DUORAM: A Bandwidth-Efficient Distributed ORAM for 2- and 3-Party Computation

Adithya Vadapalli
avadapal@uwaterloo.ca
University of Waterloo

Ryan Henry
ryan.henry@ucalgary.ca
University of Calgary

Ian Goldberg
iang@uwaterloo.ca
University of Waterloo

Abstract

We design, analyze, and implement DUORAM, a fast and bandwidth-efficient distributed ORAM protocol suitable for secure 2- and 3-party computation settings. Following Doerner and shelat's FLORAM construction (CCS 2017), DUORAM leverages $(2,2)$-distributed point functions (DPFs) to represent PIR and PIR-writing queries compactly—but with a host of innovations that yield massive asymptotic reductions in communication cost and notable speedups in practice, even for modestly sized instances. Specifically, DUORAM introduces a novel method for evaluating dot products of certain secret-shared vectors using communication that is only logarithmic in the vector length. As a result, for memories with n addressable locations, DUORAM can perform a sequence of m arbitrarily interleaved reads and writes using just $O(m \lg n)$ words of communication, compared with FLORAM's $O(m\sqrt{n})$ words. Moreover, most of this work can occur during a data-independent preprocessing phase, leaving just $O(m)$ words of online communication cost for the sequence—i.e., a *constant online communication cost per memory access*.

1 Introduction

Oblivious RAM (ORAM) allows a client to outsource data storage to one or more untrusted servers. The client can then read from or write to the outsourced storage (called a *database* or a *memory*) without revealing to the servers anything about its access patterns (i.e., which memory addresses it accesses or whether those accesses are reads or writes). Although initially proposed as a general software security tool [14], the past few years have seen increased attention on *distributed ORAM* (DORAM) constructions [5, 8, 10, 17–19, 21, 22] optimized for data-dependent yet oblivious memory accesses in secure multiparty computation (MPC) settings. Note that in such MPC settings, all parties typically know the *algorithm* being executed, and so the requirement that reads and writes be indistinguishable from each other is relaxed.

This paper presents DUORAM, a DORAM protocol with instantiations in either 2- or 3-party settings tolerating a single passive corruption. DUORAM follows a similar design to Doerner and shelat's DPF-based FLORAM construction [8]; however, despite their shared lineage and structural similarities, DUORAM offers several *substantial*—and, we argue, *surprising*—theoretical and practical advantages relative to FLORAM. Most notably, DUORAM exploits a subtle observation to achieve the (seemingly) impossible: supporting sequences of arbitrarily interleaved, oblivious reads and writes with online communication *independent of the memory size* for realistically sized memories.[1] Indeed, even the *total* (i.e., preprocessing plus online) communication cost scales logarithmically in the memory size.

1.1 Overview of state of the art

The state-of-the-art DORAM construction from prior work is Doerner and shelat's FLORAM [8], a 2-party construction built from *garbled circuits* and $(2,2)$*-distributed point functions* (DPFs). We defer an in-depth description of DPFs to Section 2.1; for now, it suffices to regard DPFs as concise, secret-shared representations of standard basis vectors (i.e., of vectors $\vec{\mathbf{e}}_{\mathtt{i}}$ comprising all 0s except for a single 1 appearing in coordinate $\mathtt{i}$). FLORAM uses DPFs to implement both *private information retrieval* (PIR) and *PIR-writing*, a pair of cryptographic primitives that respectively allow remote users to download items from and write items to databases held by remote and untrusted servers. In both instances, the security goal is to hide the address accessed (and contents of that address) from the servers.

Throughout our description of FLORAM, we consider mem-

[1] Specifically, computation parties in DUORAM exchange a constant number of secret shares per memory access. Some of these shares encode memory addresses $\mathtt{i} \in [0..n)$ while others encode w-bit words of data (such as those stored at the addressed memory); thus, strictly speaking, DUORAM's per-access online communication complexity is logarithmic in n and linear in w. Our implementation fixes $w = 64$ and bounds $\lg n \leq 64$ and therefore supports up to $n = 2^{64}$ memory locations, each holding a 64-bit word of data, for a theoretical limit of 128 EiB of memory while using a constant number of online communication words per access.

ory $D \in \{0,1\}^{n\times w}$ consisting of n words that are each w bits long. We denote the word at memory address $\mathtt{i}$ in D by $D[\mathtt{i}] \in \{0,1\}^w$. Depending on the type of memory access being performed, the computation parties hold either an encrypted copy of D (when reading) or a secret shared copy of D (when writing); a "refresh" operation converts D from its secret-shared representation to its encrypted one. FLORAM also makes use of a "stash" (the details of which we gloss over) to reduce the frequency with which the computation parties must invoke the (rather costly) refresh operation, resulting in a lower amortized cost per memory access.

Oblivious reads in FLORAM. Computation parties P_0 and P_1 hold symmetric keys k_0 and k_1 respectively alongside the memory D blinded using a pseudorandom function F; i.e., P_0 and P_1 hold in common blinded memory $\bar{D} \in \{0,1\}^{n\times w}$ such that $\bar{D}[\mathtt{i}] \leftarrow D[\mathtt{i}] \oplus \mathsf{F}(\mathsf{k}_0,\mathtt{i}) \oplus \mathsf{F}(\mathsf{k}_1,\mathtt{i})$ for each $\mathtt{i} \in [0..n)$.

Given shares of a *target address* $\mathtt{i}^* \in [0..n)$, they obtain shares of the corresponding word $D[\mathtt{i}^*]$ using simple PIR followed by an oblivious unblinding step, as follows:

1. P_0 and P_1 collaboratively sample a DPF representation of $\vec{\mathbf{e}}_{\mathtt{i}^*}$ using 2-MPC;

2. each P_b locally expands its DPF share into a bit vector $\vec{\mathbf{t}}_b \in \{0,1\}^n$ and then it computes $\mathsf{R}^{\Sigma}_b \leftarrow \bigoplus_{(\vec{\mathbf{t}}_b[i]=1)} \bar{D}[i]$; and, finally,

3. P_0 and P_1 run another 2-MPC to evaluate F at the (unknown to either party) input $\mathtt{i}^*$ to produce shares of $D[\mathtt{i}^*] = \mathsf{R}^{\Sigma}_0 \oplus \mathsf{R}^{\Sigma}_1 \oplus \mathsf{F}(\mathsf{k}_0,\mathtt{i}^*) \oplus \mathsf{F}(\mathsf{k}_1,\mathtt{i}^*)$.

Oblivious writes in FLORAM. Computation parties P_0 and P_1 hold XOR-shares D_0 and D_1 of the memory D.

Given shares of a *target index–value* pair $(\mathtt{i}^*,\mathsf{M})$, they replace $D[\mathtt{i}^*]$ with $D[\mathtt{i}^*] \oplus \mathsf{M}$ using PIR-writing, as follows:

1. P_0 and P_1 collaboratively sample a DPF representation of $\vec{v} = \mathsf{M} \cdot \vec{\mathbf{e}}_{\mathtt{i}^*}$ using 2-MPC; and

2. each P_b locally expands its DPF share into an n-element vector $\vec{v}_b \in (\{0,1\}^w)^n$ and then it computes $D'_b \leftarrow D_b \oplus \vec{v}_b$.

Of course, the parties can write an arbitrary value of their choosing first using $\mathtt{i}^*$ to read shares of $D[\mathtt{i}^*]$, and then invoking the above procedure for the target index–value pair $(\mathtt{i}^*, \mathsf{M} \oplus D[\mathtt{i}^*])$.

Refreshing the FLORAM database. Recall that FLORAM requires an encrypted copy of D for reading and an XOR-shared copy of D for writing. The refresh operation transforms XOR-shared memory (suitable for writing) into encrypted memory (suitable for reading). Given shares D_0 and D_1 of D, they compute the encrypted memory as follows:

1. For every refresh, P_0 and P_1 pick fresh keys k_0 and k_1 respectively, and mask their local copy of the database; i.e, for all $\mathtt{i}$, P_0 and P_1 compute $\mathsf{R}[\mathtt{i}]' \leftarrow D_0[\mathtt{i}] \oplus \mathsf{F}(\mathsf{k}_0,\mathtt{i})$ and $\mathsf{R}[\mathtt{i}]'' \leftarrow D_1[\mathtt{i}] \oplus \mathsf{F}(\mathsf{k}_1,\mathtt{i})$ respectively.

2. For all $\mathtt{i}$, P_0 and P_1 exchange $\mathsf{R}[\mathtt{i}]'$ and $\mathsf{R}[\mathtt{i}]''$ to compute $\widetilde{\mathsf{R}}[\mathtt{i}] \leftarrow \mathsf{R}[\mathtt{i}]' \oplus \mathsf{R}[\mathtt{i}]''$, resulting a communication complexity of $O(n)$ words.

The doubly-masked memory serves as the "read only" memory. FLORAM uses this refresh operation to initialize their ORAM with the two versions of the database, thus incurring a $O(n)$ communication cost to begin any access to the ORAM. A read operation following a write operation would require FLORAM to do another linear-cost refresh operation, to get a new version of the masked database. However, FLORAM uses a $O(\sqrt{n})$-sized *stash* to reduce the communication cost from $O(n)$ to $O(\sqrt{n})$.

1.2 Our Contributions

FLORAM proposed an ORAM with $O(n)$ local computation but a communication complexity of only $O(\sqrt{n})$, and which in practice beats prior polylog schemes. This paper presents a novel DORAM scheme, called DUORAM, which takes it a step further—we maintain the computation complexity of $O(n)$, but substantially reduce the concrete costs, while reducing the communication complexity to $O(\log n)$ and offloading almost the entire communication to a preprocessing phase. The main contributions of our work are as follows:

1. A new *preprocessing* strategy that enables the computation parties to generate and (partially) evaluate DPFs ahead of time, before any target indices or values are known;

2. novel *3-party read* and *3-party write* procedures, both of which (i) operate directly on secret-shared memory, (ii) use only a single round of interaction, and (iii) incur online communication cost independent of the memory size; and

3. 2-party variants of the above 3-party protocols that eliminate one of the three servers using a single-server Symmetric PIR-based protocol.

The second contribution listed above eliminates the need for an explicit refresh operation or a stash (it also makes initializing memory free, compared with FLORAM's $O(n)$ communication for initialization), yielding a blazing fast 3-party DORAM protocol. The third contribution yields a 2-party protocol with far lower asymptotic and concrete communication than FLORAM, albeit with a higher constant for the linear local computation.

Some of the salient features of DUORAM's online phase are: (i) online communication complexity of $O(1)$ for reading

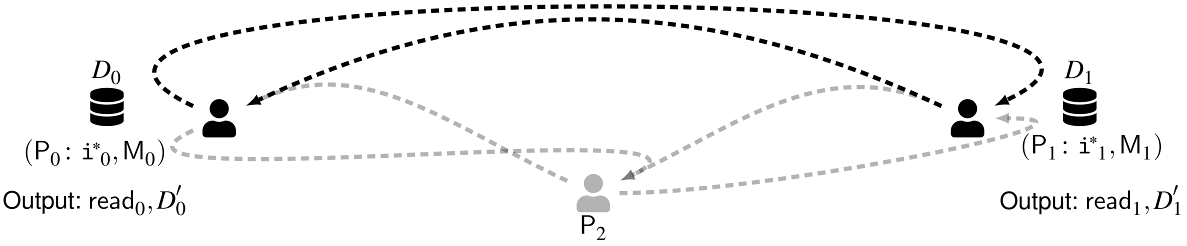

D_0
$(\mathsf{P}_0 : \mathtt{i}^*_0, \mathsf{M}_0)$
Output: read_0, D'_0
D_1
$(\mathsf{P}_1 : \mathtt{i}^*_1, \mathsf{M}_1)$
Output: read_1, D'_1
P_2

Figure 1: DUORAM System. P_0 and P_1 hold shares D_0 and D_1 of the database D, shares $\mathtt{i}^*_0$ and $\mathtt{i}^*_1$ of the index $\mathtt{i}^*$ they wish to access, and shares M_0 and M_1 of the update value M. All shares are additive, so that $\mathtt{i}^*_0 + \mathtt{i}^*_1 = \mathtt{i}^*$ and similar. The outputs of the read operation are read_0 and read_1, with the property that $\mathsf{read}_0 + \mathsf{read}_1 = D[\mathtt{i}^*]$. D'_0 and D'_1 are the outputs of the write operation, with $D'_0[\mathtt{i}] + D'_1[\mathtt{i}] = D[\mathtt{i}]$ for $\mathtt{i} \neq \mathtt{i}^*$, and $D'_0[\mathtt{i}^*] + D'_1[\mathtt{i}^*] = D[\mathtt{i}^*] + \mathsf{M}$.

and writing, (ii) doing k reads in parallel takes one round of communication, (iii) doing k updates in parallel requires one message being exchanged, and (iv) doing k dependent reads (addresses to read depend on the outcome of previous reads) takes k rounds of communication. Some features of DUORAM's preprocessing phase are: (i) the communication complexity is $O(m \lg n)$ for m accesses on memory of size n, (ii) the computation complexity is $O(n)$ *cheap* operations per ORAM access for memory of size n, and (iii) round complexity of $O(\lg n)$, independent of the number of accesses. The concrete advantages of DUORAM are borne out in our experiments, where we observe that even throttling throughput to as little as 1 Mbit/s has only a nominal impact on DUORAM's performance. This is in contrast with FLORAM, which did not finish the computations after more than ten hours.

1.3 System Overview

DUORAM can be instantiated either as a 2-party or a 3-party protocol. The DUORAM protocol at a very high level works as follows. The database is (additively) secret shared among two parties, namely, P_0 and P_1. The two parties also hold the additive shares of a target index in the database that they want to access.[2] If they wish to perform a write (or an update) operation, the two parties also hold the shares of the value they wish to write at the target index. Three-party DUORAM has a (stateful) auxiliary party, namely P_2. The auxiliary party does not hold the database or index shares and merely facilitates the secure multiparty computation. Our 2-party DUORAM replaces the third party with a Computational Symmetric PIR (CSPIR) protocol. Figure 1 describes the DUORAM system. The party with reduced opacity is the auxiliary party that does not hold either the database shares or the shares of the database index.

Organization. This paper is organized as follows. In Section 2 we discuss the background needed for DUORAM. Section 3 describes the various DUORAM protocols and we have detailed discussion of the 3-party protocol in Section 4. Section 5 describes our CSPIR-based 2-party DORAM protocol. We experimentally evaluate the protocols in Section 6. Section 7 describes related work, and Section 8 concludes.

2 Background

2.1 Distributed Point Functions (DPFs)

In the most simple terms, Distributed Point Functions (DPFs) are a concise way to share a standard basis vector (or more generally a 1-hot vector, which is a scaled standard basis vector) among multiple parties. Gilboa and Ishai [12] were the first to introduce DPFs. Boyle, Gilboa, and Ishai [3, 4] improved upon the original DPF construction. In this paper, we will concern ourselves with the most compact DPF construction, which appears in Boyle, Gilboa, and Ishai's follow-up paper [4].

We will begin the discussion of DPFs by first describing a point function. A point function is a function that evaluates to 0 at every value in their domain, except at one special point (called the "target point"), where it evaluates to a non-zero value (called the "target value"). In the definition below (Definition 1), $\mathtt{i}^*$ is the "target point" and M is the "target value".

Definition 1. *A point function is a function $p_{\mathtt{i}^*,\mathsf{M}} : [0,n) \to \{0,1\}^*$ such that $p_{\mathtt{i}^*,\mathsf{M}}(\mathtt{i}^*) = \mathsf{M}$ and $p_{\mathtt{i}^*,\mathsf{M}}(\mathtt{i}) = 0$ otherwise.*

Observe that we can represent a point function as a binary tree (see Figure 2). Distributed Point Functions are a concise way to share a point function among two or more parties. An (m,t)-DPF distributes a point function among m parties, such that no coalition of fewer than t parties can learn the target point or the target value. This paper deals specifically with $(2,2)$-DPFs, where the goal is to share a point function among

[2] Storing the target indices as additive shares rather than XOR shares makes DUORAM more efficient. However, the database can either be additive or XOR shares. We choose to keep them as additive shares purely for consistency.

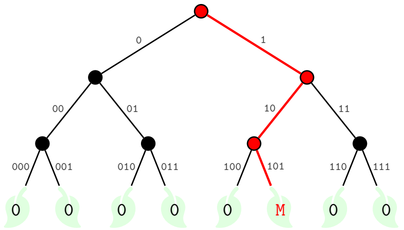


Figure 2: The Point Function Binary Tree with target value as M

two parties succinctly. Definition 2 (which is Definition 4 from Sabre [28]) formally defines $(2,2)$-DPFs.

Definition 2. *A* $(2,2)$-distributed point function, *or* $(2,2)$-DPF, *is a pair of PPT algorithms* $(\textsc{Gen},\textsc{Eval})$ *defining secret-shared representations of point functions; that is, given (i) a security parameter* $\lambda \in \mathbb{N}$, *(ii) a target point* $\mathtt{i}^*$, *and (iii) a target value* M, *we have*

1. **Correctness:** *If* $(k_0,k_1) \leftarrow \textsc{Gen}(1^\lambda,\mathtt{i}^*,\mathsf{M})$, *then, for all* $\mathtt{i} \in [0,n)$,
$$\textsc{Eval}(k_0,\mathtt{i}) + \textsc{Eval}(k_1,\mathtt{i}) = \begin{cases} \mathsf{M} & \textit{if } \mathtt{i} = \mathtt{i}^*\textit{, and} \\ 0 & \textit{otherwise.} \end{cases}$$

2. **Simulatability:** *There exists a PPT simulator* $\mathcal{S}$ *such that, for a tuple of target index and target value,* $(\mathtt{i}^*,\mathsf{M})$, *and bit* $b \in \{0,1\}$, *the distribution ensembles* $\{\mathcal{S}(1^\lambda,b)\}_{\lambda\in\mathbb{N}}$ *and* $\{k_b \mid (k_0,k_1) \leftarrow \textsc{Gen}(1^\lambda,\mathtt{i}^*,\mathsf{M})\}_{\lambda\in\mathbb{N}}$ *are computationally indistinguishable.*

The k_b *output by* $\textsc{Gen}$ *are called* $(2,2)$-DPF keys.

2.1.1 DPF construction

We will now describe the most compact construction of DPFs due to Boyle, Gilboa, and Ishai [4]. The key ingredient used in their construction is the so-called length-doubling PRG, which we represent by $G_{2\times}$, to construct Goldreich, Goldwasser, Micali styled PRFs [13]. We denote the left and right halves of the outputs of $G_{2\times}(\cdot)$ as $G_L(\cdot)$ and $G_R(\cdot)$ respectively. Boyle, Gilboa, and Ishai's construction follows from the following observation: $G_{2\times}(\mathsf{s}_0) = G_{2\times}(\mathsf{s}_1)$ if and (essentially) only if $\mathsf{s}_0 = \mathsf{s}_1$.

The $\textsc{Gen}$ algorithm (for a DPF tree with height h) begins with selecting two random seeds, namely, $\mathsf{s}_0^{()}$ and $\mathsf{s}_1^{()}$. The main idea is to use a length-doubling PRG recursively on the two seeds to construct a pair of binary trees. In our notation (which we borrow from Sabre [28]), the nodes in the binary tree generated by recursively applying the length-doubling PRG have a subscript, a bit indicating which party's tree the node belongs to, and a superscript, a binary string indicating the path from the root to the node (the tree's root is $\mathsf{s}_b^{()}$). For any node $\mathsf{s}_b^{(\mathtt{i})}$, where $\mathtt{i}$ is a binary string, the outputs of the length-doubling PRG are $(\mathsf{s}_b^{(\mathtt{i}\|0)},\mathsf{s}_b^{(\mathtt{i}\|1)})$. We call two nodes with the same superscript but different subscripts a *node pair*.

The main idea is to use the length-doubling PRG recursively on the two seeds to construct a pair of binary trees that are equal everywhere except on the path to leaf $\mathtt{i}^*$, and that the node pair for leaf $\mathtt{i}^*$ XOR to M. Of course, the trees generated by just the length-doubling PRG will not have this form. Therefore, there are two other ingredients associated with DPFs, which allow us to fix these random trees, namely, (i) correction words, and (ii) flag bits. There is a "correction word" associated with each binary tree *level* and a flag bit associated with each *node*. For the tree pairs of height h, suppose that $(\mathsf{cw}^{(1)},\dots,\mathsf{cw}^{(h)})$ represent the correction words. In the first step, the two root seeds are expanded to get $(\mathsf{s}_0^{(0)},\mathsf{s}_0^{(1)}) \leftarrow G_{2\times}(\mathsf{s}_0^{()})$ and $(\mathsf{s}_1^{(0)},\mathsf{s}_1^{(1)}) \leftarrow G_{2\times}(\mathsf{s}_1^{()})$. Next, exactly one pair of children $(\mathsf{s}_b^{(0)},\mathsf{s}_b^{(1)})$ is transformed such that, $\mathsf{s}_b^{(0)} \leftarrow \mathsf{s}_b^{(0)} \oplus \mathsf{cw}^{(1)}$ and $\mathsf{s}_b^{(1)} \leftarrow \mathsf{s}_b^{(1)} \oplus \mathsf{cw}^{(1)}$, while the other pair of children $(\mathsf{s}_{1-b}^{(0)},\mathsf{s}_{1-b}^{(1)})$ remains the same. In other words, for exactly one of the seeds, the "correction word" associated with level 1 is XOR-ed into the children. The flag bit associated with the seed determines if the party XORs correction word into its children. Naturally, the flag associated with one of the seeds is set to 1, while the other is 0. (In general, the flag bits will be the same for the two nodes in each node pair, except on the path from the root to leaf $\mathtt{i}^*$, where they will be different.) The correction words are designed so that after the transformation we have the property that either $\mathsf{s}_0^{(0)} = \mathsf{s}_1^{(0)}$ (if $\mathtt{i}^*$ starts with a 1 bit) or $\mathsf{s}_0^{(1)} = \mathsf{s}_1^{(1)}$ (if $\mathtt{i}^*$ starts with a 0 bit). Notice that there are two types of node pairs, (i) *equal*, where the two nodes in the pair are the same, and (ii) *unequal*, where the two nodes in the pair are not the same. Observe that the children of the two nodes from an *equal* pair are the same; thus, they are both part of an *equal* pair in the next layer. Similarly, the children of the nodes from an *unequal* pair are part of an *unequal* pair in the next layer. At each layer, applying the correction word transforms exactly one of the children of a node from an *unequal* pair (the child *not* on the path to leaf $\mathtt{i}^*$) to being part of an *equal* pair. Applying h correction words results in leaves such that all the leaf pairs except $\mathtt{i}^*$ of the two trees XOR to 0, and leaf pair $\mathtt{i}^*$ XORs to something random. Therefore, DPFs also have a notion of a final correction word, denoted as $\mathcal{F}$. Unlike the correction words, $(\mathsf{cw}^{(1)},\dots,\mathsf{cw}^{(h)})$, whose goal is to equalize a particular node in the path from the root to the leaf, $\mathcal{F}$ converts the random node at the target location $\mathtt{i}^*$ into the target value M. Figure 3 shows an example construction of DPFs. We remark that the two DPF trees in Figure 3 reconstruct the point function tree in Figure 2. We can evaluate the DPF over the entire domain of the point function using the function $\textsc{EvalFull}$. The function produces $(\vec{v}_b,\vec{\mathbf{t}}_b) \leftarrow \textsc{EvalFull}(\mathtt{k}_b)$ where $\vec{v}_b$ is the vector of labels of all the leaves in tree b and $\vec{\mathbf{t}}_b$ is the vector of the flags on all the leaves in tree b. We have the

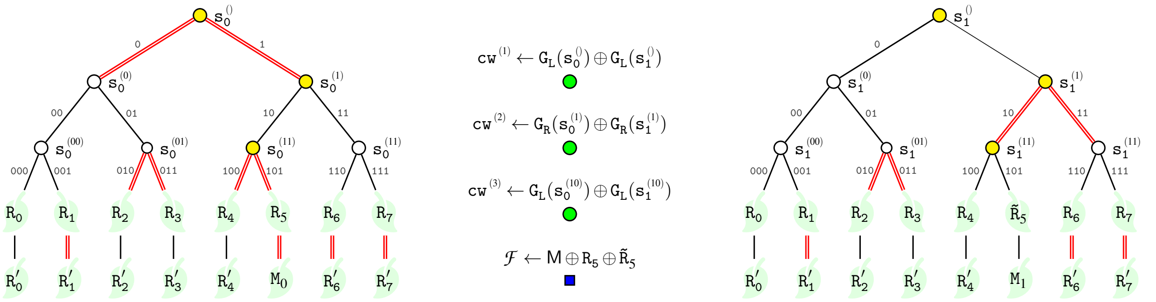

$s_0^{()}$
0
1
$s_0^{(0)}$
$s_0^{(1)}$
00
01
10
11
$s_0^{(00)}$
$s_0^{(01)}$
$s_0^{(11)}$
$s_0^{(11)}$
000
001
010
011
100
101
110
111
R_0 R_1 R_2 R_3 R_4 R_5 R_6 R_7
R'_0 R'_1 R'_2 R'_3 R'_4 M_0 R'_6 R'_7
$cw^{(1)} \leftarrow G_L(s_0^{()}) \oplus G_L(s_1^{()})$
$cw^{(2)} \leftarrow G_R(s_0^{(1)}) \oplus G_R(s_1^{(1)})$
$cw^{(3)} \leftarrow G_L(s_0^{(10)}) \oplus G_L(s_1^{(10)})$
$\mathcal{F} \leftarrow M \oplus R_5 \oplus \tilde{R}_5$
$s_1^{()}$
$s_1^{(0)}$
$s_1^{(1)}$
$s_1^{(00)}$
$s_1^{(01)}$
$s_1^{(11)}$
$s_1^{(11)}$
R_0 R_1 R_2 R_3 R_4 $\tilde{R}_5$ R_6 R_7
R'_0 R'_1 R'_2 R'_3 R'_4 M_1 R'_6 R'_7

Figure 3: The yellow-colored nodes are *unequal* pairs and the white colored nodes are *equal* pairs. The green colored circles represent the correction words and the blue colored square is the final correction word. A red-colored double edge indicates that the correction associated with that level is XOR-ed into the PRG output. In other words, it also indicates that red color edges emanating from a parent indicate that the flag bit associated with the parent is set to 1, and black edges indicate that the flag bit is set to 0. For instance, in the root layer, we have $s_0^{(0)} \leftarrow G_L(s_0^{()}) \oplus cw^{(1)}; s_0^{(1)} \leftarrow G_R(s_0^{()}) \oplus cw^{(1)}; s_1^{(0)} \leftarrow G_L(s_1^{()}); s_1^{(1)} \leftarrow G_R(s_1^{()})$

property that $\vec{\mathbf{t}}_0 \oplus \vec{\mathbf{t}}_1 = \vec{\mathbf{e}}_{i^*}$ and $\vec{v}_0 \oplus \vec{v}_1 = \vec{\mathbf{e}}_{i^*} \cdot M$. Also, we have for all i that $\vec{v}_b[i] = \textsc{Eval}(k_b, i)$.

2.2 Secure Multi-Party Computation (MPC)

We will first present an informal description of MPC. Consider a situation where parties $P_0, P_1, \ldots, P_n$ hold secret inputs $x_0, x_1, \ldots, x_n$ respectively and would like to compute the function $y = f(x_0, \ldots, x_n)$ without revealing the secret inputs. The goal is that no party i learns anything else except y. In other words, P_i should not learn any x_j (except the information about x_j implied by y), where $j \neq i$. A special case of MPC is 2-MPC, which has exactly two parties in the protocol. It is defined as follows:

Definition 3 (2-MPC, Informal)**.** *Parties P_0 and P_1 hold private inputs x_0 and x_1 respectively. Their goal is to compute a function $f(x_0, x_1)$ with the following properties: (i)* ***Privacy:*** *No party should learn anything more than what is implied by the final output, and (ii)* ***Correctness:*** *Parties P_0 and P_1 learn the correct output.*

Loosely speaking, $(2+1)$-MPC, also known as a server-aided MPC protocol, is a 2-party MPC protocol with a third party (which does not hold any secret input) that does not collude with any other two parties. The third party merely facilitates the MPC by sending some randomness to the two MPC parties.

2.2.1 Multiplication and Dot Product on Secret Shares

We first define the notion of multiplicative triples (on XOR shares) and dot-product triples (on additive shares), which facilitate computing multiplication and dot products on secret shares. In Appendix A, we describe two main ways to generate these triples, namely a $(2+1)$-party protocol (using a stateless third party) and a 2-party protocol (using oblivious transfer, or OT). The former leads to the Du-Atallah [9] protocol for multiplication (and dot products), which is an efficient $(2+1)$ variant of the well-known Beaver triples [1] based multiplication.

Definition 4 (Multiplicative Triples)**.** *(X_0, Y_0, Z_0) and (X_1, Y_1, Z_1) are called multiplicative triple if for a random T, the following holds true: (i) $Z_0 = (X_0 \wedge Y_1) \oplus T$, and (ii) $Z_1 = (X_1 \wedge Y_0) \oplus T$.*

Definition 5 (Dot-Product Triples)**.** *$(\vec{X}_0, \vec{Y}_0, Z_0)$ and $(\vec{X}_1, \vec{Y}_1, Z_1)$ are called dot-product triple if for a random T, the following holds true: (i) $Z_0 = \langle \vec{X}_0, \vec{Y}_1 \rangle + T$, and (ii) $Z_1 = \langle \vec{X}_1, \vec{Y}_0 \rangle - T$.*

P_0 and P_1 hold (x_0, y_0) and (x_1, y_1) respectively and their goal is to compute shares of $(x_0 \oplus x_1) \wedge (y_0 \oplus y_1)$. Suppose P_0 and P_1 hold the multiplicative triples (X_0, Y_0, Z_0) and (X_1, Y_1, Z_1). First, P_0 sends $(x_0 \oplus X_0)$ and $(y_0 \oplus Y_0)$ to P_1 and vice versa. P_0 computes $z_0 \leftarrow (x_0 \wedge (y_0 \oplus (y_1 \oplus Y_1))) \oplus Y_0 \wedge (x_1 \oplus X_1) \oplus Z_0$ and P_1 computes $z_1 \leftarrow (x_1 \wedge (y_1 \oplus (y_0 \oplus Y_0))) \oplus Y_1 \wedge (x_0 \oplus X_0) \oplus Z_1$ respectively. Now, observe that $z_0 \oplus z_1 = (x_0 \oplus x_1) \wedge (y_0 \oplus y_1)$.

Similarly, to compute dot products, suppose that parties P_0 and P_1 hold $(\vec{x}_0, \vec{y}_0)$ and $(\vec{x}_1, \vec{y}_1)$ respectively, such that $\vec{x} = \vec{x}_0 + \vec{x}_1$ and $\vec{y} = \vec{y}_0 + \vec{y}_1$. Their goal is to obtain shares of $\langle \vec{x}, \vec{y} \rangle$. Suppose, P_0 and P_1 hold the dot-product triples $(\vec{X}_0, \vec{Y}_0, Z_0)$ and $(\vec{X}_1, \vec{Y}_1, Z_1)$. P_0 sends $(\vec{x}_0 + \vec{X}_0)$ and $(\vec{y}_0 + \vec{Y}_0)$ to P_1 and vice versa. P_0 computes $z_0 \leftarrow \langle \vec{x}_0, (\vec{y}_0 + (\vec{y}_1 + \vec{Y}_1)) \rangle - \langle \vec{Y}_0, (\vec{x}_1 + \vec{X}_1) \rangle + Z_0$ and P_1 computes $z_1 \leftarrow \langle \vec{x}_1, (\vec{y}_1 + (\vec{y}_0 + \vec{Y}_0)) \rangle - \langle \vec{Y}_1, (\vec{x}_0 + \vec{X}_0) \rangle + Z_1$ respectively. Now, observe that $z_0 + z_1 = \langle (\vec{x}_0 + \vec{x}_1), (y_0 + y_1) \rangle$.

3 Communication Efficient 3-Party DORAM

We will now present the construction of our communication-efficient DORAM, which we call Duoram. The main feature of Duoram is that we avoid the need for a linear communication cost *refresh* operation like the one used by Floram, described in Section 1.1. Avoiding the *refresh* operation is possible because Duoram, unlike Floram, stores the database as additive secret shares for both Read and Write operations. Furthermore, since Duoram does not ever change how data is stored in memory, its initialization is free. Another surprising consequence of avoiding the refresh operation is that we have an online communication cost independent of the database size.

Notation. Throughout the paper, we will use up to three parties, namely P_0, P_1, and P_2. Parties P_0 and P_1 are the two *primary* parties. P_2 is the *helper* party. P_2's role is restricted to either maintaining a state or assisting in the MPC protocol by sending correlated randomness to the primary parties. In Duoram's 2-party version we replace P_2 with a CSPIR protocol. We denote by $D \in \{0,1\}^{n\times w}$ the database into which we read or write. D_0 and D_1 denote the additive shares of the database. In other words, $D_0 + D_1 = D$. All additive secret sharings are treated as integers mod 2^w, or vectors of same. We use $D[\mathtt{i}] \in \{0,1\}^w$ to denote the $\mathtt{i}^{\text{th}}$ word of the database. Similarly, $D_b[\mathtt{i}]$ denotes the $\mathtt{i}^{\text{th}}$ word of D_b, thus they are the shares of the $\mathtt{i}^{\text{th}}$ word of the database. We denote by $\zeta_b \in \{0,1\}^{n\times w}$ the blinding factors held by P_b. We use $\widetilde{D}_b$ to denote the blinded shares of the database, i.e., $\widetilde{D}_b \leftarrow D_b + \zeta_b$. In our Read protocol, P_b implicitly receives $\widetilde{D}_{1-b}$ from P_{1-b}. We denote by $\mathtt{i}^*$ the database index into which we want to read or write. M denotes the value by which we want to update the value at index $\mathtt{i}^*$. The shares of $\mathtt{i}^*$ are $\mathtt{i}^*{}_0$ and $\mathtt{i}^*{}_1$. Similarly, M_0 and M_1 are the shares of M.

3.1 Duoram operations

P_0 holds $(D_0, \widetilde{D}_1, \zeta_0, \mathtt{i}^*{}_0, \mathsf{M}_0)$ and P_1 holds $(D_1, \widetilde{D}_0, \zeta_1, \mathtt{i}^*{}_1, \mathsf{M}_1)$. Duoram initializes the database shares and the Du-Atallah blinding vectors to all zeros, thus making initialization free, compared to the $O(n)$ communication cost for initializing Floram. Formally, the initialization results in $D_0 \leftarrow \mathbf{0}$, $D_1 \leftarrow \mathbf{0}$, $\zeta_0 \leftarrow \mathbf{0}$, $\zeta_1 \leftarrow \mathbf{0}$, $\widetilde{D}_0 \leftarrow \mathbf{0}$, $\widetilde{D}_1 \leftarrow \mathbf{0}$.

Read The goal is to read from the database, D, a word from the target index, $\mathtt{i}^*$. In other words, P_0 and P_1, who hold the additive shares of $\mathtt{i}^*$, along with the shares of the database D, want to obtain the additive shares of $D[\mathtt{i}^*]$. We observe that $D[\mathtt{i}^*] = \langle D, \vec{\mathbf{e}}_{\mathtt{i}^*}\rangle$. While we describe the details of Read operation in Section 4, we mention here that we perform the read operation via a variant of a Du-Atallah dot-product protocol to compute the dot-product of the flag vectors associated with a DPF at $\mathtt{i}^*$ with the database. The flag vectors from the DPFs are XOR shares of a standard basis vector. In Section 4.1.1 we will show a procedure to convert them to additive shares.

Update The goal is to add a value to the element at the target index of the database. Parties P_0 and P_1 hold $(\mathsf{M}_0, \mathtt{i}^*{}_0)$ and $(\mathsf{M}_1, \mathtt{i}^*{}_1)$ respectively. Their goal is to obtain shares of a new database, whose $\mathtt{i}^{*\text{th}}$ value is updated by M. A Write operation is a Read operation (with the output read_b) followed by an Update operation with $\mathsf{M}_b - \mathsf{read}_b$.

Notice that, once we have performed an Update operation, to perform another Read operation, we require P_0 and P_1 to get fresh blinding factors from P_2 and exchange the new blinded shares of the database, which naively would result in $O(n)$ communication per Read operation. Our novel RefreshBlinds operation, which is a key innovation of Duoram, allows us to avoid this $O(n)$ communication. Rather than having the natural exchange of the blinded shares, the RefreshBlinds operation re-randomizes the existing blinding factors and updates the blinded shares corresponding to the re-randomized blinding factors, thus avoiding the expensive communication of blinding factors and blinded shares. The goal here is to update, (i) the blinding factors ζ_0 and ζ_1 held by P_0 and P_1 respectively, and (ii) the blinded database shares, namely, $\widetilde{D}_0$ (held by P_1, received from P_0) and $\widetilde{D}_1$ (held by P_0, received from P_1), such that they correspond to the new blinding factors.

3.2 High-level working of Duoram

For the high-level exposition of Duoram, we consider a trusted source, given a target location $\mathtt{i}^*$ and a target value M,[3] who creates six pairs of word vectors (mod 2^w) with the property that $\vec{\mathsf{b}}_0 + \vec{\mathsf{b}}_1 = \vec{\mathsf{c}}_0 + \vec{\mathsf{c}}_1 = \vec{\mathsf{d}}_0 + \vec{\mathsf{d}}_1 = \vec{\mathbf{e}}_{\mathtt{i}^*}$, and $\vec{\mathsf{b}}^*_0 + \vec{\mathsf{b}}^*_1 = \vec{\mathsf{c}}^*_0 + \vec{\mathsf{c}}^*_1 = \vec{\mathsf{d}}^*_0 + \vec{\mathsf{d}}^*_1 = \mathsf{M} \cdot \vec{\mathbf{e}}_{\mathtt{i}^*}$. The trusted source sends: (i) $(\vec{\mathsf{b}}_0, \vec{\mathsf{c}}_0, \vec{\mathsf{d}}_0)$ to P_0, (ii) $(\vec{\mathsf{b}}_1, \vec{\mathsf{c}}_1, \vec{\mathsf{d}}_1)$ to P_1, and (iii) $(\vec{\mathsf{c}}_1, \vec{\mathsf{d}}_0)$ to P_2, and correspondingly: (i) $(\vec{\mathsf{b}}^*_0, \vec{\mathsf{c}}^*_0, \vec{\mathsf{d}}^*_0)$ to P_0, (ii) $(\vec{\mathsf{b}}^*_1, \vec{\mathsf{c}}^*_1, \vec{\mathsf{d}}^*_1)$ to P_1, and (iii) $(\vec{\mathsf{c}}^*_1, \vec{\mathsf{d}}^*_0)$ to P_2. We note the $\vec{\mathsf{b}}_b$ and $\vec{\mathsf{b}}^*_b$ vectors (and similarly for $\vec{\mathsf{c}}$ and $\vec{\mathsf{d}}$) are *almost* the flag and label vectors of a DPF, except the $\vec{\mathsf{b}}$, $\vec{\mathsf{c}}$, and $\vec{\mathsf{d}}$ vectors are additively shared, while DPFs are XOR-shared (and the flag vectors of DPFs are bit vectors, not word vectors). We will see later how to convert DPFs into these additively shared vectors.

3.2.1 Read Operation

The main idea is to use Du-Atallah-style dot products to compute the shares of $\langle D, \vec{\mathbf{e}}_{\mathtt{i}^*}\rangle$. However, there is a critical difference between the original Du-Atallah to compute the dot product (described in Section 2.2.1) and the one Duoram uses. Unlike the standard Du-Atallah-styled dot-products, where the database shares and the standard-basis vector shares must be blinded and exchanged, our protocol only requires the blinded shares of the database to be traded. Cruicially, whereas previous protocols required linear communication

[3] The trusted source is for exposition only; we will remove it shortly. Also note that this formulation requires that $\mathtt{i}^*$ and M be known at DPF generation time, a requirement we will also remove.

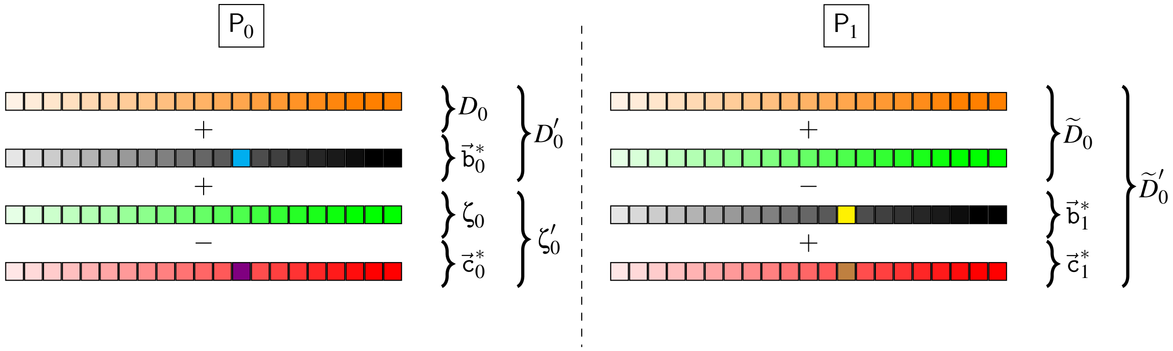


Figure 4: Duoram's RefreshBlinds operation. Since $\vec{\mathtt{b}}_0^* + \vec{\mathtt{b}}_1^* = \vec{\mathtt{c}}_0^* + \vec{\mathtt{c}}_1^* = \mathsf{M}\cdot\vec{\mathbf{e}}_{\mathtt{i}^*}$, we have $D_0' + \zeta_0' = \widetilde{D}_0'$. P_0 holds D_0 (depicted in orange) and updates it as $D_0 + \vec{\mathtt{b}}_0^*$ ($\vec{\mathtt{b}}_0^*$ depicted as black, with $\mathtt{i}^{*\text{th}}$ value depicted in blue). P_1 holding $\widetilde{D}_0 \leftarrow D_0 + \zeta_0$ (ζ_0 is depicted as green) "corrects" by subtracting $\vec{\mathtt{b}}_1^*$ (which is exactly the same as $-\vec{\mathtt{b}}_0^*$ except at index $\mathtt{i}^*$), which corrects all the indices but index $\mathtt{i}^*$. To correct that, P_0 subtracts $\vec{\mathtt{c}}_0^*$ (depicted as red) from ζ_0 and P_1 adds $\vec{\mathtt{c}}_1^*$ (depicted as red), to $\widetilde{D}_0 + \vec{\mathtt{b}}_1^*$.

when the database shares were updated, our innovation enables the parties to refresh their copies of each other's blinded databases with only logarithmic communication. Further, almost all of this communication can be done in the preprocessing phase, before the update locations or values are knows, with only a single word exchanged in the online phase. We achieve this at the cost of having three distinct shares of the same standard basis vector. Our Read protocol works as follows:

1. P_2 selects $\rho \in \{0,1\}^w$ uniformly at random and sends $\gamma_0 \leftarrow -\langle \zeta_0, \vec{\mathtt{c}}_1 \rangle + \rho$ to P_0 and $\gamma_1 \leftarrow -\langle \zeta_1, \vec{\mathtt{d}}_0 \rangle - \rho$ to P_1.
2. P_0 outputs $\mathsf{read}_0 \leftarrow \langle D_0 + \widetilde{D}_1, \vec{\mathtt{b}}_0 \rangle - \langle \zeta_0, \vec{\mathtt{c}}_0 - \vec{\mathtt{b}}_0 \rangle + \gamma_0$.
3. P_1 outputs $\mathsf{read}_1 \leftarrow \langle D_1 + \widetilde{D}_0, \vec{\mathtt{b}}_1 \rangle - \langle \zeta_1, \vec{\mathtt{d}}_1 - \vec{\mathtt{b}}_1 \rangle + \gamma_1$.

Lemma 1. *After the* Read *operation,* $\mathsf{read}_0 + \mathsf{read}_1 = D[\mathtt{i}^*]$.

The proof of Lemma 1 appears in Appendix B. The active participation of P_2 here is what makes Duoram a 3-party protocol and not a $(2+1)$-party protocol.

3.2.2 Update Operation

Our Update operation works as follows. For $b \in \{0,1\}$, P_b simply locally sets $D_b' \leftarrow D_b + \vec{\mathtt{b}}_b^*$.

Lemma 2. *After the* Update *operation, we have: (i)* $D_0'[\mathtt{i}] + D_1'[\mathtt{i}] = D[\mathtt{i}]$ *for all* $\mathtt{i} \neq \mathtt{i}^*$, *and (ii)* $D_0'[\mathtt{i}^*] + D_1'[\mathtt{i}^*] = D[\mathtt{i}^*] + \mathsf{M}$.

The proof of Lemma 2 follows from the definitions of DPFs. Our Update operation is (almost) the same as the one used in Floram. However, Duoram has an additional step to refresh the blinds to prepare itself for the next read operation. Critically, Duoram's RefreshBlinds requires only $O(1)$ words of communication, unlike Floram's $O(\sqrt{n})$ amortized cost. communication refresh operation, and this communication can even be sent in the same flows as already used for the Update protocol.

RefreshBlinds Performing the read operation in a Du-Atallah style has the following drawback. The blinded shares exchanged become "stale" after one Update operation and cannot be used for the next read operation. Suppose that P_0 holds $(D_0, \zeta_0, \widetilde{D}_1)$ and P_1 holds $(D_1, \zeta_1, \widetilde{D}_0)$, that they update the database with some value M, and that D_0' and D_1' are the updated database shares. We wish for each party to update their blinds ζ_b and their blinded versions $\widetilde{D}_{1-b}$ of the other party's database share so that $\widetilde{D}_b'$ (held by P_{1-b}) equals $D_b' + \zeta_b'$ (held by P_b). From the point of view of P_1, for example, we first observe that, the blinded shares, namely, $\widetilde{D}_0$ is wrong because P_0 updated D_0 by $\vec{\mathtt{b}}_0^*$. However, P_1 holds $\vec{\mathtt{b}}_1^*$, which is equal to $-\vec{\mathtt{b}}_0^*$ at every location, except at $\mathtt{i}^*$. Thus, P_1 can (almost) correct the blinded shares. In order to completely correct the blinded share $\widetilde{D}_0$, we exploit the fact that (i) $\vec{\mathtt{c}}_0^*[\mathtt{i}^*] + \vec{\mathtt{c}}_1^*[\mathtt{i}^*] = \vec{\mathtt{b}}_0^*[\mathtt{i}^*] + \vec{\mathtt{b}}_1^*[\mathtt{i}^*] = \mathsf{M}$, and (ii) only the $\mathtt{i}^*$ location needs to be corrected. Therefore, we use another pair of vectors which reconstruct to $\mathsf{M}\cdot\vec{\mathbf{e}}_{\mathtt{i}^*}$, namely $\vec{\mathtt{c}}_0^*$ and $\vec{\mathtt{c}}_1^*$. We have P_0 update the blind by $\vec{\mathtt{c}}_0^*$ and P_1 do one more update of $\widetilde{D}_0$ by $\vec{\mathtt{c}}_1^*$. Figure 4 gives a pictorial depiction of the RefreshBlinds from the point of view of P_0. More formally, we have the following:

1. P_0 and P_1 update the blinds as $\zeta_0' \leftarrow \zeta_0 - \vec{\mathtt{c}}_0^*$ and $\zeta_1' \leftarrow \zeta_1 - \vec{\mathtt{d}}_1^*$ respectively.
2. P_0 and P_1 also update the blinded shares they received as $\widetilde{D}_1' \leftarrow \widetilde{D}_1 + \vec{\mathtt{d}}_0^* - \vec{\mathtt{b}}_0^*$ and $\widetilde{D}_0' \leftarrow \widetilde{D}_0 + \vec{\mathtt{c}}_1^* - \vec{\mathtt{b}}_1^*$ respectively.
3. P_2 updates the blinds as $\zeta_0' \leftarrow \zeta_0 - \vec{\mathtt{c}}_0^*$ and $\zeta_1' \leftarrow \zeta_1 - \vec{\mathtt{d}}_1^*$.

Lemma 3. *For* $b \in \{0,1\}$, $D_b' + \zeta_b' = \widetilde{D}_b'$.

The proof of Lemma 3 appears in Appendix B. For the next Read operation Duoram uses $(\zeta_0', \widetilde{D}_1')$ and $(\zeta_1', \widetilde{D}_0')$ as the blinds and blinded shares, thus avoiding $O(\sqrt{n})$ amortized communication.

4 3-Party Duoram: The details

In this section, we fill in the gaps from the high-level discussion in Section 3.2. 3-party Duoram proceeds in two phases: the preprocessing phase and the online phase. The main idea of the preprocessing phase is that parties P_0 and P_1 receive DPFs with a random target point and a random target value before the computation begins or its inputs are known. Then they can "adjust" the DPFs accordingly to the required index and value. In other words, Duoram generates the DPFs before it has the knowledge of (i) *what* to write into the database, and (ii) *where* to write into the database, thus postponing those decisions to the online phase.

DPFs without the final correction word. Before we proceed with the descriptions of the preprocessing and online phases, we consider the concept of DPFs without the final correction word, $\mathcal{F}$. Such DPFs were used in Pirsona [27] and are critical to the Duoram Update protocol. In such DPFs, we replace the final correction word with $\mathcal{F}_0$ and $\mathcal{F}_1$ sent to P_0 and P_1 respectively, such that $\mathcal{F}_0 + \mathcal{F}_1 = -(\vec{v}_0[\texttt{i}^*] + \vec{v}_1[\texttt{i}^*])$, where $\vec{v}_0$ and $\vec{v}_1$ are evaluations of the DPF without $\mathcal{F}$. We denote the DPFs without the final correction word as $\bar{\mathtt{k}}_b = (\mathsf{s}_b^{()}, \mathsf{cw}^{(0)}, \ldots, \mathsf{cw}^{(h)}, \mathcal{F}_b)$.

4.1 Preprocessing Phase

The goal of the preprocessing phase is to achieve the DPF distribution in Section 3, without the trusted source.

4.1.1 A $(2+1)$-party protocol to generate DPFs

The key idea is to replace the trusted source of DPFs with $(2+1)$-party MPC protocol to create DPFs (without the final correction word) with random target point $\texttt{ri}$. The protocol begins with parties P_0 and P_1 selecting random indices $\widetilde{\texttt{ri}}_0$ and $\widetilde{\texttt{ri}}_1$ respectively. These values serve as XOR shares for the random value $\texttt{ri} = \widetilde{\texttt{ri}}_0 \oplus \widetilde{\texttt{ri}}_1$. The parties then use an MPC share conversion procedure to convert these XOR-shares to additive shares $\texttt{ri}_0$ and $\texttt{ri}_1$ respectively, such that $\texttt{ri}_0 + \texttt{ri}_1 = \texttt{ri}$. This conversion is needed because the DPF generation protocol takes as input XOR shares, while the remaining Duoram protocols use additive shares.

We run a $(2+1)$-party MPC protocol (due to Doerner and shelat [8]) on the XOR-shares $\widetilde{\texttt{ri}}_0$ and $\widetilde{\texttt{ri}}_1$, to generate DPFs (and their evaluation) without the final correction word at the location $\texttt{ri}$. The presentation of the protocol appears in Appendix C. The protocol results in P_b holding $(\mathring{\vec{v}}_b, \mathring{\vec{\mathbf{t}}}_b)$ such that, (i) $\mathring{\vec{v}}_0[\texttt{i}] \oplus \mathring{\vec{v}}_1[\texttt{i}] = 0$ for all $\texttt{i} \neq \texttt{ri}$, and (ii) $\mathring{\vec{\mathbf{t}}}_0 \oplus \mathring{\vec{\mathbf{t}}}_1 = \vec{\mathbf{e}}_{\texttt{ri}}$.

We interpret $\mathring{\vec{v}}_0$ as a word vector and call it $\vec{v}_0$, and interpret $\mathring{\vec{v}}_1$ as a word vector, negate each element (recall all word operations are mod 2^w), and call it $\vec{v}_1$. The final correction word shares can be locally computed by each party, and are defined as $\mathcal{F}_0 = -\sum_{\texttt{i}}(\vec{v}_0[\texttt{i}])$ and $\mathcal{F}_1 = -\sum_{\texttt{i}}(\vec{v}_1[\texttt{i}])$. Therefore, we have $\vec{v}_0[\texttt{ri}] + \vec{v}_1[\texttt{ri}] + \mathcal{F}_0 + \mathcal{F}_1 = 0$.

Duoram also requires additive shares of the flag vectors rather than XOR shares. We next describe a procedure that, given the above $\vec{v}_b$ and $\mathcal{F}_b$ values, converts the flag vectors $\mathring{\vec{\mathbf{t}}}_0$ and $\mathring{\vec{\mathbf{t}}}_1$ such that $\mathring{\vec{\mathbf{t}}}_0 \oplus \mathring{\vec{\mathbf{t}}}_1 = \vec{\mathbf{e}}_{\texttt{ri}}$, into word vectors $\vec{\mathbf{t}}_0$ and $\vec{\mathbf{t}}_1$ such that $\vec{\mathbf{t}}_0 + \vec{\mathbf{t}}_1 = \vec{\mathbf{e}}_{\texttt{ri}}$. (We will note later that this share conversion can even be skipped if the database were XOR-shared.)

Converting to additive shares. The share conversion algorithm begins with the two parties interpreting the flag vectors as words, and P_1 multiplies its word vector by -1. However, this leads to additive shares of $\pm 1 \cdot \vec{\mathbf{e}}_{\texttt{ri}}$. More specifically, if $\vec{\mathbf{t}}_0[\texttt{ri}] = 1$ and $\vec{\mathbf{t}}_1[\texttt{ri}] = 0$, these are additive shares of $\vec{\mathbf{e}}_{\texttt{ri}}$ as required, but if $\vec{\mathbf{t}}_0[\texttt{ri}] = 0$ and $\vec{\mathbf{t}}_1[\texttt{ri}] = 1$, these are additive shares of $-\vec{\mathbf{e}}_{\texttt{ri}}$. ($\vec{\mathbf{t}}_0$ and $\vec{\mathbf{t}}_1$ are the same for all other indices.) Our idea is that the parties compute the shares of the unknown sign, blind it with some random value, exchange them and reconstruct their sum, and multiply their word vectors with the sum. At this point, the word vectors add to 0 everywhere except at index $\texttt{ri}$ as required, but at $\texttt{ri}$, the sum is $1\pm$(the sum of the blinds), instead of just 1. The final step of the protocol fixes this offset. A formalization of the protocol appears in Appendix D.

The preprocessing phase for the Update protocol can be summarized as follows:

1. P_0 and P_1 use the $(2+1)$-MPC protocol to generate DPFs, namely, $(\bar{\mathtt{k}}_0^{(1)}, \bar{\mathtt{k}}_0^{(2)}, \bar{\mathtt{k}}_0^{(3)})$ and $(\bar{\mathtt{k}}_1^{(1)}, \bar{\mathtt{k}}_1^{(2)}, \bar{\mathtt{k}}_1^{(3)})$ respectively.
2. P_0 sends $\bar{\mathtt{k}}_0^{(2)}$ to P_2, and P_1 sends $\bar{\mathtt{k}}_1^{(3)}$ to P_2.
3. They convert the XOR-shared flags to additive shares.

At the end of the preprocessing phase, we have: (i) P_b holds $(\vec{v}_b^{(t)}, \vec{\mathbf{t}}_b^{(t)}, \mathcal{F}_b^{(t)}, \texttt{ri}_b)$, for $b \in \{0,1\}$, $t \in \{1,2,3\}$, and (ii) The auxiliary party P_2 holds $(\vec{v}_0^{(2)}, \vec{\mathbf{t}}_0^{(2)})$, $(\vec{v}_1^{(3)}, \vec{\mathbf{t}}_1^{(3)})$.

The preprocessing for the Read protocol almost the same, requiring three DPFs. For the Read protocol however, the parties do not need to hold any $\vec{v}_b$ or any final correction word.

4.2 3-Party Online Phase

During the multi-party computation, P_0 and P_1 will want to read or update some index $\texttt{i}^*$ of the shared database, where the index itself is shared between them. From the preprocessing phase, they already have shares of a random standard basis vector $\vec{\mathbf{e}}_{\texttt{ri}}$ along with shares of $\texttt{ri}$. The parties first use the *cyclic shift* protocol to shift shares of $\vec{\mathbf{e}}_{\texttt{ri}}$ to become shares of $\vec{\mathbf{e}}_{\texttt{i}^*}$.

4.2.1 Adjusting the random DPFs

Cyclic Shifts: Postponing the decision of *where* to write. The 3-party online phase begins with a cyclic shift protocol

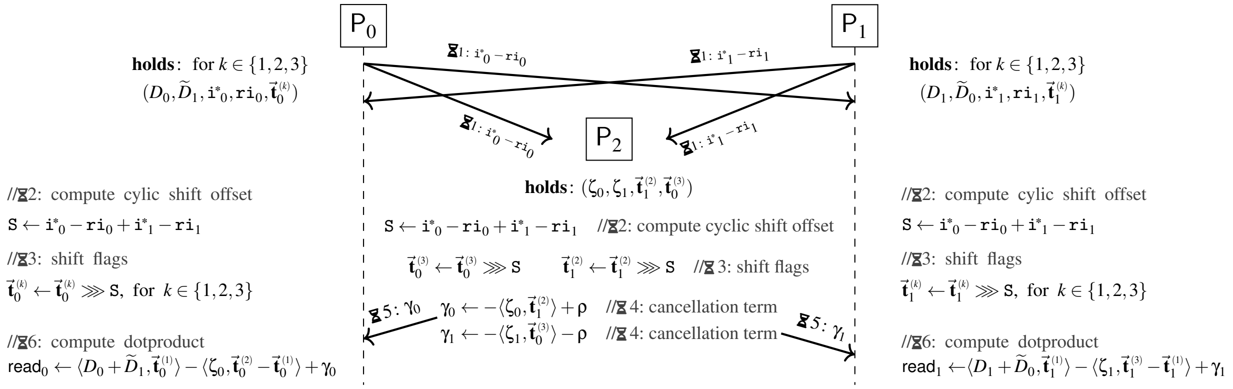

P_0
P_1
P_2
holds: for $k \in \{1,2,3\}$ $(D_0, \widetilde{D}_1, \mathtt{i^*}_0, \mathtt{ri}_0, \vec{\mathbf{t}}_0^{(k)})$
holds: for $k \in \{1,2,3\}$ $(D_1, \widetilde{D}_0, \mathtt{i^*}_1, \mathtt{ri}_1, \vec{\mathbf{t}}_1^{(k)})$
⧗1: $\mathtt{i^*}_0 - \mathtt{ri}_0$
⧗1: $\mathtt{i^*}_1 - \mathtt{ri}_1$
⧗1: $\mathtt{i^*}_0 - \mathtt{ri}_0$
⧗1: $\mathtt{i^*}_1 - \mathtt{ri}_1$
holds: $(\zeta_0, \zeta_1, \vec{\mathbf{t}}_1^{(2)}, \vec{\mathbf{t}}_0^{(3)})$
//⧗2: compute cylic shift offset
$\mathtt{S} \leftarrow \mathtt{i^*}_0 - \mathtt{ri}_0 + \mathtt{i^*}_1 - \mathtt{ri}_1$
//⧗3: shift flags
$\vec{\mathbf{t}}_0^{(k)} \leftarrow \vec{\mathbf{t}}_0^{(k)} \ggg \mathtt{S}$, for $k \in \{1,2,3\}$
//⧗6: compute dotproduct
$\mathsf{read}_0 \leftarrow \langle D_0 + \widetilde{D}_1, \vec{\mathbf{t}}_0^{(1)} \rangle - \langle \zeta_0, \vec{\mathbf{t}}_0^{(2)} - \vec{\mathbf{t}}_0^{(1)} \rangle + \gamma_0$
$\mathtt{S} \leftarrow \mathtt{i^*}_0 - \mathtt{ri}_0 + \mathtt{i^*}_1 - \mathtt{ri}_1$ //⧗2: compute cyclic shift offset
$\vec{\mathbf{t}}_0^{(3)} \leftarrow \vec{\mathbf{t}}_0^{(3)} \ggg \mathtt{S}$ $\vec{\mathbf{t}}_1^{(2)} \leftarrow \vec{\mathbf{t}}_1^{(2)} \ggg \mathtt{S}$ //⧗3: shift flags
⧗5: γ_0
$\gamma_0 \leftarrow -\langle \zeta_0, \vec{\mathbf{t}}_1^{(2)} \rangle + \rho$ //⧗4: cancellation term
$\gamma_1 \leftarrow -\langle \zeta_1, \vec{\mathbf{t}}_0^{(3)} \rangle - \rho$ //⧗4: cancellation term
⧗5: γ_1
//⧗2: compute cylic shift offset
$\mathtt{S} \leftarrow \mathtt{i^*}_0 - \mathtt{ri}_0 + \mathtt{i^*}_1 - \mathtt{ri}_1$
//⧗3: shift flags
$\vec{\mathbf{t}}_1^{(k)} \leftarrow \vec{\mathbf{t}}_1^{(k)} \ggg \mathtt{S}$, for $k \in \{1,2,3\}$
//⧗6: compute dotproduct
$\mathsf{read}_1 \leftarrow \langle D_1 + \widetilde{D}_0, \vec{\mathbf{t}}_1^{(1)} \rangle - \langle \zeta_1, \vec{\mathbf{t}}_1^{(3)} - \vec{\mathbf{t}}_1^{(1)} \rangle + \gamma_1$

Figure 5: Online phase of the 3-party READ Protocol. Protocol to do a read operation at the index $\mathtt{i^*} = \mathtt{i^*}_0 + \mathtt{i^*}_1$. For $k \in \{1,2,3\}$, $\vec{\mathbf{t}}_0^{(k)} + \vec{\mathbf{t}}_1^{(k)} = \vec{\mathbf{e}}_{\mathtt{ri}}$. The numbers next to ⧗ is the time stamp when the action is performed.

that adjusts the shares of the standard basis vector at a random location $\mathtt{ri}$ to the shares of the standard basis vector at the target index $\mathtt{i^*}$. P_0 and P_1 hold $\mathtt{i^*}_0$ and $\mathtt{i^*}_1$ respectively, such that $\mathtt{i^*} = \mathtt{i^*}_0 + \mathtt{i^*}_1$. They also hold $(\mathtt{ri}_0, \vec{v}_0')$ and $(\mathtt{ri}_1, \vec{v}_1')$ such that $\mathtt{ri} = \mathtt{ri}_0 + \mathtt{ri}_1$, and $\vec{v}_0' + \vec{v}_1' = \vec{\mathbf{e}}_{\mathtt{ri}}$. Their goal is to get vectors $\vec{v}_0$ and $\vec{v}_1$ such that $\vec{v}_0 + \vec{v}_1 = \vec{\mathbf{e}}_{\mathtt{i^*}}$. They exchange $\mathtt{i^*}_b - \mathtt{ri}_b$, reconstruct $(\mathtt{i^*}_0 + \mathtt{i^*}_1 - \mathtt{ri}_0 - \mathtt{ri}_1)$, and cyclic right shift $\vec{v}_b'$ by $(\mathtt{i^*}_0 + \mathtt{i^*}_1 - \mathtt{ri}_0 - \mathtt{ri}_1)$. Similarly, P_2 receives $\mathtt{i^*}_b - \mathtt{ri}_b$ from P_b and can also compute the required offset $(\mathtt{i^*}_0 + \mathtt{i^*}_1 - \mathtt{ri}_0 - \mathtt{ri}_1)$. We denote a cyclic right shift by $\ggg$.

In this protocol, which is key to being able to move the DPF generation to preprocessing, it is required that index shares be additive. For consistency, DUORAM keeps the database also as additive shares, which also simplifies linked data structures where addresses are stored in the database. For computations without this need, however, the database can be stored with XOR shares in DUORAM, removing the need for the share conversion procedure described in Section 4.1.1.

4.2.2 READ Protocol

The READ protocol does not need any other details to be filled in from our high-level discussion. After performing the cyclic shift protocol to move the target point from $\mathtt{ri}$ to $\mathtt{i^*}$, we use the 3-party protocol in Section 3.2.1. Figure 5 describes the 3-party online phase of the READ protocol.

4.2.3 UPDATE Protocol

Postponing the decision of *what* to write. We have the following situation. P_0 and P_1 hold D_0 and D_1, the shares of the database D. The two parties also hold (i) M_0 and M_1, the shares of the target value, and (ii) $(\vec{v}_0, \vec{v}_1)$, the evaluations of $\bar{\mathtt{k}}_0$ and $\bar{\mathtt{k}}_1$, DPFs at $\mathtt{i^*}$. Their goal is to add $\mathsf{M} = \mathsf{M}_0 + \mathsf{M}_1$ to the value at the target index $\mathtt{i^*}$ of the database D. The idea is that the parties P_b exchange $\mathsf{M}_b + \mathcal{F}_b$ to reconstruct $\mathsf{M} - (\vec{v}_0[\mathtt{i^*}] + \vec{v}_1[\mathtt{i^*}])$. The reconstruction serves as the new final correction word to write M in the desired location.

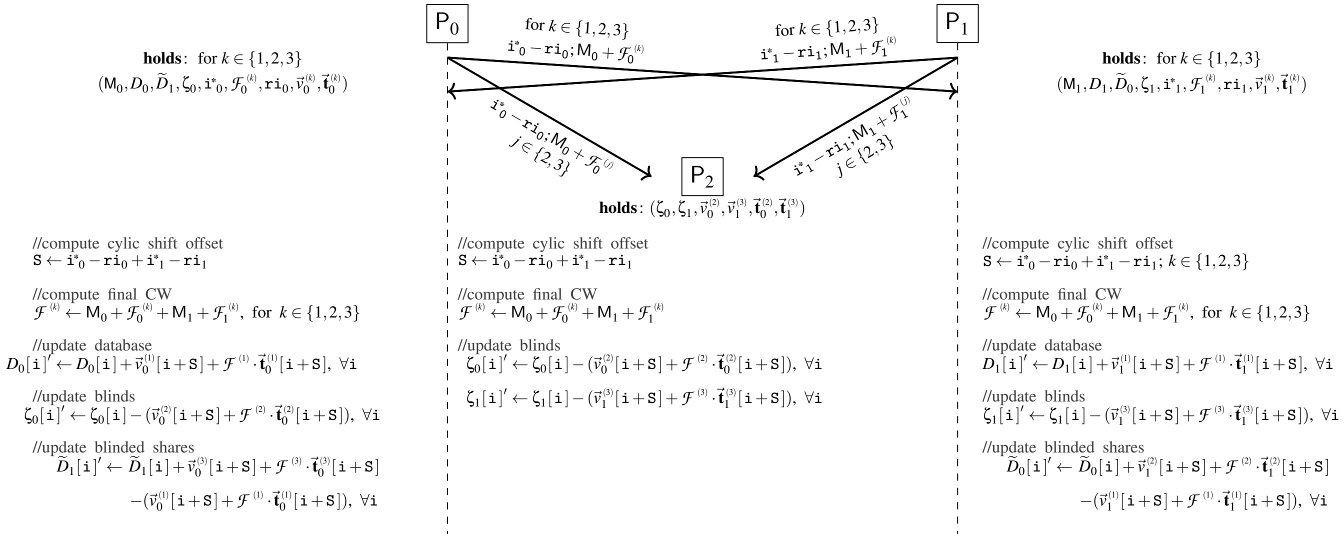
We next formally describe the operations described in Figure 4, which shows P_0 updating its database share and blind, and P_1 updating its blinded copy of P_0's database share. The symmetric updates must also be performed; we present the complete online phase of the update protocol, including both sets of updates, in Figure 6.

1. P_0 and P_1 exchange $((\mathsf{M}_0 + \mathcal{F}_0^{(1)}), (\mathsf{M}_0 + \mathcal{F}_0^{(2)}))$ and $((\mathsf{M}_1 + \mathcal{F}_1^{(1)}), (\mathsf{M}_1 + \mathcal{F}_1^{(2)}))$ respectively.
2. They reconstruct $\mathcal{F}^{(1)} \leftarrow (\mathsf{M}_0 + \mathcal{F}_0^{(1)} + \mathsf{M}_1 + \mathcal{F}_1^{(1)})$ and $\mathcal{F}^{(2)} \leftarrow (\mathsf{M}_0 + \mathcal{F}_0^{(2)} + \mathsf{M}_1 + \mathcal{F}_1^{(2)})$.
3. P_0 updates $D_0[i]' \leftarrow D_0[\mathtt{i}] + \left(\vec{v}_0^{(1)}[i] + (\vec{\mathbf{t}}_0^{(1)}[\mathtt{i}] \cdot \mathcal{F}^{(1)})\right)$.
4. P_0 updates $\zeta_0[i]' \leftarrow \zeta_0[\mathtt{i}] - \left(\vec{v}_0^{(2)}[\mathtt{i}] + (\vec{\mathbf{t}}_0^{(2)}[\mathtt{i}] \cdot \mathcal{F}^{(2)})\right)$.
5. P_1 updates $\widetilde{D}_0[\mathtt{i}]' \leftarrow \widetilde{D}_0[\mathtt{i}] + \vec{v}_1^{(1)}[\mathtt{i}] + \mathcal{F}^{(1)} \cdot \vec{\mathbf{t}}_1^{(1)}[\mathtt{i}] - (\vec{v}_1^{(2)}[\mathtt{i}] + \mathcal{F}^{(2)} \cdot \vec{\mathbf{t}}_1^{(2)}[\mathtt{i}])$.

In the above protocol, Steps 1 and 2 are used to compute the final correction words. Step 3 is used to update the database, Step 4 is used to update the blinds, and Step 5 is used to update the blinded shares. P_2 also updates their blinds in a similar manner.

5 2-Party DUORAM

This section presents the 2-party instantiation of DUORAM. The update protocol is (nearly) the same as the one in 3-party DUORAM, with the number of communication words in the online phase independent of the database size. The 2-party read protocol uses a single-server Computational Symmetric PIR (CSPIR) protocol, resulting in communication logarithmic in the database size (beating FLORAM's $O(\sqrt{n})$), but with more local computation. Our experiments in the next section show that it can be overall faster than FLORAM for typical network configurations.



P_0
P_1
P_2
holds: for $k \in \{1,2,3\}$ $(\mathsf{M}_0, D_0, \widetilde{D}_1, \zeta_0, \mathtt{i^*}_0, \mathcal{F}_0^{(k)}, \mathtt{ri}_0, \vec{v}_0^{(k)}, \vec{\mathbf{t}}_0^{(k)})$
for $k \in \{1,2,3\}$ $\mathtt{i^*}_0 - \mathtt{ri}_0; \mathsf{M}_0 + \mathcal{F}_0^{(k)}$
for $k \in \{1,2,3\}$ $\mathtt{i^*}_1 - \mathtt{ri}_1; \mathsf{M}_1 + \mathcal{F}_1^{(k)}$
holds: for $k \in \{1,2,3\}$ $(\mathsf{M}_1, D_1, \widetilde{D}_0, \zeta_1, \mathtt{i^*}_1, \mathcal{F}_1^{(k)}, \mathtt{ri}_1, \vec{v}_1^{(k)}, \vec{\mathbf{t}}_1^{(k)})$
$\mathtt{i^*}_0 - \mathtt{ri}_0; \mathsf{M}_0 + \mathcal{F}_0^{(j)}$ $j \in \{2,3\}$
$\mathtt{i^*}_1 - \mathtt{ri}_1; \mathsf{M}_1 + \mathcal{F}_1^{(j)}$ $j \in \{2,3\}$
holds: $(\zeta_0, \zeta_1, \vec{v}_0^{(2)}, \vec{v}_1^{(3)}, \vec{\mathbf{t}}_0^{(2)}, \vec{\mathbf{t}}_1^{(3)})$
//compute cylic shift offset
$\mathtt{S} \leftarrow \mathtt{i^*}_0 - \mathtt{ri}_0 + \mathtt{i^*}_1 - \mathtt{ri}_1$
//compute final CW
$\mathcal{F}^{(k)} \leftarrow \mathsf{M}_0 + \mathcal{F}_0^{(k)} + \mathsf{M}_1 + \mathcal{F}_1^{(k)}$, for $k \in \{1,2,3\}$
//update database
$D_0[\mathtt{i}]' \leftarrow D_0[\mathtt{i}] + \vec{v}_0^{(1)}[\mathtt{i}+\mathtt{S}] + \mathcal{F}^{(1)} \cdot \vec{\mathbf{t}}_0^{(1)}[\mathtt{i}+\mathtt{S}]$, $\forall \mathtt{i}$
//update blinds
$\zeta_0[\mathtt{i}]' \leftarrow \zeta_0[\mathtt{i}] - (\vec{v}_0^{(2)}[\mathtt{i}+\mathtt{S}] + \mathcal{F}^{(2)} \cdot \vec{\mathbf{t}}_0^{(2)}[\mathtt{i}+\mathtt{S}])$, $\forall \mathtt{i}$
//update blinded shares
$\widetilde{D}_1[\mathtt{i}]' \leftarrow \widetilde{D}_1[\mathtt{i}] + \vec{v}_0^{(3)}[\mathtt{i}+\mathtt{S}] + \mathcal{F}^{(3)} \cdot \vec{\mathbf{t}}_0^{(3)}[\mathtt{i}+\mathtt{S}]$
$-(\vec{v}_0^{(1)}[\mathtt{i}+\mathtt{S}] + \mathcal{F}^{(1)} \cdot \vec{\mathbf{t}}_0^{(1)}[\mathtt{i}+\mathtt{S}])$, $\forall \mathtt{i}$
//compute cylic shift offset
$\mathtt{S} \leftarrow \mathtt{i^*}_0 - \mathtt{ri}_0 + \mathtt{i^*}_1 - \mathtt{ri}_1$
//compute final CW
$\mathcal{F}^{(k)} \leftarrow \mathsf{M}_0 + \mathcal{F}_0^{(k)} + \mathsf{M}_1 + \mathcal{F}_1^{(k)}$
//update blinds
$\zeta_0[\mathtt{i}]' \leftarrow \zeta_0[\mathtt{i}] - (\vec{v}_0^{(2)}[\mathtt{i}+\mathtt{S}] + \mathcal{F}^{(2)} \cdot \vec{\mathbf{t}}_0^{(2)}[\mathtt{i}+\mathtt{S}])$, $\forall \mathtt{i}$
$\zeta_1[\mathtt{i}]' \leftarrow \zeta_1[\mathtt{i}] - (\vec{v}_1^{(3)}[\mathtt{i}+\mathtt{S}] + \mathcal{F}^{(3)} \cdot \vec{\mathbf{t}}_1^{(3)}[\mathtt{i}+\mathtt{S}])$, $\forall \mathtt{i}$
//compute cylic shift offset
$\mathtt{S} \leftarrow \mathtt{i^*}_0 - \mathtt{ri}_0 + \mathtt{i^*}_1 - \mathtt{ri}_1$; $k \in \{1,2,3\}$
//compute final CW
$\mathcal{F}^{(k)} \leftarrow \mathsf{M}_0 + \mathcal{F}_0^{(k)} + \mathsf{M}_1 + \mathcal{F}_1^{(k)}$, for $k \in \{1,2,3\}$
//update database
$D_1[\mathtt{i}]' \leftarrow D_1[\mathtt{i}] + \vec{v}_1^{(1)}[\mathtt{i}+\mathtt{S}] + \mathcal{F}^{(1)} \cdot \vec{\mathbf{t}}_1^{(1)}[\mathtt{i}+\mathtt{S}]$, $\forall \mathtt{i}$
//update blinds
$\zeta_1[\mathtt{i}]' \leftarrow \zeta_1[\mathtt{i}] - (\vec{v}_1^{(3)}[\mathtt{i}+\mathtt{S}] + \mathcal{F}^{(3)} \cdot \vec{\mathbf{t}}_1^{(3)}[\mathtt{i}+\mathtt{S}])$, $\forall \mathtt{i}$
//update blinded shares
$\widetilde{D}_0[\mathtt{i}]' \leftarrow \widetilde{D}_0[\mathtt{i}] + \vec{v}_1^{(2)}[\mathtt{i}+\mathtt{S}] + \mathcal{F}^{(2)} \cdot \vec{\mathbf{t}}_1^{(2)}[\mathtt{i}+\mathtt{S}]$
$-(\vec{v}_1^{(1)}[\mathtt{i}+\mathtt{S}] + \mathcal{F}^{(1)} \cdot \vec{\mathbf{t}}_1^{(1)}[\mathtt{i}+\mathtt{S}])$, $\forall \mathtt{i}$

Figure 6: Online phase of the 3-party UPDATE Protocol. Protocol to add the value $\mathsf{M} = \mathsf{M}_0 + \mathsf{M}_1$ to the index $\mathtt{i^*} = \mathtt{i^*}_0 + \mathtt{i^*}_1$ in the database $D = D_0 + D_1$. For $k \in \{1,2,3\}$, we have, $\vec{\mathbf{t}}_0^{(k)} + \vec{\mathbf{t}}_1^{(k)} = \vec{\mathbf{e}}_{\mathtt{ri}}$ and $\vec{v}_0^{(k)} + (\vec{\mathbf{t}}_0 \cdot \mathcal{F}^{(k)}) + \vec{v}_1^{(k)} + (\vec{\mathbf{t}}_1 \cdot \mathcal{F}^{(k)}) = \mathbf{0}$. At the end of the protocol, for $b \in \{0,1\}$, P_b gets $(\widetilde{D}'_{1-b}, D'_b, \zeta'_b)$, such that $\widetilde{D}'_b = D'_b + \zeta'_b$. The next READ operation uses ζ'_b and $\widetilde{D}'_b$ as blinding factors and blinded shares. For the next UPDATE operation, new $(\vec{v}_b^{(k)}, \vec{\mathbf{t}}_b^{(k)})$ are received. All array subscripts are taken mod n.

Like in the 3-party setup, the 2-party protocol can be divided into an online and preprocessing phase. The idea once again is to generate and exchange CSPIR queries at a random location in the preprocessing phase and then use the cyclic shift protocol to correct it. Similarly, for an update operation, we can generate random DPFs at a random location with a random target value.

2-Party READ. The read operation in the 2-party DUORAM relies on Computational Symmetric PIR (CSPIR), which works as follows. The parties have precomputed and exchanged CSPIR queries for lookups at random indices $\mathtt{ri}_0$ and $\mathtt{ri}_1$ respectively. They also have shares $\mathtt{i^*}_0$ and $\mathtt{i^*}_1$ of the target index $\mathtt{i^*}$. For $b \in \{0,1\}$:

1. P_b sends $(\mathtt{i^*}_b - \mathtt{ri}_b)$ to P_{1-b}.
2. P_b blinds each of the elements of D_b with a random value $\mathsf{r}_b \in \{0,1\}^w$, and rotates the resulting blinded vector by $\mathtt{i^*}_b + (\mathtt{i^*}_{1-b} - \mathtt{ri}_{1-b})$. In other words, P_b computes $D'_b[\mathtt{i}] \leftarrow (D_b[\mathtt{i} + \mathtt{i^*}_b + (\mathtt{i^*}_{1-b} - \mathtt{ri}_{1-b})] + \mathsf{r}_b)$.
3. P_{1-b} performs P_b's preprepared CSPIR query with index $\mathtt{ri}_b$ on D'_{1-b} and sends the result to P_b, who recovers $c_b = D_{1-b}[\mathtt{i^*}] + \mathsf{r}_{1-b}$.
4. P_b outputs: $\mathsf{read}_b \leftarrow c_b - \mathsf{r}_b$.

Lemma 4. *After the 2-party* READ $\mathsf{read}_0 + \mathsf{read}_1 = D[\mathtt{i^*}]$.

The proof of Lemma 4 can be found in Appendix B. Our implementation uses the SPIRAL CPIR protocol by Menon and Wu [23]. We note that CPIR protocols like SPIRAL are extremely parallelizable with more hardware. The protocol, however, cannot be used as-is, as it is not symmetric; that is, the client may learn more than just one database element. Therefore, we augment the SPIRAL protocol into a SPIR protocol using the generic OT-based PIR-to-SPIR transform by Naor and Pinkas [24].

2-Party UPDATE. Observe that since we do the reading via CSPIR, there is no notion of blinds or blinded shares; thus, the REFRESHBLINDS operation is no longer needed. The online phase of the 2-party UPDATE protocol is then the same as the online phase of the 3-party DUORAM, but without the additional REFRESHBLINDS operation. Thus, somewhat counter-intuitively, the online phase of the 2-party DUORAM is cheaper than the online phase of the 3-party DUORAM. However, the preprocessing phase differs because the multiplicative triples, rather than being generated via an auxiliary party, are generated via OT, though we only need to precompute one DPF per UPDATE operation instead of three DPFs per UPDATE and READ.

6 Evaluation

We next evaluate the performance of DUORAM on different ORAM operations. We classify the READ operations as either *dependent* or *independent* reads for our evaluation. We call a block of k READ operations *dependent* if the target index for each read is known only after the completion of the previous read. This notion models following pointers in a

Table 1: Comparing computation, bandwidth, and the number of messages sent across various DUORAM operations for a database containing n words of size w. The gray color represents the preprocessing cost. FLORAM is shown for comparison. In this table, we assume $w \geq \lg n$.

	3P-DUORAM			2P-DUORAM			FLORAM		
Operation	Computation	Bandwidth	Messages	Computation	Bandwidth	Messages	Computation	Bandwidth	Messages
k `Ind Reads`	$O(k{\cdot}n)+O(k{\cdot}n)$	$O(k{\cdot}w{\cdot}\lg n)+2{\cdot}k{\cdot}w$	$O(\lg n)+2$	$O(k{\cdot}n)+O(k{\cdot}n)$	$O(k{\cdot}w{\cdot}\lg n)+O(k{\cdot}w)$	$O(1)+2$	$O(k{\cdot}n)$	$O(k{\cdot}w{\cdot}\lg n)$	$O(k)$
k `Dep Reads`	$O(k{\cdot}n)+O(k{\cdot}w)$	$O(k{\cdot}w{\cdot}\lg n)+2{\cdot}k{\cdot}w$	$O(\lg n)+2{\cdot}k$	$O(k{\cdot}n)+O(k{\cdot}n)$	$O(k{\cdot}w{\cdot}\lg n)+O(k{\cdot}w)$	$O(1)+2{\cdot}k$	$O(k{\cdot}n)$	$O(k{\cdot}w{\cdot}\lg n)$	$O(k)$
k `Writes`	$O(k{\cdot}n)+O(k{\cdot}n)$	$O(k{\cdot}w{\cdot}\lg n)+9{\cdot}k{\cdot}w$	$O(\lg n)+3{\cdot}k$	$O(k{\cdot}n)+O(k{\cdot}n)$	$O(k{\cdot}w{\cdot}\lg n)+O(k{\cdot}w)$	$O(\lg n)+3{\cdot}k$	$O(k{\cdot}n)$	$O(k{\cdot}w{\cdot}\sqrt{n})$	$O(k)$
`Interleaved`	$O(k{\cdot}n)+O(k{\cdot}n)$	$O(k{\cdot}w{\cdot}\lg n)+11{\cdot}k{\cdot}w$	$O(\lg n)+5{\cdot}k$	$O(k{\cdot}n)+O(k{\cdot}n)$	$O(k{\cdot}w{\cdot}\lg n)+O(k{\cdot}w)$	$O(\lg n)+5{\cdot}k$	$O(k{\cdot}n)$	$O(k{\cdot}w{\cdot}\sqrt{n})$	$O(k)$

linked data structure or traversing a binary tree, for example. We call a block of k READ operations *independent* if the k target indices are known in advance of performing any of the reads. However, we do not make this distinction for the WRITE operations, which are always considered dependent. A READ operation followed by a WRITE operation is called an interleaved operation. Thus, k interleaved operations are k read and k write operations interleaved with one another. Note that an interleaved operation actually involves two reads, since a write operation is a read operation followed by an update operation.

6.1 Analytical Evaluation

Before we present our experimental results, we give an analytical accounting of computation and communication costs of the DUORAM variants. We summarize the costs in Table 1.[4] Observe that the amortized online communication cost of 3P-DUORAM is constant. Another important thing to note is that the number of messages sent for k independent reads is independent of k.

6.2 Experimental Evaluation

Experimental Setup We implemented and benchmarked DUORAM. We wrote a proof-of-concept reference implementation in C++. Our implementation uses Boost.Asio v1.18.1 for asynchronous communication.[5] We ran the parties in separate docker containers and simulated network parameters with `tc qdisc add dev eth0 root netem delay Xms rate Ymbit`, to set the latency to `X` ms, and restrict the rate to `Y` Mbit/s. We implemented the PRGs with AES.

6.2.1 Head-to-Head Comparison with FLORAM

In this section, we do a head-to-head comparison of FLORAM[6] with DUORAM. The following paragraphs we compare the (i) wall-clock time, and (ii) bandwidth consumption. The standard latency we use is 30 ms.[7] We vary network parameters and database sizes, and compare, 2P- and 3P-DUORAM with FLORAM for different ORAM operations. Specifically, we make our comparisons under the following conditions: (i) varying the database size while keeping the latency and throughput constant at 30 ms and 100 Mbit/second, respectively, (ii) varying the network latency while keeping the number of items at 2^{20} and the throughput at 100 Mbit/second, and (iii) varying the throughput while keeping the number of items constant at 2^{20} and the network latency at 30 ms.

Figure 7 compares the performance of DUORAM with FLORAM for doing *interleaved* operations. The comparative behaviours of DUORAM and FLORAM for *read* and *write* operations are very similar; those plots can be found in Figures 10 and 11 in Appendix E. For our standard network settings, we see that 2-party DUORAM is faster than FLORAM until the database size reaches around 2^{24} items. On the other hand, 3-party DUORAM consistently performs better than FLORAM for all database sizes. Observe that as we increase the latency, the performance of FLORAM worsens much more than DUORAM's performance owing to the additional rounds in FLORAM. Decreasing the bandwidth capacity has a minimal impact on the performance of DUORAM because it sends so much less data, as can be seen in Figure 8. However, observe that for all the ORAM operations that we evaluate, FLORAM experiences a significant dip in performance as the bandwidth capacity is throttled. The dip is more significant when we are doing interleaved operations, as FLORAM's interleaved operations require a refresh after every $\sqrt{n}/8$ iterations.

Figure 8 compares the bandwidth consumption between DUORAM and FLORAM. We must point out the plot of 2P-DUORAM, includes a one-time setup cost (per client software installation) of 10 MiB. We notice that the difference between FLORAM and DUORAM is the starkest in the interleaved operations case. This is because FLORAM requires a $O(n)$ communication cost before every $\sqrt{n}/8$ iterations. Most of FLORAM's bandwidth consumption in the case of dependent READ and WRITE operations comes in the initialization phase. In the case of the WRITE operation, the online phase

[4] Table 1 in the FLORAM paper [8] says it requires only $O(1)$ messages to be sent per access, and that the is value we report in the table. However, Figure 6 in that same work shows that it requires $\Omega(\lg n)$ rounds of communication.

[5] Our source code is available at https://git-crysp.uwaterloo.ca/avadapal/duoram.

[6] Code retrieved from https://gitlab.com/neucrypt/floram/.

[7] A value chosen from the low end of one-way latency values from https://www.cloudping.co/grid. Note that low latencies benefit FLORAM much more than DUORAM, as we will soon see.

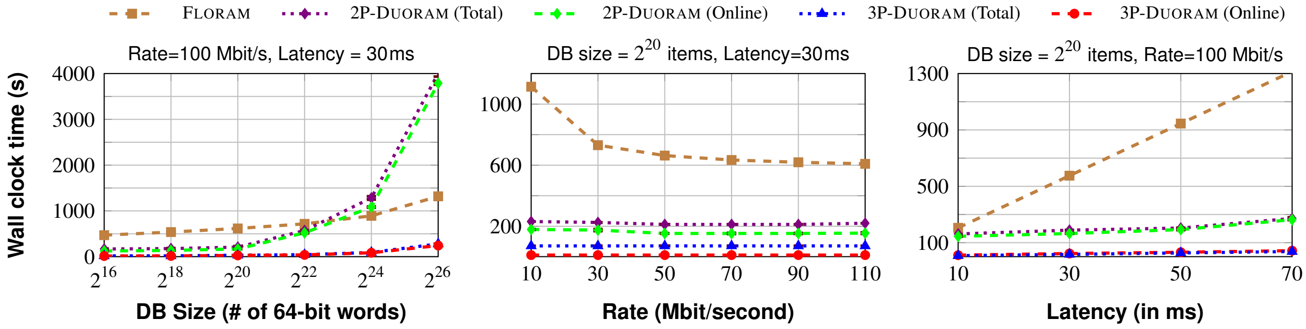


Figure 7: Comparing FLORAM and DUORAM to do 128 interleaved operations for different parameters of *database size*, *latency*, and *bandwidth* on databases with 8-byte words. (The error bars are too small and thus not visible.)

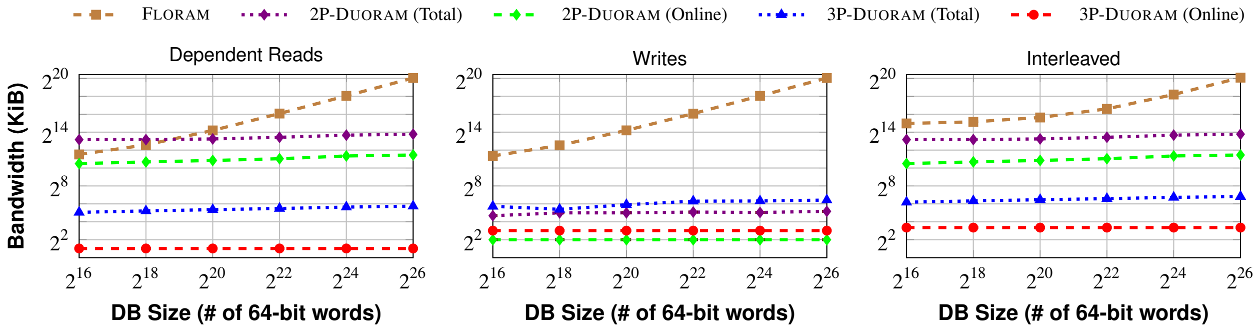


Figure 8: Comparing bandwidth consumption to do 128 dependent read, 128 write, and 128 interleaved ORAM operations in DUORAM and FLORAM; the y-axis is log-scaled.

of 2-party DUORAM performs slightly better than the 3-party version because it does not need to run REFRESHBLINDS. To illustrate and highlight the low bandwidth requirements of DUORAM, we reduced the bandwidth capacity to as low as 1 Mbit per second and set the latency to 100 ms. Even under these extreme network settings, the performance of DUORAM does not suffer much. For example, while 2-party DUORAM took about 10 seconds to do one read operation (including both preprocessing and online time) on a database of 2^{20} items, FLORAM took over 1.5 hours. For 2^{25} items, DUORAM took about 30 seconds, while FLORAM did not finish running after more than 10 hours. Table 3 in Appendix E gives the detailed results of this experiment.

6.2.2 Scaling DUORAM

In the following experiments, we examine how DUORAM scales as we increase the number of cores. Like in the previous experiments, we set to use the network parameters as 30 ms of latency and 100 Mbit/s bandwidth. Figure 9 shows how the performance of DUORAM and FLORAM vary as we increase the number of cores. We observe that, as we increase the number of cores, 2-party DUORAM sees a significant improvement in performance, while the same improvement is not observed in FLORAM's performance. This is because the bottlenecks for FLORAM and DUORAM are different. Bandwidth is the bottleneck for FLORAM; thus, increasing the parallelism does not affect FLORAM's performance in the restricted bandwidth setting. The upshot of this finding is that it is much more expensive to scale the performance of FLORAM as compared to DUORAM, as buying extra bandwidth is more expensive. For instance, the current rates[8] for a long-term Amazon EC2 instance: \$0.0195/CPU-hour and \$0.09/GB of outbound traffic.[9] Table 2 compares the cost in USD to do ORAM operations in DUORAM and FLORAM. All the costs in costs in Table 2 in micro-dollars, denoted as μ\$ (1 micro-dollar = 10^{-6} USD).

7 Related Work

In this section, we do a brief survey of the previous work leading in this direction. Goldreich and Ostrovsky [14] first introduced ORAMs in a general client-server context. Their work was the so-called square-root ORAM, i.e., adding a *square root* overhead. The problem they considered was for a client to perform some computation over a memory of size n held by some untrusted server, while hiding its access patterns to the memory. For the next two decades, there have several works [2,6,7,15,16,20,25,26,30,31] addressing this problem

[8] https://aws.amazon.com/ec2/pricing/on-demand/

[9] These are the same values used by SPRIAL [23] for their dollar costs.

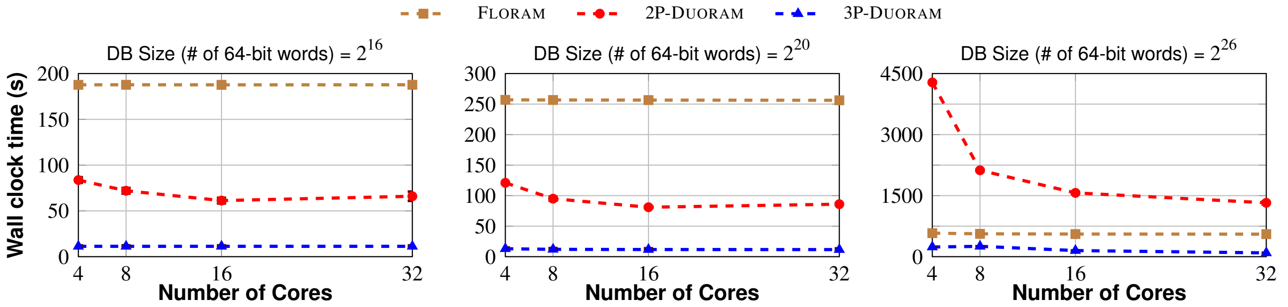


Figure 9: Comparing the performance of DUORAM and FLORAM to do 128 read operations by varying the number of cores being used for various database sizes. (The error bars are too small and thus not visible.)

Table 2: Comparing monetary costs of DUORAM and FLORAM while setting the throughput to be 100 Mbit/s, and latency to 30 ms.

	2P-DUORAM		3P-DUORAM		FLORAM	
operation	CPU Cost	Bandwidth Cost	CPU Cost	Bandwidth Cost	CPU Cost	Bandwidth Cost
128 `Reads`	μ\$3800	μ\$1200	μ\$3000	μ\$5	μ\$6400	μ\$20000
128 `Writes`	μ\$5100	μ\$5	μ\$4700	μ\$5	μ\$8000	μ\$20000
128 `Interleaved`	μ\$8900	μ\$1200	μ\$7700	μ\$5	μ\$13000	μ\$38000

with the goal of reducing communication overhead between the client and the server. ORAM can be used in the context of secure computation. In such a setting, the client operations are implemented as a circuit. Wang, Chan, and Shi's work [29] introduced a tree-based ORAM that optimizes the circuit size. Gentry, Goldman, Halevi, Julia, Raykova, and Wichs [11] improve upon the Tree ORAM. There have been other notable works using Tree ORAM. For example, Gordon, Katz, and Wang [17] in 2018 presented a 2-server ORAM combining Tree ORAM with an extension of a 2-server PIR protocol. There have been other notable works in this direction. Jarecki and Wei [19] present a 3-party MPC ORAM. Faber, Jarecki, Kentros, and Wei [10] show a 3-party ORAM, a version of Secure Computation protocol, a variant of the binary tree ORAM by Shi et al [25].

Zahur, Wang, Raykova, Gascon, Doerner, Evans, and Katz [32] revisited square-root ORAM. Their work showed that relaxing the asymptotic bounds of access complexity would produce smaller circuits. They proposed an ORAM scheme of $O(\sqrt{n(\lg n)^3})$ that yields significant improvements over any of the tree-based ORAM schemes in practice. The FLORAM work by Doerner and shelat [8], which we discussed in this paper and is the closest to our work, took it a step further and presented an ORAM scheme with $O(n)$ local computation while improving upon square-root ORAM in practice.

Three-party DPF-based DORAM schemes have also been studied by Bunn, Katz, Kushilevitz, and Ostrovsky [5]. However, they use 3-party DPFs that have size $O(\sqrt{n})$. Hamlin and Varin [18] present a 2-server DORAM for secure computation that achieves both constant round communication and sub-linear work. However, unlike DUORAM, their work has a $O(\sqrt{n} \cdot \lg n)$ bandwidth cost. Sub-logarithmic DORAM has also been studied by Kushilevitz and Mour [21]. Their three-party protocol requires memory to be laid down in a complicated data structure, which is different from DUORAM, where the memory is laid down in an array. It is worth pointing out they also present a four-server ORAM protocol whose memory layout is as simple as the one DUORAM uses.

8 Conclusion

In this paper, we presented 2-party and 3-party variants of DUORAM, a Distributed ORAM protocol. One of the crucial improvements that DUORAM offers compared to previous work like FLORAM is that it uses much less bandwidth and communication rounds, and so is much less sensitive to network bandwidth and latency. Two key innovations that enable this are our novel constructions for (i) evaluating dot products of certain secret-shared vectors using communication that is only logarithmic in the vector length, and (ii) generating distributed point functions (and CSPIR queries) in a preprocessing phase, before the target point or message is known, both of which vastly reduce the online cost of the protocol.

References

[1] Donald Beaver. Efficient Multiparty Protocols Using Circuit Randomization. In *CRYPTO*, pages 420–432,

1991.

[2] Dan Boneh, David Mazieres, and Raluca Popa. Remote Oblivious Storage: Making Oblivious RAM Practical. Technical report, MIT, 2011. https://dspace.mit.edu/handle/1721.1/62006.

[3] Elette Boyle, Niv Gilboa, and Yuval Ishai. Function Secret Sharing. In *Advances in Cryptology - EUROCRYPT 2015*, pages 337–367, 2015.

[4] Elette Boyle, Niv Gilboa, and Yuval Ishai. Function secret sharing: Improvements and extensions. In *CCS*, pages 1292–1303, 2016.

[5] Paul Bunn, Jonathan Katz, Eyal Kushilevitz, and Rafail Ostrovsky. Efficient 3-Party Distributed ORAM. In *Security and Cryptography for Networks*, pages 215–232, 2020.

[6] Kai-Min Chung, Zhenming Liu, and Rafael Pass. Statistically-secure ORAM with $O((\log n)^2)$ Overhead. In *Asiacrypt*, pages 62–81, 2014.

[7] Ivan Damgård, Sigurd Meldgaard, and Jesper Buus Nielsen. Perfectly Secure Oblivious RAM without Random Oracles. In *TCC*, 2011.

[8] Jack Doerner and Abhi Shelat. Scaling ORAM for Secure Computation. In *CCS*, pages 523–535. ACM, 2017.

[9] Wenliang Du and Mikhail J. Atallah. Protocols for Secure Remote Database Access with Approximate Matching. In *E-Commerce Security and Privacy (Part II)*, Advances in Information Security, Feb 2001.

[10] Sky Faber, Stanislaw Jarecki, Sotirios Kentros, and Boyang Wei. Three-Party ORAM for Secure Computation. In *ASIACRYPT 2015*, pages 360–385, Berlin, Heidelberg, 2015. Springer-Verlag.

[11] Craig Gentry, Kenny Goldman, Shai Halevi, Charanjit Julta, Mariana Raykova, and Daniel Wichs. Optimizing ORAM and Using it Efficiently for Secure Computation. In *Privacy Enhancing Technologies Symposium*, pages 1–18, 2013.

[12] Niv Gilboa and Yuval Ishai. Distributed Point Functions and Their Applications. In *Advances in Cryptology - EUROCRYPT 2014*, pages 640–658, 2014.

[13] Oded Goldreich, Shafi Goldwasser, and Silvio Micali. How to Construct Random Functions. *J. ACM*, 33(4):792–807, 1986.

[14] Oded Goldreich and Rafail Ostrovsky. Software Protection and Simulation on Oblivious RAMs. *J. ACM*, 43(3):431–473, 1996.

[15] Michael T. Goodrich and Michael Mitzenmacher. MapReduce Parallel Cuckoo Hashing and Oblivious RAM Simulations. *CoRR*, abs/1007.1259, 2010. http://arxiv.org/abs/1007.1259.

[16] Michael T. Goodrich, Michael Mitzenmacher, Olga Ohrimenko, and Roberto Tamassia. Oblivious Ram Simulation with Efficient Worst-Case Access Overhead. In *ACM Cloud Computing Security Workshop*, pages 95–100, 2011.

[17] S. Dov Gordon, Xiao Wang, and Jonathan Katz. Simple and Efficient Two-Server ORAM. In *Asiacrypt*, pages 141–157, 2018.

[18] Ariel Hamlin and Mayank Varia. Two-server Distributed ORAM with Sublinear Computation and Constant rounds. In *PKC*, pages 499–527, 2021.

[19] Stanislaw Jarecki and Boyang Wei. 3PC ORAM with Low Latency, Low Bandwidth, and Fast Batch Retrieval. In *Applied Cryptography and Network Security*, pages 360–378, 2018.

[20] Eyal Kushilevitz, Steve Lu, and Rafail Ostrovsky. On the (in)Security of Hash-Based Oblivious RAM and a New Balancing Scheme. In *SODA*, pages 143–156, 2012.

[21] Eyal Kushilevitz and Tamer Mour. Sub-logarithmic Distributed Oblivious RAM with Small Block Size. In *PKC*, pages 3–33, 2019.

[22] Steve Lu and Rafail Ostrovsky. Distributed Oblivious RAM for Secure Two-Party Computation. In *Theory of Cryptography*, pages 377–396, 2013.

[23] Samir Jordan Menon and David J. Wu. Spiral: Fast, High-Rate Single-Server PIR via FHE Composition. In *IEEE Symposium on Security and Privacy (SP)*, pages 930–947, 2022.

[24] Moni Naor and Benny Pinkas. Oblivious Transfer and Polynomial Evaluation. In *STOC*, pages 245–254, 1999.

[25] Elaine Shi, T.-H. Hubert Chan, Emil Stefanov, and Mingfei Li. Oblivious RAM with $O((\log n)^3)$ Worst-Case Cost. In *Asiacrypt*, pages 197–214, 2011.

[26] Emil Stefanov, Marten Van Dijk, Elaine Shi, T.-H. Hubert Chan, Christopher Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. Path ORAM: An Extremely Simple Oblivious RAM Protocol. *J. ACM*, 65(4), 2018.

[27] Adithya Vadapalli, Fattaneh Bayatbabolghani, and Ryan Henry. You May Also Like... Privacy: Recommendation Systems Meet PIR. *Proc. Priv. Enhancing Technol.*, 2021(4):30–53, 2021.

[28] Adithya Vadapalli, Kyle Storrier, and Ryan Henry. Sabre: Sender-Anonymous Messaging with Fast Audits. In *IEEE Symposium on Security and Privacy (SP)*, pages 1953–1970, 2022.

[29] Xiao Wang, Hubert Chan, and Elaine Shi. Circuit ORAM: On Tightness of the Goldreich-Ostrovsky Lower Bound. In *CCS*, pages 850–861, 2015.

[30] Peter Williams and Radu Sion. Usable PIR. In *NDSS*, 2008.

[31] Peter Williams, Radu Sion, and Bogdan Carbunar. Building Castles out of Mud: Practical Access Pattern Privacy and Correctness on Untrusted Storage. In *CCS*, pages 139–148, 2008.

[32] Samee Zahur, Xiao Wang, Mariana Raykova, Adrià Gascón, Jack Doerner, David Evans, and Jonathan Katz. Revisiting Square-Root ORAM: Efficient Random Access in Multi-party Computation. In *IEEE Symposium on Security and Privacy*, pages 218–234, 2016.

A Generating Multiplicative and Dot-Product Triples

This section presents two methods to generate the multiplication triples that were discussed in Section 2.2.1. We use XOR-shared multiplicative triples in the 2-party and 3-party READ and UPDATE preprocessing protocols.

Using a third party. Here, a stateless third party (which does not collude with either other party) samples the 5-tuple (X_0, X_1, Y_0, Y_1, T) uniformly and sends the multiplicative-triples $(X_0, Y_0, (X_0 \wedge Y_1) \oplus T)$ and $(X_1, Y_1, (X_1 \wedge Y_0) \oplus T)$ to P_0 and P_1 respectively. This process of generating multiplicative triples using a noncolluding third party results in a protocol called the Du-Atallah protocol [9]. Du-Atallah's multiplication protocol is a variant of the more celebrated Beaver triples protocol, where we replace the OT with a third party.

Using Oblivious Transfer. The Du-Atallah triples used in MPC bitwise multiplication can be generated without the third party by using Oblivious Transfer with the following protocol:

1. P_0 picks (X_0, Y_0) at random from $\{0,1\}^w$ and P_1 picks (X_1, Y_1) at random from $\{0,1\}^w$.
2. P_0 and P_1 pick random words $T_0 \in \{0,1\}^w$ and $T_1 \in \{0,1\}^w$ respectively. Let $T = T_0 \oplus T_1$.
3. P_0 acts as the sender in w parallel 1-of-2 oblivious transfers with the i^{th} bits of $(T_0, X_0 \oplus T_0)$ as the input. P_1 uses the bits of Y_1 as the selection bits. Therefore, P_1 learns $(X_0 \wedge Y_1) \oplus T_0$ and computes $(X_0 \wedge Y_1) \oplus T_0 \oplus T_1 = (X_0 \wedge Y_1) \oplus T$.
4. In parallel, P_1 acts as a sender with the words $(T_1, X_1 \oplus T_1)$, and P_0 uses Y_0 as the selection word, so P_0 learns $(X_1 \wedge Y_0) \oplus T_1$ and computes $(X_1 \wedge Y_0) \oplus T_1 \oplus T_0 = (X_1 \wedge Y_0) \oplus T$.

B Proofs of DUORAM Operations

This section presents the correctness proofs of various DUORAM protocols. We will begin this section by proving the correctness of the READ protocol.

Proof of Lemma 1.

$$\begin{aligned}
\mathsf{read}_0 + \mathsf{read}_1 &= \langle D_0 + \widetilde{D}_1, \vec{\mathtt{b}}_0 \rangle - \langle \zeta_0, \vec{\mathtt{c}}_0 - \vec{\mathtt{b}}_0 \rangle + \gamma_0 \\
&+ \langle D_1 + \widetilde{D}_0, \vec{\mathtt{b}}_1 \rangle - \langle \zeta_1, \vec{\mathtt{d}}_1 - \vec{\mathtt{b}}_1 \rangle + \gamma_1 \\
\\
&= \langle D_0 + (D_1 + \zeta_1), \vec{\mathtt{b}}_0 \rangle - \langle \zeta_0, \vec{\mathtt{c}}_0 - \vec{\mathtt{b}}_0 \rangle + \gamma_0 \\
&+ \langle D_1 + (D_0 + \zeta_0), \vec{\mathtt{b}}_1 \rangle - \langle \zeta_1, \vec{\mathtt{d}}_1 - \vec{\mathtt{b}}_1 \rangle + \gamma_1 \\
\\
&= \langle D, \vec{\mathtt{b}}_0 \rangle + \langle \zeta_1, \vec{\mathtt{b}}_0 \rangle - \langle \zeta_0, (\vec{\mathtt{c}}_0 - \vec{\mathtt{b}}_0) \rangle + \gamma_0 \\
&+ \langle D, \vec{\mathtt{b}}_1 \rangle + \langle \zeta_0, \vec{\mathtt{b}}_1 \rangle - \langle \zeta_1, (\vec{\mathtt{d}}_1 - \vec{\mathtt{b}}_1) \rangle + \gamma_1 \\
\\
&= \langle D, \vec{\mathbf{e}}_{\mathtt{i}^*} \rangle + \langle \zeta_1, \vec{\mathbf{e}}_{\mathtt{i}^*} \rangle + \langle \zeta_0, \vec{\mathbf{e}}_{\mathtt{i}^*} \rangle \\
&- \langle \zeta_0, \vec{\mathtt{c}}_0 \rangle - \langle \zeta_1, \vec{\mathtt{d}}_1 \rangle + \gamma_0 + \gamma_1 \\
\\
&= \langle D, \vec{\mathbf{e}}_{\mathtt{i}^*} \rangle + \langle \zeta_1, \vec{\mathbf{e}}_{\mathtt{i}^*} \rangle + \langle \zeta_0, \vec{\mathbf{e}}_{\mathtt{i}^*} \rangle \\
&- \langle \zeta_0, \vec{\mathtt{c}}_0 \rangle - \langle \zeta_1, \vec{\mathtt{d}}_1 \rangle - \langle \zeta_0, \vec{\mathtt{c}}_1 \rangle - \langle \zeta_1, \vec{\mathtt{d}}_0 \rangle \\
\\
&= \langle D, \vec{\mathbf{e}}_{\mathtt{i}^*} \rangle + \langle \zeta_1, \vec{\mathbf{e}}_{\mathtt{i}^*} \rangle + \langle \zeta_0, \vec{\mathbf{e}}_{\mathtt{i}^*} \rangle - \langle \zeta_1, \vec{\mathbf{e}}_{\mathtt{i}^*} \rangle - \langle \zeta_0, \vec{\mathbf{e}}_{\mathtt{i}^*} \rangle \\
\\
&= \langle D, \vec{\mathbf{e}}_{\mathtt{i}^*} \rangle
\end{aligned}$$

□

Next, we will prove the correctness of the REFRESH-BLINDS protocol.

Proof of Lemma 3.

$$\begin{aligned}
\widetilde{D}'_0 &= \widetilde{D_0} + \vec{\mathtt{c}}^*_1 - \vec{\mathtt{b}}^*_1 \\
&= (D_0 + \zeta_0) + \vec{\mathtt{c}}^*_1 - \vec{\mathtt{b}}^*_1 \\
&= (D_0 + \zeta_0) + (\mathsf{M} \cdot \vec{\mathbf{e}}_{\mathtt{i}^*} - \vec{\mathtt{c}}^*_0) - (\mathsf{M} \cdot \vec{\mathbf{e}}_{\mathtt{i}^*} - \vec{\mathtt{b}}^*_0) \\
&= (D_0 + \vec{\mathtt{b}}^*_0) + (\zeta_0 - \vec{\mathtt{c}}^*_0) \\
&= (D'_0 + \zeta'_0)
\end{aligned}$$

and similarly for $\widetilde{D}'_1$. □

Finally, we will prove the correctness of the 2P-DUORAM READ protocol.

Proof of Lemma 4.

$$\begin{aligned}\mathsf{read}_0+\mathsf{read}_1 &= c_0-\mathsf{r}_0+c_1-\mathsf{r}_1\\ &= D_1[\,\mathtt{i}^*]+\mathsf{r}_1-\mathsf{r}_0+D_0[\,\mathtt{i}^*]+\mathsf{r}_0-\mathsf{r}_1\\ &= D_1[\,\mathtt{i}^*]+D_0[\,\mathtt{i}^*]\\ &= D[\,\mathtt{i}^*]\end{aligned}$$

□

C DPF Generation Algorithm

The following protocol is the DPF generation presented in the FLORAM paper by Doerner and shelat [8]. Suppose that we want to create DPF at the target location, $\mathtt{i}^*$. Represent $\mathtt{i}^*$ as a binary bit vector $\vec{\mathtt{i}^*}$. The parties P_0 and P_1 start with XOR shares $\vec{\mathtt{i}^*}_0$ and $\vec{\mathtt{i}^*}_1$ of $\vec{\mathtt{i}^*}$, and create random seeds $v_0^{()} \in \{0,1\}^\lambda$ and $v_1^{()} \in \{0,1\}^\lambda$ respectively to use as the roots of binary trees. They use a length-doubling PRG to construct the remainder of the trees, as outlined below. We denote by $\vec{v}_{b,\ell}$ the nodes at level ℓ, and by $\vec{\mathbf{t}}_{b,\ell}$ the flags at level ℓ, both in the tree share held by P_b. We first set the least significant bit (lsb) of P_b's root as b; i.e. set $\mathsf{lsb}(v_0^{()}) \leftarrow 0$ and $\mathsf{lsb}(v_1^{()}) \leftarrow 1$. Set $\vec{\mathbf{t}}_{0,0}[0] \leftarrow \mathsf{lsb}(v_0^{()})$ and $\vec{\mathbf{t}}_{1,0}[0] \leftarrow \mathsf{lsb}(v_1^{()})$.

For each layer ℓ starting at the root with $\ell = 0$:

1. For $b \in \{0,1\}$, P_b uses the PRG to construct the labels on the children of each node in this level. The left and right children of node i at this layer are denoted as $(v_{b,\ell}^{(i\|\mathsf{L})}, v_{b,\ell}^{(i\|\mathsf{R})})$. Therefore, we have $\vec{v}_{b,\ell+1}[2\cdot\mathtt{i}] = v_{b,\ell}^{(\mathtt{i}\|L)}$ and $\vec{v}_{b,\ell+1}[2\cdot\mathtt{i}+1] = v_{b,\ell}^{(\mathtt{i}\|R)}$.

2. For $b \in \{0,1\}$, P_b computes $L_b \leftarrow \bigoplus_{j=0}^{2^\ell-1} v_{b,\ell}^{(j\|\mathsf{L})}$ and $R_b \leftarrow \bigoplus_{j=0}^{2^\ell-1} v_{b,\ell}^{(j\|\mathsf{R})}$.

3. P_0 and P_1 use an MPC multiplication protocol to compute the correction word $\mathsf{cw}^{(\ell+1)} \leftarrow \big((\vec{\mathtt{i}^*}_0[\ell]\oplus\vec{\mathtt{i}^*}_1[\ell])\cdot(L_0\oplus L_1)\big) \oplus \big((1\oplus\vec{\mathtt{i}^*}_0[\ell]\oplus\vec{\mathtt{i}^*}_1[\ell])\cdot(R_0\oplus R_1)\big)$.

4. P_b computes $\mathsf{cwt}_L^b \leftarrow \mathsf{lsb}(L_b) \oplus \vec{\mathtt{i}^*}_b[\ell]$ and $\mathsf{cwt}_R^b \leftarrow \mathsf{lsb}(R_b) \oplus \vec{\mathtt{i}^*}_b[\ell]$, and exchanges those values with the other party; the parties then both compute $\mathsf{cwt}_L \leftarrow \mathsf{cwt}_L^0 \oplus \mathsf{cwt}_L^1 \oplus 1$ and $\mathsf{cwt}_R \leftarrow \mathsf{cwt}_R^0 \oplus \mathsf{cwt}_R^1$.

5. P_b computes $\vec{\mathbf{t}}_{b,\ell+1}[2\cdot\mathtt{i}] \leftarrow \mathsf{lsb}(\vec{v}_{b,\ell+1}[2\cdot\mathtt{i}]) \oplus (\vec{\mathbf{t}}_{b,\ell}[\mathtt{i}]\cdot\mathsf{cwt}_L)$ and $\vec{\mathbf{t}}_{b,\ell+1}[2\cdot\mathtt{i}+1] \leftarrow \mathsf{lsb}(\vec{v}_{b,\ell+1}[2\cdot\mathtt{i}+1]) \oplus (\vec{\mathbf{t}}_{b,\ell}[\mathtt{i}]\cdot\mathsf{cwt}_R)$, for all $\mathtt{i} \in [0,2^\ell)$.

6. P_b updates $\vec{v}_{b,\ell+1}[2\cdot\mathtt{i}] \leftarrow \vec{v}_{b,\ell+1}[2\cdot\mathtt{i}] \oplus (\vec{\mathbf{t}}_{b,\ell}[\mathtt{i}]\cdot\mathsf{cw}^{(\ell+1)})$ and $\vec{v}_{b,\ell+1}[2\cdot\mathtt{i}+1] \leftarrow \vec{v}_{b,\ell+1}[2\cdot\mathtt{i}+1] \oplus (\vec{\mathbf{t}}_{b,\ell}[\mathtt{i}]\cdot\mathsf{cw}^{(\ell+1)})$, for all $\mathtt{i} \in [0,2^\ell)$.

D Converting an XOR-shared standard basis vector to additive shares

In this section, we elaborate upon the informal description of the share conversion algorithm described in Section 4.1.1. Recall that P_0 and P_1 hold DPFs without the final correction word; that is, P_0 holds $(\vec{v}_0, \mathring{\vec{\mathbf{t}}}_0, \mathcal{F}_0)$ and P_1 holds $(\vec{v}_1, \mathring{\vec{\mathbf{t}}}_1, \mathcal{F}_1)$ such that $\mathring{\vec{\mathbf{t}}}_0 \oplus \mathring{\vec{\mathbf{t}}}_1 = \vec{\mathbf{e}}_{\mathtt{ri}}$ and $\vec{v}_0 + \vec{v}_1 = -(\mathcal{F}_0+\mathcal{F}_1)\cdot\vec{\mathbf{e}}_{\mathtt{ri}}$. Their goal is to end up with vectors $\vec{\mathbf{t}}_0$ and $\vec{\mathbf{t}}_1$ such that $\vec{\mathbf{t}}_0 + \vec{\mathbf{t}}_1 = \vec{\mathbf{e}}_{\mathtt{ri}}$. An important point to note is that, in this case, $\mathring{\vec{\mathbf{t}}}_b$ and the pair $(\vec{v}_b, \mathcal{F}_b)$ are not of the same DPF (but have the same target index). In other words, we use an additional DPF to perform the share conversion.

1. P_0 interprets its flag vector as a word vector. P_1 also interprets its flag vector as a word vector and multiplies it by -1. In other words, P_0 computes $\widehat{\vec{\mathbf{t}}}_0[\mathtt{i}] \leftarrow (\mathring{\vec{\mathbf{t}}}_0[\mathtt{i}])$ for all $\mathtt{i}$, and P_1 computes $\widehat{\vec{\mathbf{t}}}_1[\mathtt{i}] \leftarrow -(\mathring{\vec{\mathbf{t}}}_1[\mathtt{i}])$ for all $\mathtt{i}$.

2. For $b \in \{0,1\}$, P_b computes $\mathsf{pm}_b \leftarrow \sum_i \widehat{\vec{\mathbf{t}}}_b[\mathtt{i}]$.

3. For $b \in \{0,1\}$, P_b selects a random word r_b to blind pm_b, and the P_b exchange $\mathsf{pm}_b + \mathsf{r}_b$.

4. For $b \in \{0,1\}$, P_b updates $\widehat{\vec{\mathbf{t}}}'_b \leftarrow \widehat{\vec{\mathbf{t}}}_b \cdot ((\mathsf{pm}_{1-b}+\mathsf{r}_{1-b}) + \mathsf{pm}_b + \mathsf{r}_b)$.

5. The parties use MPC to compute shares $\widetilde{\mathcal{F}}_0$ and $\widetilde{\mathcal{F}}_1$ of $(\mathcal{F}_0+\mathcal{F}_1)\cdot(\mathsf{pm}_0+\mathsf{pm}_1)$.

6. The parties then compute $\mathcal{F}'_0 \leftarrow \widetilde{\mathcal{F}}_0 + \mathsf{r}_0$ and $\mathcal{F}'_1 \leftarrow \widetilde{\mathcal{F}}_1 + \mathsf{r}_1$ respectively.

7. The parties, finally reconstruct $\mathcal{F}' \leftarrow \mathcal{F}'_0 + \mathcal{F}'_1$.

8. For $b \in \{0,1\}$, P_b updates $\vec{\mathbf{t}}_b \leftarrow \widehat{\vec{\mathbf{t}}}'_b - \vec{v}_b - (\widehat{\vec{\mathbf{t}}}_b \cdot \mathcal{F}')$.

Lemma 5. *After running the above protocol,* $\vec{\mathbf{t}}_0 + \vec{\mathbf{t}}_1 = \mathring{\vec{\mathbf{t}}}_0 \oplus \mathring{\vec{\mathbf{t}}}_1 = \vec{\mathbf{e}}_{\mathtt{ri}}$.

Proof. Denote $\mathsf{pm} = \mathsf{pm}_0 + \mathsf{pm}_1$. First observe that $\widehat{\vec{\mathbf{t}}}_0[\mathtt{i}] + \widehat{\vec{\mathbf{t}}}_1[\mathtt{i}] = 0$ for $\mathtt{i} \neq \mathtt{ri}$, so $\mathsf{pm} = \widehat{\vec{\mathbf{t}}}_0[\mathtt{ri}] + \widehat{\vec{\mathbf{t}}}_1[\mathtt{ri}] = \mathring{\vec{\mathbf{t}}}_0[\mathtt{ri}] - \mathring{\vec{\mathbf{t}}}_1[\mathtt{ri}]$, which is either $+1$ or -1. Also note that $\widehat{\vec{\mathbf{t}}}_0 + \widehat{\vec{\mathbf{t}}}_1 = \mathsf{pm}\cdot\vec{\mathbf{e}}_{\mathtt{ri}}$.

Let $\mathsf{r} = \mathsf{r}_0 + \mathsf{r}_1$, $\mathcal{F} = \mathcal{F}_0 + \mathcal{F}_1$, $\widetilde{\mathcal{F}} = \widetilde{\mathcal{F}}_0 + \widetilde{\mathcal{F}}_1$.

Then $\widehat{\vec{\mathbf{t}}}'_b = \widehat{\vec{\mathbf{t}}}_b \cdot (\mathsf{pm}+\mathsf{r})$, so $\widehat{\vec{\mathbf{t}}}'_0 + \widehat{\vec{\mathbf{t}}}'_1 = (\mathsf{pm}\cdot\vec{\mathbf{e}}_{\mathtt{ri}})\cdot(\mathsf{pm}+\mathsf{r}) = \vec{\mathbf{e}}_{\mathtt{ri}}\cdot(1+\mathsf{pm}\cdot\mathsf{r})$. Also $\widetilde{\mathcal{F}} = \mathcal{F}\cdot\mathsf{pm}$, and $\mathcal{F}' = \widetilde{\mathcal{F}} + \mathsf{r} = \mathcal{F}\cdot\mathsf{pm} + \mathsf{r}$. Finally, recall that $\vec{v}_0 + \vec{v}_1 = -\mathcal{F}\cdot\vec{\mathbf{e}}_{\mathtt{ri}}$, so that $\vec{\mathbf{t}}_0 + \vec{\mathbf{t}}_1 = (\widehat{\vec{\mathbf{t}}}'_0 + \widehat{\vec{\mathbf{t}}}'_1) - (\vec{v}_0+\vec{v}_1) - (\widehat{\vec{\mathbf{t}}}_0 + \widehat{\vec{\mathbf{t}}}_1)\cdot\mathcal{F}' = \vec{\mathbf{e}}_{\mathtt{ri}}\cdot\big((1+\mathsf{pm}\cdot\mathsf{r}) + \mathcal{F} - \mathsf{pm}\cdot\mathcal{F}'\big) = \vec{\mathbf{e}}_{\mathtt{ri}}\cdot((1+\mathsf{pm}\cdot\mathsf{r}) + \mathcal{F} - \mathsf{pm}\cdot(\mathcal{F}\cdot\mathsf{pm}+\mathsf{r})) = \vec{\mathbf{e}}_{\mathtt{ri}}$. □

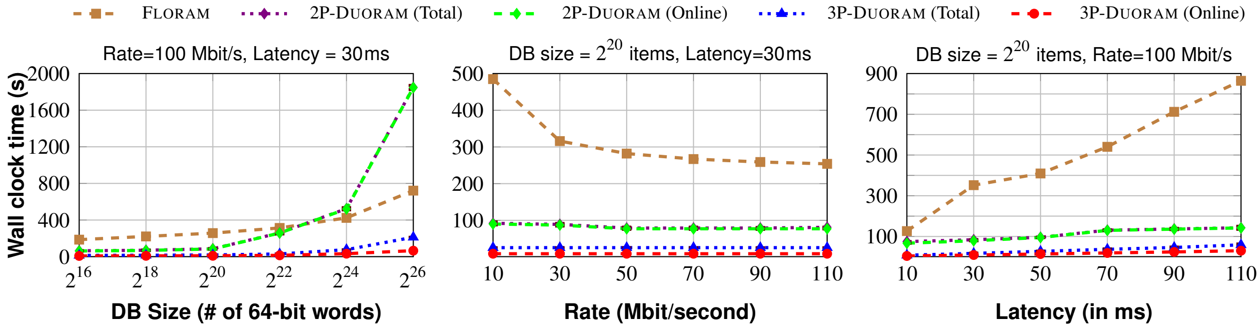


Figure 10: Comparing FLORAM and DUORAM to do 128 dependent reads for different parameters of *database size*, *latency*, and *bandwidth* on databases with 8-byte words. (The error bars are too small and thus not visible.)

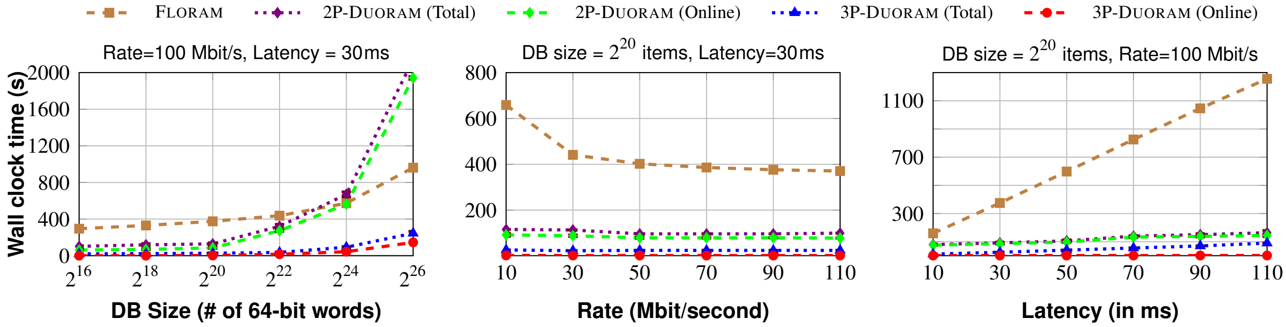


Figure 11: Comparing FLORAM and DUORAM to do 128 writes for different parameters of *database size*, *latency*, and *bandwidth* on databases with 8-byte words. (The error bars are too small and thus not visible.)

Table 3: Comparing wall-clock time and bandwidth to do one READ Operation between 2P-DUORAM and FLORAM while setting the throughput to be 1 Mbit/s and latency to 100 ms.

		2P-DUORAM (preprocessing)		2P-DUORAM (online)		FLORAM	
operation	**size**	Wall Clock Time	Bandwidth	Wall Clock Time	Bandwidth	Wall Clock Time	Bandwidth
Read	2^{20}	9 ± 1 s	17.9 KiB	0.94 ± 0.03 s	14 KiB	5640 ± 10 s	1808 KiB
Read	2^{25}	21 ± 2 s	18.4 KiB	9.4 ± 0.1 s	20 KiB	–	–

E Additional Experiments

This section shows the results of some additional experiments that were very similar to experiments we reported in Section 6.2. Figures 10 and 11 compare the performance of DUORAM with FLORAM to do 128 dependent *read* and 128 *write* operations, respectively. These results are comparable to those for the 128 *interleaved* operations reported in Figure 7 in that section.

Table 3 compares our CSPIR-based 2-Party DUORAM with FLORAM when we reduce the network throughput to 1 Mbit/s. We observe that under such poor network conditions, when the number of items is 2^{20}, our 2-Party DUORAM is over 500 times faster than FLORAM. On the other hand, when the number of items is 2^{25}, FLORAM did not finish execution even after running for 10 hours.