = i \\ 0 & \text{otherwise} \end{cases}$$
   and so
   $$\hat{I}^1 \oplus \hat{I}^2 \oplus \hat{I}^3 = (\underbrace{0, \ldots, 0, \overbrace{1}^{i}, 0, \ldots, 0}_{\sqrt{N}}).$$

(d) The parties re-randomize their shares. Namely, each party P_s chooses uniform Δ_s and sends it to P_{s+1}; it then sets its output share equal to $I_s = \hat{I}_s \oplus \Delta_s \oplus \Delta_{s-1}$.

2. Here we describe the sub-protocol corresponding to step 2. Let I^1, I^2, I^3 be the respective outputs of the parties after step 1, above, and let $I^s[k]$ denote the kth bit of I^s. For $k = 1, \ldots, \sqrt{N}$ do:
   (a) Party P_s chooses $\alpha_k^s, \beta_k^s, \gamma_k^s, \delta_k^s \leftarrow \{0,1\}^\kappa$. Next, P_1 sends $I^1[k]$, α_k^1, β_k^1, δ_k^1 to P_3. Analogously, P_2 sends $I^2[k]$, β_k^2, γ_k^2, δ_k^2 to P_1, and P_3 sends $I^3[k]$, α_k^3, γ_k^3, δ_k^3 to P_2.
   (b) The parties run the following steps:
      i. P_2 chooses α'_k, β'_k, $\delta'_k \leftarrow \{0,1\}^\kappa$ and $z \leftarrow \{0,1\}$, and sends those values to P_3. Parties P_2 and P_3 then compute:

$$\begin{array}{ll} \underline{P_2} & \\ x_0 := \alpha_k^2 \oplus \alpha'_k & \\ x_1 := \beta_k^2 \oplus \beta'_k & \\ x_2 := \delta_k^2 \oplus \delta'_k & \end{array} \qquad \begin{array}{l} \underline{P_3} \\ y_0 := \alpha_k^1 \oplus \alpha_k^3 \oplus \alpha'_k \\ y_1 := \beta_k^1 \oplus \beta_k^3 \oplus \beta'_k \\ y_2 := \delta_k^1 \oplus \delta_k^3 \oplus \delta'_k \end{array}$$

      ii. P_2 computes two ordered pairs S_0, S_1 as follows:

$$S_{I^2[k] \oplus I^3[k]} = \begin{cases} (x_0, x_1) & \text{if } z = 0 \\ (x_1, x_0) & \text{if } z = 1 \end{cases}$$

$$S_{1 \oplus I^2[k] \oplus I^3[k]} = \begin{cases} (x_0, x_2) & \text{if } z = 0 \\ (x_2, x_0) & \text{if } z = 1. \end{cases}$$

      P_3 computes two ordered pairs T_0, T_1 as follows:

$$T_{I^3[k]} = \begin{cases} (y_0, y_1) & \text{if } z = 0 \\ (y_1, y_0) & \text{if } z = 1 \end{cases}$$

$$T_{1 \oplus I^3[k]} = \begin{cases} (y_0, y_2) & \text{if } z = 0 \\ (y_2, y_0) & \text{if } z = 1. \end{cases}$$

      iii. P_1 runs a 1-out-of-2 oblivious-transfer protocol[1] with P_2, where P_1 uses selection bit $I^1[k]$ and P_2 uses inputs S_0, S_1; let (x, x') be the output of P_1. Similarly, P_1 runs a 1-out-of-2 oblivious-transfer protocol with P_3, where P_1 uses selection bit $I^1[k] \oplus I^2[k]$ and P_3 uses inputs T_0, T_1; let (y, y') be the output of P_1. Finally, P_1 defines $S_k^1 = \{x \oplus y,\ x' \oplus y'\}$.

      Note that if $k \neq i$ then $I^1[k] \oplus I^2[k] \oplus I^3[k] = 0$; in that case, we have $\{x, x'\} = \{x_0, x_1\}$ and $\{y, y'\} = \{y_0, y_1\}$, and so

$$S_k^1 = \{\alpha_k^1 \oplus \alpha_k^2 \oplus \alpha_k^3,\ \beta_k^1 \oplus \beta_k^2 \oplus \beta_k^3\}.$$

[1] In our setting, with three parties and one semi-honest corruption, a simple oblivious-transfer protocol with information-theoretic security can be constructed using standard techniques.

On the other hand, if $k = i$ then $I^1[k] \oplus I^2[k] \oplus I^3[k] = 1$; in that case, $\{x, x'\} = \{x_0, x_2\}$ and $\{y, y'\} = \{y_0, y_2\}$, and so

$$S_k^1 = \{\alpha_k^1 \oplus \alpha_k^2 \oplus \alpha_k^3,\ \delta_k^1 \oplus \delta_k^2 \oplus \delta_k^3\}.$$

(c) The parties run (b) two more times, changing the roles of the parties (and modifying the values used) in the analogous way so that each party P_s ends up learning the appropriate S_k^s.

3. The sub-protocol corresponding to step 3 proceeds as follows:
   (a) The parties run a protocol analogous to that of step 1 to generate uniform $\hat{C}^1, \hat{C}^2, \hat{C}^3 \in (\{0,1\}^B)^{\sqrt{N}}$, held by the respective parties, such that

$$\hat{C}^1 \oplus \hat{C}^2 \oplus \hat{C}^3 = \mathbf{e}_{j,v} = (\underbrace{0^B, \ldots, 0^B, \overbrace{v}^{j}, 0^B, \ldots, 0^B}_{\sqrt{N}}). \tag{1}$$

   In detail:
      i. P_2 chooses uniform r^2 and sends r^2 to P_1 and $j^2 \oplus r^2$ to P_3. Party P_3 chooses uniform r^3 and sends r^3 to P_1 and $j^3 \oplus r^3$ to P_2. Then P_2 and P_3 each compute $\omega = j^2 \oplus r^2 \oplus j^3 \oplus r^3$, and P_1 computes

$$j^1 \oplus r^2 \oplus r^3 = j \oplus \omega.$$

      ii. P_1 runs the $\mathsf{Gen}(1^\kappa, j \oplus \omega, v^1)$, where Gen is the key-generation algorithm for a 2-server DPF with domain $[\sqrt{N}]$ and range $\{0,1\}^B$. (Note the differences from the corresponding part of step 1.) This results in keys K^2, K^3 that are sent to P_2 and P_3, respectively.
      iii. For $k = 1, \ldots, \sqrt{N}$, party P_2 sets $\hat{C}_1^2[k] = \mathsf{Eval}(K^2, k \oplus \omega)$. Similarly, P_3 computes $\hat{C}_1^3 \in \{0,1\}^{\sqrt{N}}$. Note that

$$\hat{C}_1^2[k] \oplus \hat{C}_1^3[k] = \begin{cases} v^1 & \text{if } k = j \\ 0 & \text{otherwise.} \end{cases} \tag{2}$$

   The above steps are carried out twice more in a symmetric fashion, with each of P_2 and P_3 acting as client in a 2-server DPF. This results in P_1 and P_3 holding $\hat{C}_2^1$ and $\hat{C}_2^3$, respectively, satisfying a relation as in (2) but with v^2 in place of v^1, and P_1 and P_2 holding $\hat{C}_3^1$ and $\hat{C}_3^2$, respectively, also satisfying a relation as in (2) but with v^3 in place of v^1. Next, P_1 sets $\hat{C}^1 = \hat{C}_2^1 \oplus \hat{C}_3^1$, with P_2, P_3 acting analogously. Finally, the parties re-randomize their shares (in the same way as in earlier steps). The end result is that the parties hold uniform $\hat{C}^1, \hat{C}^2, \hat{C}^3$ satisfying (1).
   (b) Recall that from step 2, each party P_s holds sets $S_1^s, \ldots, S_{\sqrt{N}}^s$. Each party P_s now computes

$$C^s = \hat{C}^s \oplus \left(\bigoplus_k G(S_k^s) \right), \tag{3}$$

where $G(\{s_1, s_2\}) = G(s_1) \oplus G(s_2)$.
It can be verified that

$$C^1 \oplus C^2 \oplus C^3 = \hat{C}^1 \oplus \hat{C}^2 \oplus \hat{C}^3 \oplus G(a_i) \oplus G(b_i) \oplus G(c_i) \oplus G(d_i),$$

where we define

$$\begin{aligned} a_i &= \alpha_i^1 \oplus \alpha_i^2 \oplus \alpha_i^3 \\ b_i &= \beta_i^1 \oplus \beta_i^2 \oplus \beta_i^3 \\ c_i &= \gamma_i^1 \oplus \gamma_i^2 \oplus \gamma_i^3 \\ d_i &= \delta_i^1 \oplus \delta_i^2 \oplus \delta_i^3. \end{aligned}$$

(c) Each party P_s sends C^s to the other two parties, so they can all compute $C = C^1 \oplus C^2 \oplus C^3$.

Note that after the above, each party P_s has a key K^s corresponding to the output of the Gen algorithm for our 3-server DPF from Section 2.

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 o^1, o^2, o^3 with $o = o^1 \oplus o^2 \oplus o^3 = D[y]$.
2. The parties run a secure multi-party computation implementing the following functionality:
   If $\mathsf{op} = \mathsf{read}$ then set $w = 0^B$; otherwise, set $w = v \oplus o$. Output random additive shares w^1, w^2, w^3 of w.
   This functionality can be realized using a simple protocol with information-theoretic security in our setting. Specifically, let $\mathsf{op} \in \{0, 1\}$ with 0 indicating read. The parties hold additive shares of op, v, and o, and need only to compute a (random) additive sharing of

   $$w = \mathsf{op} \cdot (v \oplus o).$$

   This can be computed via a standard protocol for distributed multiplication.
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.

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.*

Proof. We prove security by showing that our $\mathsf{GetValue}$ protocol securely realizes an appropriately defined functionality for reading a value from the parties' shared data, and that our $\mathsf{ShiftValue}$ protocol securely computes the Gen algorithm of our 3-server DPF. (Our notion of securely realizing a functionality is the standard one from the secure-computation literature, for one semi-honest corruption. In particular, we consider indistinguishability of the joint distribution consisting of

the corrupted party's view and the outputs of the other parties.) Security of the overall DORAM protocol then follows in a straightforward manner.

Security of GetValue**.** In proving security of the GetValue protocol, we consider the functionality $\mathcal{F}_{\mathsf{read}}$ in Figure 2. This functionality is non-reactive, and takes the current array (shared as in our protocol) as input from the parties

Functionality $\mathcal{F}_{\mathsf{read}}$

On input D^1, D^2, y^1 from P_1, along with D^2, D^3, y^2 from P_2 and D^1, D^3, y^3 from P_3, do:

1. Let $D = D^1 \oplus D^2 \oplus D^3$ and $y = y^1 \oplus y^2 \oplus y^3$.
2. Set $o = D[y]$.
3. Let o^1, o^2, o^3 be random, additive shares of o. Return o^s to party s.

Fig. 2. Functionality $\mathcal{F}_{\mathsf{read}}$ for a distributed read access.

Claim. Our GetValue protocol securely realizes $\mathcal{F}_{\mathsf{read}}$ for a single, semi-honest corruption.

Proof. Since the protocol is symmetric, we may without loss of generality assume P_1 is corrupted. We show a simulator that takes as input values D^1, D^2, and y^1 used as input by P_1 along with an output value o^1, and simulates a view of P_1 in an execution of the protocol. The simulator works as follows:

First iteration: Choose uniform r_1^2 and r_1^3, and send these to P_1 on behalf of P_2 and P_3, respectively.

Second iteration, step 1: Choose uniform r_2^1 on behalf of P_1 (i.e., as part of P_1's random tape). Choose uniform r_2^3 and send it to P_1 on behalf of P_3.

Second iteration, step 2: Run the 2-server PIR scheme using address 1 to obtain keys K_2^1, K_2^3. Send K_2^1 to P_1 on behalf of P_2. Let o_2^1 be the local value that P_1 would compute in this step.

Third iteration: The third iteration is simulated analogously to the second iteration. Let o_3^1 be the local value that P_1 would compute in this step.

Re-randomization step: Choose uniform Δ_1 on behalf of P_1, and let Δ_3 be such that

$$o_2^1 \oplus o_3^1 \oplus \Delta_1 \oplus \Delta_3 = o^1.$$

Send Δ_3 to P_1 on behalf of P_3.

Security of the PIR scheme readily implies that the distribution of the simulated view of P_1 in the ideal world is computationally indistinguishable from the distribution of its view in a real execution of the protocol. Moreover, even conditioned on P_1's view, the outputs of P_2 and P_3 are uniform (subject to XORing to the correct output) in both the ideal- and real-world executions. □

Next, we show that the ShiftValue protocol securely computes the Gen algorithm of our 3-server DPF, i.e., that it securely realizes the functionality that takes additive shares of y and v from the three parties, computes $K^1, K^2, K^3 \leftarrow \mathsf{Gen}(1^\kappa, y, v)$, and returns K^s to P_s. For ease of exposition, we show that each step of the ShiftValue protocol securely computes the corresponding step of Gen.

Security of ShiftValue (step 1). The desired functionality here is simple: the parties's inputs are additive shares i^1, i^2, i^3 of a value $i = i^1 \oplus i^2 \oplus i^3$, and the parties' outputs are I^1, I^2, I^3, respectively, that are uniformly distributed subject to:

$$I^1 \oplus I^2 \oplus I^3 = (\underbrace{0, \ldots, 0, \overbrace{1}^{i}, 0, \ldots, 0}_{\sqrt{N}}).$$

Claim. The first step of the GetValue protocol securely realizes the functionality just described for a single, semi-honest corruption.

Proof. We consider separately the case where P_1 is corrupted, and the case where either P_2 or P_3 is corrupted. For corrupted P_1, we define a simulator that takes as input a value i^1 used as input by P_1 along with an output value I^1, and simulates a view of P_1 in an execution of the protocol. The simulator works as follows:

1. Choose uniform r^2 and r^3, and send these to P_1 on behalf of P_2 and P_3, respectively.
2. Choose uniform Δ_1 on behalf of P_1. Set $\Delta_3 = I^1 \oplus \Delta_1$ and send that value to P_1 on behalf of P_3.

It is immediate that the distribution of the simulated view of P_1 in the ideal world is identical to the distribution of its view in a real execution of the protocol. Moreover, even conditioned on P_1's view, the outputs of P_2 and P_3 are uniform (subject to XORing to the correct output) in both the ideal- and real-world executions.

For the other case, assume P_2 is corrupted without loss of generality, and let i^2, I^2 be the corresponding input and output values given to the simulator. Here, the simulator works as follows:

1. Choose uniform r^2 on behalf of P_2. Choose uniform $\hat{r}^3$ and send it to P_2 on behalf of P_3.
2. Run $K^2, K^3 \leftarrow \mathsf{Gen}(1^\kappa, 1, 1)$, where Gen denotes the key-generation algorithm for a 2-server DPF with the appropriate domain and range. Send K^2 to P_2 on behalf of P_1. Let $\hat{I}^2$ be the local value that P_1 would compute in this step.
3. Choose uniform Δ_2 on behalf of P_2. Set $\Delta_1 = I^2 \oplus \hat{I}^2 \oplus \Delta_2$ and send that value to P_2 on behalf of P_2.

Security of the DPF readily implies that the distribution of the simulated view of P_2 in the ideal world is computationally indistinguishable from the distribution

of its view in a real execution of the protocol. Moreover, even conditioned on P_2's view, the outputs of P_1 and P_3 are uniform (subject to XORing to the correct output) in both the ideal- and real-world executions. □

Security of ShiftValue **(step 2).** The desired ideal functionality in this case takes additive shares I^1, I^2, I^3 of $I = I^1 \oplus I^2 \oplus I^3$ (which is a unary representation of i) as input from the parties, and then does the following for $k = 1, \ldots, \sqrt{N}$: choose $a_k, b_k, c_k, d_k \leftarrow \{0,1\}^\kappa$. Then if $k \neq i$, output to the parties

$$S_k^1 = \{a_k, b_k\}, \quad S_k^2 = \{b_k, c_k\}, \quad S_k^3 = \{c_k, a_k\},$$

respectively, while if $k = i$, output to the parties

$$S_k^1 = \{a_k, d_k\}, \quad S_k^2 = \{b_k, d_k\}, \quad S_k^3 = \{c_k, d_k\},$$

respectively. (In all cases, the elements in the set are randomly permuted so their order does not reveal information.)

In analyzing step 2 of ShiftValue, we assume the parties re-randomize their shares of I before running the protocol. This is justified by the fact that the parties re-randomize their shares at the end of step 1.

Claim. The second step of the GetValue protocol (with re-randomization of shares done first) securely realizes the functionality just described for a single, semi-honest corruption.

Proof. We analyze the protocol in a hybrid model where the parties have access to an oblivious-transfer (OT) functionality. By symmetry, we may assume without loss of generality that P_1 is corrupted. We describe a simulator that takes as input a value I^1 used as input by P_1 along with output values $S_1^1, \ldots, S_{\sqrt{N}}^1$, where $S_k^1 = \{a_k, b_k\}$, and simulates a view of P_1 in an execution of the protocol. The simulator works as follows:

1. Choose uniform Δ_1 on behalf of P_1. Choose uniform Δ_3 and send it to P_1 on behalf of P_3. (This simulates the re-randomization step.)
2. For $k = 1, \ldots, \sqrt{N}$, do:
   (a) Choose uniform $\alpha_k^1, \beta_k^1, \gamma_k^1, \delta_k^1$ on behalf of P_1. Choose uniform $I^2[k]$, β_k^2, γ_k^2, δ_k^2 and send them to P_1 on behalf of P_2.
   (b) Choose uniform x, y such that $x \oplus y = a_k$, and uniform x', y' such that $x' \oplus y' = b_k$. Simulate the OTs (with P_1 as receiver and P_2 as sender in one execution and P_3 as sender in the other execution) by giving P_1 outputs (x, x') and (y, y') from the two executions with probability $1/2$, and outputs (x', x) and (y', y) with probability $1/2$.
   (c) Choose uniform $\beta_k'', \gamma_k'', \delta_k''$, and z'' on behalf of P_1.
   (d) Choose uniform $\alpha_k''', \gamma_k''', \delta_k'''$, and z''' and send them to P_1 on behalf of P_2.

It is immediate that the distribution of the simulated view of P_1 in the ideal world is identical to the distribution of its view in a real execution of the protocol. Moreover, the distribution of the outputs of P_2 and P_3, conditioned on the inputs of all the parties and the output of P_1, is identically distributed in the ideal- and real-world executions. □

Security of ShiftValue **(step 3).** The proof of security for this step follows closely along the lines of the proof for step 1, and is therefore omitted. □

References

1. I. Abraham, C. W. Fletcher, K. Nayak, B. Pinkas, and L. 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. E. Boyle, N. Gilboa, and Y. Ishai. Function secret sharing: Improvements and extensions. In *23rd ACM Conf. on Computer and Communications Security (CCS)*, pages 1292–1303. ACM Press, 2016.
5. J. Doerner and A. Shelat. Scaling ORAM for secure computation. In *24th ACM Conf. on Computer and Communications Security (CCS)*, pages 523–535. ACM Press, 2017.
6. S. Faber, S. Jarecki, S. Kentros, and B. 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. S. Garg, D. Gupta, P. Miao, and O. 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. C. Gentry, K. Goldman, S. Halevi, C. Jutla, M. Raykova, and D. 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. M. T. Goodrich and M. 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. D. Gordon, J. Katz, V. Kolesnikov, F. Krell, T. Malkin, M. Raykova, and Y. 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. C. Hazay and A. 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. In *Applied Cryptography and Network Security (ACNS)*, volume 10892 of *LNCS*, pages 360–378. Springer, 2018. Available at `https://eprint.iacr.org/2018/347`.

15. J. Katz, D. Gordon, and X. Wang. Simple and efficient two-server ORAM. Asiacrypt 2018, to appear. Available at `https://eprint.iacr.org/2018/005`.
16. 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.
17. E. Kushilevitz and T. Mour. Sub-logarithmic distributed oblivious RAM with small block size. Available at `https://arxiv.org/pdf/1802.05145.pdf`, 2018.
18. E. Kushilevitz, S. Lu, and R. Ostrovsky. On the (in)security of hash-based oblivious RAM and a new balancing scheme. In *23rd SODA*, pages 143–156. ACM-SIAM, 2012.
19. C. Liu, Y. Huang, E. Shi, J. Katz, and M. W. Hicks. Automating efficient RAM-model secure computation. In *IEEE Symposium on Security and Privacy*, pages 623–638. IEEE, 2014.
20. S. Lu and R. Ostrovsky. Distributed oblivious RAM for secure two-party computation. In *9th Theory of Cryptography Conference—TCC 2013*, LNCS, pages 377–396. Springer, 2013.
21. 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.
22. K. Nayak, X.S. Wang, S. Ioannidis, U. Weinsberg, N. Taft, and E. Shi. GraphSC: Parallel secure computation made easy. In *IEEE Symposium on Security and Privacy*, pages 377–394. IEEE, 2015.
23. R. Ostrovsky and V. Shoup. Private information storage. In *29th Annual ACM Symposium on Theory of Computing (STOC)*, pages 294–303. ACM Press, 1997.
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. E. Shi, T.-H. H. Chan, E. Stefanov, and M. 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. E. Stefanov, M. van Dijk, E. Shi, C. W. Fletcher, L. Ren, X. Yu, and S. 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, T.-H. H. Chan, and E. 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.
28. X. Wang, S. D. Gordon, A. McIntosh, and J. Katz. Secure computation of MIPS machine code. In *ESORICS 2016, Part II*, volume 9879 of *LNCS*, pages 99–117. Springer, 2016.
29. X. S. Wang, Y. Huang, T.-H. H. Chan, A. Shelat, and E. Shi. SCORAM: Oblivious RAM for secure computation. In *21st ACM Conf. on Computer and Communications Security (CCS)*, pages 191–202. ACM Press, 2014.
30. P. Williams and R. Sion. Single round access privacy on outsourced storage. In *19th ACM Conf. on Computer and Communications Security (CCS)*, pages 293–304. ACM Press, 2012.
31. 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.