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

Adithya Vadapalli, University of Waterloo; Ryan Henry, University of Calgary; Ian Goldberg, University of Waterloo

https://www.usenix.org/conference/usenixsecurity23/presentation/vadapalli


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 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—technical and practical advantages relative to FLORAM. Most notably, DUORAM exploits a subtle observation to support sequences of arbitrarily interleaved, oblivious reads and writes with online communication independent of the memory size for realistically sized memories. 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 \( e_i \) comprising all 0s except for a single 1 appearing in coordinate \( i \)). 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-

* An extended version of this paper is available [28].
Table 1: Comparing computation, bandwidth, and the number of rounds of DUORAM with previous work. The gray color represents the preprocessing cost. We note that Kushilevitz and Mour’s work [21] is a 4-party protocol; if we turn DUORAM into a 4-party or (3 + 1)-party protocol, then the Rounds will reduce to 1 + 1 and the Computation to \( O(\lg n) + O(n) \). The improvement is because we will no longer require an MPC to generate the DPFs; the fourth party can act as a dealer to create and distribute the DPFs.

<table>
<thead>
<tr>
<th></th>
<th># Parties</th>
<th>Rounds</th>
<th>Bandwidth</th>
<th>Computation</th>
</tr>
</thead>
<tbody>
<tr>
<td>FLORAM [8]</td>
<td>2</td>
<td>( O(\lg n) )</td>
<td>( O(\sqrt{n}) )</td>
<td>( O(n) )</td>
</tr>
<tr>
<td>Hamlin and Varia [18]</td>
<td>2</td>
<td>( O(1) )</td>
<td>( O(\sqrt{n\lg n}) )</td>
<td>( O(\sqrt{n^2} \lg n) )</td>
</tr>
<tr>
<td>Jarecki and Wei [19]</td>
<td>3</td>
<td>( O(\lg n) )</td>
<td>( O(\lg^3 n) )</td>
<td>( O(\lg^2 n) )</td>
</tr>
<tr>
<td>Bunn et al. [5]</td>
<td>3</td>
<td>( O(\sqrt{n}) )</td>
<td>( O(\sqrt{n}) )</td>
<td>( O(n) )</td>
</tr>
<tr>
<td>Kushilevitz and Mour [21]</td>
<td>4</td>
<td>( O(1) )</td>
<td>( O(\lg n) )</td>
<td>( O(n) )</td>
</tr>
<tr>
<td>DUORAM</td>
<td>2 or 3</td>
<td>( O(\lg n) + 1 )</td>
<td>( O(\lg n) + O(1) )</td>
<td>( O(n) + O(n) )</td>
</tr>
</tbody>
</table>

ory \( \mathbf{D} \in \{0, 1\}^{n \times w} \) consisting of \( n \) words that are each \( w \) bits long. We denote the word at memory address \( i \) in \( \mathbf{D} \) by \( \mathbf{D}[i] \in \{0, 1\}^w \). Depending on the type of memory access being performed, the computation parties hold either an encrypted copy of \( \mathbf{D} \) (when reading) or a secret shared copy of \( \mathbf{D} \) (when writing); a “refresh” operation converts \( \mathbf{D} \) 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 \( \mathbf{D} \) blinded using a pseudorandom function \( F \); i.e., \( P_0 \) and \( P_1 \) hold in common blinded memory \( \overline{\mathbf{D}} \in \{0, 1\}^{n \times w} \) such that \( \overline{\mathbf{D}}[i] \leftarrow \mathbf{D}[i] \oplus F(k_0, i) \oplus F(k_1, i) \) for each \( i \in \{0..n\} \).

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

1. \( P_0 \) and \( P_1 \) collaboratively sample a DPF representation of \( \mathbf{e}_{i^*} \) using 2-MPC;
2. each \( P_b \) locally expands its DPF share into a bit vector \( \mathbf{t}_b \in \{0, 1\}^n \) and then it computes \( \overline{\mathbf{R}}_b[i] \leftarrow \bigoplus_{(t_b[i] = 1)} \overline{\mathbf{D}}[i] \); and, finally,
3. \( P_0 \) and \( P_1 \) run another 2-MPC to evaluate \( F \) at the (unknown to each party) input \( i^* \) to produce shares of \( \mathbf{D}[i^*] = \overline{\mathbf{R}}_0 \oplus \overline{\mathbf{R}}_1 \oplus F(k_0, i^*) \oplus F(k_1, i^*) \).

**Oblivious writes in FLORAM.** Computation parties \( P_0 \) and \( P_1 \) hold XOR-shares \( \mathbf{D}_0 \) and \( \mathbf{D}_1 \) of the memory \( \mathbf{D} \). Given shares of a *target index–value* pair \( (i^*, M) \), they replace \( \mathbf{D}[i^*] \) with \( \mathbf{D}[i^*] \oplus M \) using PIR-writing, as follows:

1. \( P_0 \) and \( P_1 \) collaboratively sample a DPF representation of \( \mathbf{v} = \mathbf{M} \cdot \mathbf{e}_{i^*} \) using 2-MPC; and
2. each \( P_b \) locally expands its DPF share into an \( n \)-element vector \( \mathbf{v}_b \in \{\{0, 1\}^w\}^n \) and then it computes \( \mathbf{D}'_b \leftarrow \mathbf{D}_b \oplus \mathbf{v}_b \).

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

**Refreshing the FLORAM database.** Recall that FLORAM requires an encrypted copy of \( \mathbf{D} \) for reading and a XOR-shared copy of \( \mathbf{D} \) for writing. The refresh operation transforms XOR-shared memory (suitable for writing) into encrypted memory (suitable for reading). Given shares \( \mathbf{D}_0 \) and \( \mathbf{D}_1 \) of \( \mathbf{D} \), 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 \( i \), \( P_0 \) and \( P_1 \) compute \( \mathbf{R}[i] = R_0[i] \oplus F(k_0, i) \) and \( \mathbf{R}[i]'' = R_1[i] \oplus F(k_1, i) \) respectively.
2. For all \( i \), \( P_0 \) and \( P_1 \) exchange \( \mathbf{R}[i]'' \) to compute \( \mathbf{R}[i]'' = \mathbf{R}[i]'' \oplus \mathbf{R}[i]'' \), 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 \( O(\sqrt{n}) \).

While we compare our work mainly against FLORAM (because it is the most closely related to our work and has a shared lineage), there have been several other works in the 3-party and 4-party DORAM setting. Table 1 compares DUORAM with some of the prominent recent work to do one access in a database holding \( n \) constant-sized words. DUORAM’s critical insight is that reducing latency and increasing bandwidth
between parties is much more challenging than increasing local computational power, and so it focuses on reducing the round complexity and bandwidth usage. Memory accesses in DuORAM require only two messages (one round) of communication.

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 $O(\lg n)$ 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 of any communication cost, compared with FLORAM’s $O(n)$ communication for initialization), yielding a very efficient 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 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 is $O(\lg n)$, 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 for a database of size $2^{25}$. This is in contrast to FLORAM, which did not complete the computations for the same database size of $2^{25}$ even 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. 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 (3P-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 (2P-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 subprotocols; detailed discussions of the 3P-DuORAM and 2P-DuORAM protocols follow in Sections 4 and 5. 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 its domain, except at one special point (called the “target point”), where it evaluates to a non-zero value. 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.
value (called the “target value”). In the definition below (Definition 1), \( \hat{i} \) is the “target point” and M is the “target value”.

Definition 1. A point function is a function \( p_{Y,M}: \{0,n\} \rightarrow \{0,1\}^* \) such that \( p_{Y,M}(\hat{i}) = M \) and \( p_{Y,M}(\hat{i}) = 0 \) 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 two parties succinctly. Definition 2 (a restatement of Definition 4 from Vadapalli, Storrier, and Henry [29]) formally defines \((2,2)\)-DPFs.

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

1. Correctness: If \((k_0,k_1) \leftarrow \text{GEN}(\hat{i}^*,M) \), then, for all \( \hat{i} \in \{0,n\} \),
   \[
   \text{EVAL}(k_0,\hat{i}) + \text{EVAL}(k_1,\hat{i}) = \begin{cases} M & \text{if } \hat{i} = \hat{i}^*, \\ 0 & \text{otherwise.} \end{cases}
   \]

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

The \( k_b \) output by \( \text{GEN} \) 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 \( G_{2^\lambda} \), to construct Goldreich, Goldwasser, Micali styled PRFs [13]. We denote the left and right halves of the outputs of \( G_{2^\lambda}(\cdot) \) as \( G_L(\cdot) \) and \( G_R(\cdot) \) respectively. Boyle, Gilboa, and Ishai’s construction follows from the following observation: \( G_L(s_0) = G_L(s_1) \) and \( G_R(s_0) = G_R(s_1) \) if and (essentially) only if \( s_0 = s_1 \). The GEN algorithm (for a DPF tree with height \( h \)) begins with selecting two random seeds, namely, \( s_0^h \) and \( s_1^h \). 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 Vadapalli et al. [29]), the nodes in the binary tree generated by recursively applying the length-doubling PRG have as a subscript the bit indicating which party’s tree the node belongs to, and as a superscript the binary string indicating the path from the root to the node (the tree’s root is \( s_0^0 \)). For any node \( s_b^i \), where \( b \) is a binary string, the outputs of the length-doubling PRG are \( (s_b^{i[0]:i[1]},s_b^{i[1]:i[1]}) \). 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 \( \hat{i}^* \), and that the node pair for leaf \( \hat{i}^* \) XOR to M. Of course, the trees generated by 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 tree pairs of height $h$, suppose that $(cw^{(0)}, \ldots, cw^{(h)})$ represent the correction words. In the first step, the two root seeds are expanded to get $(s_0^{(0)}, s_1^{(0)}) ← G_2^x(s_0^{(0)})$ and $(s_0^{(0)}, s_1^{(0)}) ← G_2^x(s_1^{(0)})$. Next, exactly one pair of children $(s_0^{(1)}, s_1^{(1)})$ is transformed such that, $s_0^{(1)} ← s_0^{(0)} \oplus cw^{(1)}$ and $s_1^{(1)} ← s_0^{(0)} \oplus cw^{(1)}$, while the other pair of children $(s_1^{(1)}, s_1^{(1)}/s_1^{(1)}/s_1^{(1)})$ remains the same. In other words, for exactly one of the seeds, the “correction word” with level 1 is XOR-ed into the children. The flag bit associated with the seed determines if a party XORs the 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 pair, except on the path from the root to leaf $1^4$, where they will be different. The correction words are designed so that after the transformation, we have the property that either $s_0^{(0)} = s_1^{(0)}$ (if $1^4$ starts with a 1 bit) or $s_0^{(0)} = s_1^{(0)}$ (if $1^4$ 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 $1^4$) to being part of an equal pair. Applying $h$ correction words results in leaves such that all the leaf pairs except $1^4$ of the two trees XOR to 0, and leaf pair $1^4$ XORs to something random. Therefore, DPFs also have a notion of a final correction word, denoted as $\mathcal{F}$. Unlike the correction words, $(cw^{(1)}, \ldots, cw^{(h)})$, 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 $1^4$ 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 $\text{EVALFULL}$, which traverses the DPF tree in a depth-first manner to reduce the amortized number of PRG evaluations from $O(\log n)$ to $O(1)$ cost per leaf node [4]. The function produces $(v_b, t_b) ← \text{EVALFULL}(k_b)$ where $v_b$ is the vector of labels of all the leaves in tree $b$ and $t_b$ is the vector of the flags on all the leaves in tree $b$. We have the property that $t_0 \oplus t_1 = e_1$, and $v_0 \oplus v_1 = e_1 \cdot M$. Also, we have for all $i$ that $v_b[i] = \text{EVAL}(k_b, i)$.

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 learns anything else except $y$. In other words, $P_i$ should not learn anything else except $y$. We have the following properties:

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 server-aided 2-MPC, is a 2-party MPC protocol with a third party (which neither holds any secret input nor receives any messages during the protocol) that does not collide with any other two parties. The third party merely facilitates the MPC by sending some correlated randomness to the two MPC parties.
2.2.1 AND and Dot Product on Secret Shares

We first define the notion of AND triples (on XOR shares) and dot-product triples (on vectors of additive shares), which facilitate computing (bitwise) AND and dot products on secret shares. In Appendix A, we describe two main ways to generate these triples, namely a (2 + 1)-MPC 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 AND and dot products, which are denoted as (\cdot, \cdot), which is an efficient (2 + 1)-MPC variant of the well-known Beaver triples [1] based AND protocol.

Definition 4 (AND Triples). \((X_0, Y_0, Z_0) \in \{0, 1\}^{|x|} \times \{0, 1\}^{|y|} \times \{0, 1\}^{|z|}\) and \((X_1, Y_1, Z_1) \in \{0, 1\}^{|x|} \times \{0, 1\}^{|y|} \times \{0, 1\}^{|z|}\) are called AND triples if for a random \(T \in \{0, 1\}^{|t|}\), the following holds true: (i) \(Z_0 = (X_0 \land Y_1) \oplus T\), and (ii) \(Z_1 = (X_1 \land Y_0) \oplus T\).

Definition 5 (Dot-product Triples). \((X_0, Y_0, Z_0) \in \mathbb{Z}_2^{n_x} \times \mathbb{Z}_2^{n_y} \times \mathbb{Z}_2^{n_z}\) and \((X_1, Y_1, Z_1) \in \mathbb{Z}_2^{n_x} \times \mathbb{Z}_2^{n_y} \times \mathbb{Z}_2^{n_z}\) are called dot-product triples if for a random \(T \in \mathbb{Z}_2^{n_x}\) the following holds true: (i) \(Z_0 = \langle X_0, Y_1 \rangle + T\), and (ii) \(Z_1 = \langle X_1, Y_0 \rangle - T\).

P and \(P_1\) hold \((x_0, y_0)\) and \((x_1, y_1)\) respectively and their goal is to compute shares of \((x_0 \oplus x_1) \land (y_0 \land y_1)\). Suppose \(P_0\) and \(P_1\) hold the AND triples \((X_0, Y_0, Z_0)\) and \((X_1, Y_1, Z_1)\). First, \(P_0\) sends \((x_b, y_b)\) and \((y_b, y_b)\) to \(P_1\), for \(b \in \{0, 1\}\). \(P_0\) computes \(z_0 = \langle x_0 + y_0 + (y_1 + y_1) \rangle \oplus Z_0 \oplus (x_1 \land y_0)\) and \(P_1\) computes \(z_1 = \langle x_1 + y_1 + (y_0 + y_0) \rangle \oplus Z_1 \oplus (x_0 \land y_1)\) respectively. Now, observe that \(z_0 \land z_1 = \langle x_0 \oplus x_1 \rangle \land (y_0 \land y_1)\).

Similary, to compute dot product triples, assume that parties \(P_0\) and \(P_1\) hold \((x_0, y_0)\) and \((x_1, y_1)\) respectively, such that \(x = x_0 + x_1\) and \(y = y_0 + y_1\). Their goal is to obtain shares of \(\langle x, y \rangle\). \(P_0\) and \(P_1\) hold the dot-product triples \((X_0, Y_0, Z_0)\) and \((X_1, Y_1, Z_1)\). \(P_0\) sends \((x_0 + x_b)\) and \((y_b + y_b)\) to \(P_1\), for \(b \in \{0, 1\}\). \(P_0\) computes \(z_0 = \langle x_0 + y_0 + (y_1 + y_1) \rangle - \langle y_0, (x_1 + x_1) \rangle + Z_0\) and \(P_1\) computes \(z_1 = \langle x_1 + y_1 + (y_0 + y_0) \rangle - \langle Y_1, (x_0 + x_0) \rangle + Z_1\) respectively. Now, observe that \(z_0 \land z_1 = \langle x_0 \oplus x_1 \rangle \land (y_0 \land y_1)\).

3 Communication Efficient 3-Party DORAM

We 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 required 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 change how data is stored in memory, its initialization does not involve any communication. The parties only need to allocate sufficient (zeroed) storage. Another surprising consequence of avoiding the refresh operation is that we have an online communication cost that is independent of the size of the database.

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 CP-SIA 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, \(D_0 \oplus D_1 = D\). All additive secret sharings are treated as integers mod \(2^w\), or vectors of same.

We use \(D[i]\) denote the \(i\)th word of the database. Similarly, \(D_b[i]\) denotes the \(i\)th word of \(D_b\), thus they are the shares of the \(i\)th word of the database. We denote by \(\mathbb{E}(\cdot)\) 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 \(\mathbb{E}(\cdot)\). The shares of \(\mathbb{E}(\cdot)\) are \(\mathbb{E}(\cdot)_0\) and \(\mathbb{E}(\cdot)_1\). Similarly, \(M_0\) and \(M_1\) are the shares of \(M\). In our setting, \(P_0\) holds \((D_0, D_1, \mathbb{E}(\cdot)_0, \mathbb{E}(\cdot)_1, M_0)\), \(P_1\) holds \((D_1, D_0, \mathbb{E}(\cdot)_1, \mathbb{E}(\cdot)_0, M_1)\), and \(P_2\) holds \((\zeta_0, \zeta_1)\). DUORAM initializes the database shares and the Du-Atallah blinding vectors to all zeros, thus making initialization communication-fee, compared to the O(n) communication cost for initializing FLORAM. Formally, the initialization results in \(D_0 \leftarrow 0, D_1 \leftarrow 0, \mathbb{E}(\cdot)_0 \leftarrow 0, \mathbb{E}(\cdot)_1 \leftarrow 0, \mathbb{E}(\cdot) \leftarrow 0, D_0 \leftarrow 0, D_1 \leftarrow 0\).

3.1 High-level Working of DUORAM

For the high-level exposition of DUORAM, we consider a trusted source, given a target location \(\mathbb{E}(\cdot)\) and a target value \(M\), which creates six pairs of word vectors (mod \(2^w\)) with the property that \(t_0^{(i)} + t_1^{(i)} = t_0^{(i)} + t_1^{(i)} = e_v\) and \(v_0^{(i)} + v_1^{(i)} = v_0^{(i)} + v_1^{(i)} = M \cdot e_v\). The trusted source sends: (i) \((t_0^{(i)}, t_0^{(i)}, t_0^{(i)})\) to \(P_0\), (ii) \((t_0^{(i)}, t_1^{(i)}, t_1^{(i)})\) to \(P_1\), and (iii) \((t_0^{(i)}, t_0^{(i)}, t_0^{(i)})\) to \(P_2\), and correspondingly: (i) \((v_0^{(i)} + v_0^{(i)} + v_0^{(i)})\) to \(P_0\), (ii) \((v_1^{(i)} + v_1^{(i)} + v_1^{(i)})\) to \(P_1\), and (iii) \((v_0^{(i)} + v_1^{(i)})\) to \(P_2\). We note the \(t_0^{(i)}\) and \(v_0^{(i)}\) vectors (for \(k \in \{1, 2, 3\}\)) are almost the flag and label vectors of a DPF, except the aforementioned 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.

---

\(^3\)The trusted source is for exposition only; we will remove it shortly. Also note that this formulation requires that \(\mathbb{E}(\cdot)\) and \(M\) be known at DPF generation time, a requirement we will also remove.
3.1.1 Read Operation

The goal of the Read protocol is to read from the database, \( D \), the word addressed by the target index, \( i^* \). In other words, \( P_0 \) and \( P_1 \), who hold the additive shares of \( i^* \), along with the shares of the database \( D \), want to obtain the additive shares of \( D_{i^*} \). We observe that \( D_{i^*} = (D, e_i) \). While we describe the details of the 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 \( i^* \) 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. While the idea is to use Du–Atallah-style dot products to compute the shares of \( (D, e_i) \), 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. Crucially, whereas previous protocols required linear communication 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 known, 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 \( \gamma_0 \leftarrow \langle \xi_0, t_1^{(3)} \rangle + \rho \) to \( P_0 \) and \( \gamma_1 \leftarrow \langle \xi_1, t_1^{(3)} \rangle - \rho \) to \( P_1 \).
2. \( P_0 \) outputs \( \text{read}_0 \leftarrow \langle D_0 + \tilde{D}_0, t_0^{(3)} \rangle - \langle \xi_0, t_0^{(3)} - t_0^{(1)} \rangle + \gamma_0 \).
3. \( P_1 \) outputs \( \text{read}_1 \leftarrow \langle D_1 + \tilde{D}_1, t_1^{(3)} \rangle - \langle \xi_1, t_1^{(2)} - t_1^{(1)} \rangle + \gamma_1 \).

Lemma 1. After the Read operation, \( \text{read}_0 + \text{read}_1 = D_{i^*} \).

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.

3.1.2 Update Operation

The goal is to add a value to the element at the target index of the database. Parties \( P_0 \) and \( P_1 \) hold \((M_0, \xi_0)\) and \((M_1, \xi_1)\) respectively. Their goal is to obtain shares of a new database, whose \( i^{th} \) value is updated by \( M \). A Write operation is a Read operation (with the output \( \text{read}_b \)) followed by an Update operation with \( M_b - \text{read}_b \).

Our Update operation works as follows. For \( b \in \{0, 1\} \), \( P_b \) simply locally sets \( D_b \leftarrow D_b + v_b^{(1)} \).

Lemma 2. After the Update operation, we have:

(i) \( D_0[i] + D_1[i] = D[i] \) for all \( i \neq i^* \), and

(ii) \( D_0[i^*] + D_1[i^*] = D[i^*] + M \).

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, 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 \((D_0, \xi_0, \tilde{D}_1)\) and \( P_1 \) holds \((D_1, \xi_1, \tilde{D}_0)\), 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 $\xi_0$ and their blinded versions $\tilde{\mathbf{D}}_{1-b}$ of the other party’s database share so that $\tilde{\mathbf{D}}_b'$ (held by $P_b$) equals $\mathbf{D}_b' + v_0$ (held by $P_b$). From the point of view of $P_1$, for example, we first observe that, the blinded shares, namely, $\mathbf{D}_0$ is wrong because $P_0$ updated $\mathbf{D}_0$ by $v_0$). However, $P_1$ holds $v_1^{(3)}$, which is equal to $-v_0^{(3)}$ at every location, except at $i^*$. Thus, $P_1$ can (almost) correct the blinded shares. In order to completely correct the blinded share $\mathbf{D}_0$, we exploit the fact that (i) $v_0^{(3)}[i^*] + v_1^{(3)}[i^*] = v_0^{(1)}[i^*] + v_1^{(1)}[i^*] = M$, and (ii) only $i^*$th location needs to be corrected. Therefore, we use another pair of vectors that reconstruct to $M \cdot \mathbf{e}_{i^*}$, namely $v_0^{(2)}$ and $v_1^{(2)}$. We have $P_0$ update the blind by $-v_0^{(2)}$ and $P_1$ do one more update of $\mathbf{D}_0$ by $v_1^{(2)}$. 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 $\xi_0' \leftarrow \xi_0 - v_0^{(3)}$ and $\xi_1' \leftarrow \xi_1 - v_1^{(3)}$ respectively.

2. $P_0$ and $P_1$ also update the blinded shares they received as $\mathbf{D}_1' \leftarrow \mathbf{D}_1 + v_0^{(1)}$ and $\mathbf{D}_0' \leftarrow \mathbf{D}_0 - v_1^{(1)}$ respectively.

3. $P_2$ updates the blinds as $\xi_0' \leftarrow \xi_0 - v_0^{(3)}$ and $\xi_1' \leftarrow \xi_1 - v_1^{(3)}$.

Lemma 3. For $b \in \{0, 1\}$, $\mathbf{D}_b' + \xi_0' = \tilde{\mathbf{D}}_b'$.

The proof of Lemma 3 appears in Appendix B. For the next READ operation DUORAM uses $(\xi_0', \mathbf{D}_1')$ and $(\xi_1', \mathbf{D}_0')$ as the blinds and blinded shares, thus avoiding FLORAM’s $O(\sqrt{n})$ amortized communication.

Remark 1. Since the computing parties need to hold the blinds and blinded shares (in addition to thier own shares of $\mathbf{D}$) the per-party storage requirement of 3P-DUORAM is three the database size.

4 3P-DUORAM: The Details

In this section, we fill in the gaps from the high-level discussion in Section 3.1. 3P-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 read or write into the database, thus postponing those decisions to the online phase.

DPFs with deferred final correction word. Before we proceed with the descriptions of the preprocessing and online phases, we consider the concept of DPFs with deferred final correction word, which replace the final correction word $f$ (for some as-yet-unknown target value) with the shares of a final correction word for the target value 0. Such DPFs were used in Pirsona [27] and are critical to the DUORAM Update protocol. More formally, in such DPFs, we replace the final correction word with $f'_0$ and $f'_1$ sent to $P_0$ and $P_1$, respectively, such that $f'_0 + f'_1 = -(v_0[1^*] + v_1[1^*])$, where $v_0$ and $v_1$ are evaluations of the DPF without applying a final correction word. We denote the DPFs with the final correction word shares as $\tilde{k}_b = (s_b^0, cw^0, \ldots, cw^b, f'_b)$.

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 $r_i$. The protocol begins with parties $P_0$ and $P_1$ selecting random indices $R_i[0]$ and $R_i[1]$ respectively. These values serve as XOR shares for the random value $r_i = R_i[0] \oplus R_i[1]$. The parties then use an MPC share conversion procedure to convert these XOR-shares to additive shares $r_0i$ and $r_1i$ respectively, such that $r_0i + r_1i = r_i$. 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 $r_0i$ and $r_1i$, to generate DPFs (and their evaluation) without the final correction word at the location $r_i$. The presentation of the protocol appears in Appendix C. The protocol results in $P_b$ holding $(\tilde{v}_0, t_0)$ such that, (i) $\tilde{v}_0[i] \oplus t_0[i] = 0$ for all $i \neq r_i$, and (ii) $t_0 \oplus t_1 = e_{r_i}$.

We interpret $\tilde{v}_0$ as a word vector and call it $v_0$, and interpret $\tilde{t}_i$ as a word vector, negate each element (recall all word operations are mod 2$^\nu$), and call it $t_1$. The final correction word shares can be locally computed by each party, and are defined as $f_0' = -\sum (v_0[i])$ and $f_1' = -\sum (v_1[i])$. Therefore, we have $v_0[r_i] + v_1[r_i] + f_0' + f_1' = 0$.

DUORAM also requires additive shares of the flag vectors rather than XOR shares. We next describe a procedure that, given the above $v_0$ and $f_0'$ values, converts the flag vectors $t_0$ and $t_1$ such that $t_0 \oplus t_1 = e_{r_i}$. into word vectors $v_0$ and $v_1$ such that $t_0 + t_1 = e_{r_i}$ (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 $\pm 1 \cdot e_{r_i}$. More specifically, if $t_0[r_i] = 1$ and $t_1[r_i] = 0$, these are additive shares of $e_{r_0}$ as
required, but if \( t_0[r_i] = 0 \) and \( t_1[r_i] = 1 \), these are additive shares of \(-e_{r_i}\). (\( t_0 \) and \( t_1 \) 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 word, 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 \( r_i \) as required, but at \( r_i \), the sum is 1± (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, \((\overline{k}^{(0)}, \overline{\xi}^{(0)}, \zeta^{(0)})\) and \((\overline{k}^{(1)}, \overline{\xi}^{(1)}, \zeta^{(1)})\) respectively.

2. \( P_0 \) sends \( \overline{k}^{(0)} \) to \( P_2 \), and \( P_1 \) sends \( \overline{k}^{(1)} \) 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 \((v_b^{(i)}, t_b^{(i)}, f_b^{(i)}, r_{i+b})\), for \( b \in \{0, 1\} \), \( i \in \{1, 2, 3\} \), and (ii) the auxiliary party \( P_2 \) holds \((v_0^{(i)}, t_0^{(j)}, f_0^{(j)}, t_1^{(j)})\).

The preprocessing for the READ protocol is almost the same, requiring three DPFs. For the READ protocol, however, the parties do not need to hold any \( v_b \) 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 \( i^* \) 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 \( e_{r_i} \) along with shares of \( r_i \). The parties first use the cyclic shift protocol to shift shares of \( e_{r_i} \) into shares of \( e_{r_i} \).

#### 4.2.1 Adjusting the Random DPFs

**Cyclic Shifts:** Postponing the decision of \textit{where to read or write}. The 3-party online phase begins with a cyclic shift protocol that adjusts the shares of the standard basis vector at a random location \( r_i \) to the shares of the standard basis vector at the target index \( i^* \). \( P_0 \) and \( P_1 \) hold \( i_0^* \) and \( i_1^* \) respectively, such that \( i^* = i_0^* + i_1^* \). They also hold \((r_{i_0}, v_0)\) and \((r_{i_1}, v_1)\) such that \( r_{i_0} = r_{i_1} + r_{i_1} \), and \( v_0 + v_1 = e_{r_{i_1}} \). Their goal is to get vectors \( v_0 \) and \( v_1 \) such that \( v_0 + v_1 = e_{r_{i_1}} \). They exchange \( i_{i_0}^* - r_{i_0}, r_{i_0}^* - r_{i_1}, \) and cyclic right shift \( v_{i_0} \) by \((i_0^* + i_1^* - r_{i_0} - r_{i_1})\). Similarly, \( P_2 \) receives \( i_{i_1}^* - r_{i_1} \) from \( P_b \) and can also compute the required offset \((i_0^* + i_1^* - r_{i_0} - r_{i_1})\).

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 \( r_i \) to \( i^* \), we use the 3-party protocol in Section 3.1.1. Figure 5 describes the 3-party online phase of the READ protocol. Protocol 1 summarizes the entire (preprocessing and online phases) DUORAM READ protocol. All array indices are taken mod \( n \) in Protocol 1.

#### 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
Protocol 1 3P-DUORAM READ Protocol. In the online phase, 
P₀ holds (D₀, ξ₀, D₁, i₀); P₁ holds (D₁, ξ₁, D₀, i₁); P₂ holds (ξ₀, ξ₁). After the end of the protocol, P₀ and P₁ get read₀ and read₁ respectively, such that read₀ + read₁ = D[D[i₀] + i₁].

Preprocessing Phase:
1: P₀ and P₁ pick random index shares r₀ and r₁ respectively, and with the aid of P₂, use the (2 + 1)-MPC protocol to generate three DPFs, (k₀, k₁, k₂, r₀) and (k₀, k₁, k₂, r₁), all with index r = r₀ + r₁.
2: P₀ sends r₀ to P₂, and P₁ sends r₁ to P₂.
3: The parties evaluate the DPFs to get XOR-shared flag bits, P₀ and P₁ get t₀ and t₁ respectively, for k ∈ {1, 2, 3}.
P₂ gets (t₀, t₁).
4: They convert the XOR-shared flags t to additive shares t.

Online Phase:
5: P₀ and P₁ exchange (i₀ − r₀) and (i₁ − r₁). In the same round of communication, P₀ and P₁ send (i₀ − r₀) and (i₁ − r₁) respectively to P₂.
6: P₀, P₁, and P₂ reconstruct S ← i₀ − r₀ + i₁ − r₁.
7: ∀i, P₀ computes t₀[i] ← t₀[i] − S for k ∈ {1, 2}; P₁ computes t₁[i] ← t₁[i] − S for k ∈ {1, 2}; P₂ computes t₂[i] ← t₂[i] − S for k ∈ {1, 2}.
8: P₂ selects ρ ∈ {0, 1} w uniformly at random and sends y₀ ← −(ξ₀, t₀, ρ) to P₀ and y₁ ← −(ξ₁, t₁, ρ) to P₁.
9: P₀ outputs read₀ ← (D₀ + D₁, t₀) − (ξ₀, t₀ − t₀ + y₀).
10: P₁ outputs read₁ ← (D₁ + D₀, t₁) − (ξ₁, t₁ − t₁ + y₁).

the database D. The two parties also hold (i) M₀ and M₁, the shares of the target value, and (ii) DPFs k₀ and k₁ at i and their evolutions (v₀, v₁). Their goal is to add M = M₀ + M₁ to the value at the target index i* of the database D. The idea is that the parties P₀ exchange M₀ + f₀ to reconstruct M − (v₀[i*] + v₁[i*]). The reconstruction serves as the new final correction word to write M in the desired location.

P₀ and P₁ then use the technique of Figure 4 to update their own blind and their copy of the other’s blinded database share; P₂ similarly updates its copy of P₀’s and P₁’s blinds. We present the complete online phase of the update protocol, including both sets of updates, in Figure 6. The entire UPDATE protocol (preprocessing and online phases) is summarized in Protocol 2. All array indices are taken mod n in Protocol 2.

5 2P-DUORAM

This section presents the 2-party instantiation of DUORAM. The update protocol is (nearly) the same as the one in 3P-DUORAM, with the number of computational symmetric

Protocol 2 3P-DUORAM UPDATE Protocol. In the online phase, P₀ holds (D₀, ξ₀, D₁, i₀, M₀); P₁ holds D₁, (ξ₁, D₀, i₁, M₁); P₂ holds (ξ₀, ξ₁). After the end of the protocol, P₀ gets (D₀, ξ₀, D₁), P₁ gets (D₁, ξ₁, D₀), and P₂ gets (ξ₀, ξ₁). All with index r = r₀ + r₁. In the process, P₀ and P₁ learn shares f₀ and f₁ respectively of the final correction words (k ∈ {1, 2, 3}).

Preprocessing Phase:
1: P₀ and P₁ pick random index shares r₀ and r₁ respectively, and with the aid of P₂, use the (2 + 1)-MPC protocol to generate three DPFs, (k₀, k₁, k₂, r₀) and (k₀, k₁, k₂, r₁), all with index r = r₀ + r₁.
2: P₀ sends r₀ to P₂, and P₁ sends r₁ to P₂.
3: The parties evaluate the DPFs to get XOR-shared flag bits, P₀ and P₁ get t₀ and t₁ respectively, for k ∈ {1, 2, 3}.
P₂ gets (t₀, t₁).
4: They convert the XOR-shared flags t and leaves v to additive shares t and v.

Online Phase:
5: P₀ and P₁ exchange (i₀ + r₀, M₀ + f₀, (M₀ + f₀)) and (i₁ + r₁, M₁ + f₁, (M₁ + f₁)) and also send those values to P₂. P₀ and P₁ also exchange (M₀ + f₀, (M₁ + f₁)) but do not send those values to P₂.
6: P₀, P₁, P₂ reconstruct f₀ ← (M₀ + f₀ + M₁ + f₁) for k ∈ {2, 3}; P₀, P₁ also do so for k = 1.
7: P₀, P₁, P₂ reconstruct S ← i₀ + r₀ + i₁ + r₁.
8: P₀ updates ∀i:
   D₀[i] ← D₀[i] + (v₀[i] + f₀[i] − t₀[i]),
   ξ₀[i] ← ξ₀[i] + (v₀[i] − f₀[i] − t₀[i]),
   D₀[i] ← D₀[i] + (v₀[i] − f₀[i] − t₀[i]),
   ξ₀[i] ← ξ₀[i] + (v₀[i] − f₀[i] − t₀[i]).
9: P₁ updates ∀i:
   D₁[i] ← D₁[i] + (v₁[i] + f₁[i] − t₁[i]),
   ξ₁[i] ← ξ₁[i] + (v₁[i] − f₁[i] − t₁[i]),
   D₁[i] ← D₁[i] + (v₁[i] − f₁[i] − t₁[i]),
   ξ₁[i] ← ξ₁[i] + (v₁[i] − f₁[i] − t₁[i]).
10: P₂ updates ξ₀ and ξ₁ to ξ₀ and ξ₁ as P₀ and P₁ do above.

PIR (CSPIR) protocol, resulting in communication logarithmic in the database size (beating FLORAM’s O(√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.

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 2P-DUORAM relies on Computational Symmetric PIR (CSPIR), which works as follows. The parties have precomputed and exchanged CSPIR queries for lookups at random indices $r_{10}$ and $r_{11}$ respectively. Note that the encrypted queries only depend on the (random) index being looked up and on the size of the database, and not on the contents of the database, so those can be computed and exchanged during preprocessing. They also have shares $\lambda_0$ and $\lambda_1$ of the target index $i^*$. For $b \in \{0, 1\}$:

1. $P_b$ sends $(i^* - r_{1b})$ to $P_{1-b}$.
2. $P_b$ blinds each of the elements of $D_b$ with a random value $r_j \in \{0, 1\}^n$, and rotates the resulting blinded vector by $i^*_b + (i^*_1 - r_{1b})$. In other words, $P_b$ computes $D'_b[i] = D_b[i + i^*_b + (i^*_1 - r_{1b})] + r_b$.
3. $P_{1-b}$ computes a response to $P_b$'s precomputed CSPIR query with index $r_{1b}$ on $D'_b$, and sends the result to $P_b$, who recovers $c_b = D_{1-b}[i^*] + r_{1-b}$.
4. $P_b$ outputs: $\text{read}_b \leftarrow c_b - r_b$.

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

The proof of Lemma 4 can be found in Appendix B. Our implementation uses the SPIRAL CSPIR protocol by Menon and Wu [23]. We note that CSPIR 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 protocol, but without the additional REFRESHBLINDS operation. Thus, somewhat counterintuitively, the online UPDATE phase of 2P-DUORAM is cheaper than that of 3P-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 2: Comparing computation, bandwidth, and the number of sequential 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 \( w \geq \log n \).

<table>
<thead>
<tr>
<th>Operation</th>
<th>3P-DUORAM</th>
<th>2P-DUORAM</th>
<th>FLORAM</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp; Ind Reads</td>
<td>( O(kn) + O(kn) )</td>
<td>( O(kw \log n + 2 \cdot k \cdot w) + 2 \cdot k \cdot w )</td>
<td>( O(\log n) + 2 )</td>
</tr>
<tr>
<td>&amp; Dep Reads</td>
<td>( O(kn) + O(kw) )</td>
<td>( O(kw \log n + 2 \cdot k \cdot w) + 2 \cdot k \cdot w )</td>
<td>( O(\log n) + 2 \cdot k \cdot w )</td>
</tr>
<tr>
<td>&amp; Writes</td>
<td>( O(kn) + O(kn) )</td>
<td>( O(kw \log n + 9 \cdot k \cdot w) + 2 \cdot k \cdot w )</td>
<td>( O(\log n) + 2 \cdot k \cdot w )</td>
</tr>
<tr>
<td>Interleaved</td>
<td>( O(kn) + O(kn) )</td>
<td>( O(kw \log n + 11 \cdot k \cdot w) + 5 \cdot k \cdot w )</td>
<td>( O(\log n) + 3 \cdot k \cdot w )</td>
</tr>
</tbody>
</table>

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 2.\(^4\) Note that Table 2 shows the number of sequential messages rather than the number of rounds. We say that a protocol uses \( q \) sequential messages if the time spent on Internet latency is \( q \) times the one-way latency. This notion differs from rounds, where a round requires each party to wait for a response to their message before sending the next message. Thus, \( q \) rounds would mean the time spent on Internet latency is \( 2q \) times the one-way latency. Observe that the online communication cost of 3P-DUORAM is constant (for constant-sized words; it is linear in the word length). 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++.\(^5\) Our implementation uses Boost.Asio v1.18.1 for asynchronous communication. 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.

For our experimental evaluation, we compare DUORAM against the two works from Table 1 with available implementations, namely FLORAM\(^6\) and Jarecki and Wei’s [19] 3P-Circuit ORAM.\(^7\)

6.2.1 Head-to-Head Comparison with FLORAM

In this section, we do a head-to-head comparison of FLORAM with DUORAM, comparing the (i) wall-clock time, and (ii) bandwidth consumption. The standard latency we use is 30 ms,\(^8\) and the standard bandwidth 100 Mbit/s. 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 from \( 2^{16} \) to \( 2^{26} \), while keeping the latency and throughput constant at 30 ms and 100 Mbit/s, respectively, (ii) varying the throughput while keeping the number of items constant at \( 2^{20} \), and the network latency at 30 ms, and (iii) varying the network latency while keeping the number of items at \( 2^{20} \) and the throughput at 100 Mbit/s. We omit 3P-Circuit ORAM from these plots because FLORAM outperforms it in these settings.

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 11 and 12 in the extended version of this paper [28, App.E]. For our standard network settings, we see that 2P-DUORAM is faster than FLORAM until the database size reaches somewhere between \( 2^{22} \) and \( 2^{24} \) items; at this point, the linear SPIR computation of 2P-DUORAM starts exceeding the logarithmic cost of the rest of the protocol. On the other hand, 3P-DUORAM consistently performs better than FLORAM for all database sizes. Decreasing the bandwidth capacity has a minimal impact on the performance of DUORAM because it

\(^4\) Code retrieved from https://gitlab.com/neucrypt/floram/.
\(^6\) A value chosen from the low end of one-way latency values from https://www.cloudflare.co/grid. Note that low latencies benefit FLORAM much more than DUORAM, as we will soon see.
This is because FLORAM set the latency to 100 ms. Even under these extreme network conditions, the bandwidth capacity to as low as 1 Mbit per second and the bandwidth consumption in the case of dependent RORAM operations comes in the initialization phase. In the case of the interleaved operations, the online phase of 2P-D ORAM requires about 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 of 2P-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 2P-DUORAM took about 10 seconds to do one read operation (including 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 4 in the extended version of this paper [28, App.E] gives the detailed results of this experiment.

6.2.2 Internet vs. Local Network Conditions

In this section, we perform a head-to-head comparison of 3P-Circuit ORAM and FLORAM with DUORAM for different network conditions. Figure 9 compares DUORAM against 3P-Circuit ORAM and FLORAM for (i) 100 Mbit/s bandwidth and a latency of 30 ms, and (ii) 100 Gbit/s bandwidth, and no additional latency set (we measure a 30 µs latency). The former is our standard realistic Internet setting, while the latter is intended to model the parties being colocated in the same datacentre, if not the same rack. The latter setting is the least favorable for DUORAM, as DUORAM optimizes the parameters (round complexity and bandwidth) that are not relevant for colocated servers; in particular, whereas DUORAM requires only two sequential messages per READ operation, we find that Jarecki and Wei’s 3P-Circuit ORAM needs about $101 \log n + 42$, which is in the hundreds for these database sizes. We observe that to do 128 READ operations in the colocated setting, 3P-DUORAM performs better than 3P-Circuit ORAM.
operation, which takes \(O\) bandwidth consumption overtakes 3P-Circuit ORAM. Figure 9 that, to do 128 dependent reads, the bandwidth consumption of 3P-Circuit ORAM is higher than F\(LORAM\)’s. The y-axis in all three plots is log-scaled.

For the non-colocated network setting, we observe that 3P-Circuit ORAM consistently performs better than FLORAM until the database size of nearly \(2^{32}\) items. When we compare DUORAM with 3P-Circuit ORAM, we observe that somewhere between database sizes of \(2^{28}\) and \(2^{30}\) (for total time), or between \(2^{30}\) and \(2^{32}\) (for online time), 3P-Circuit ORAM starts performing better than DUORAM. Recall from Table 1 that Jarecki and Wei’s protocol has a \(O(\lg n)\) computation cost. At these sizes, the linear computation cost of DUORAM starts dominating its low roundtrip cost.

6.2.3 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 10 shows how the performance of DUORAM and FLORAM vary as we increase the number of cores. This section omits a comparison with 3P-Circuit ORAM since their implementation does not support multithreading. We observe that, as we increase the number of cores, 2P-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\(^9\) for a long-term Amazon EC2 instance: $0.0195/CPU-hour and $0.09/GB of outbound traffic.\(^10\) Table 3 compares the cost in USD to do ORAM operations in DUORAM and FLORAM.

\(^9\)https://aws.amazon.com/ec2/pricing/on-demand/

\(^{10}\)These are the same values used by SPRIAL [23] for their dollar costs.
All the costs in costs in Table 3 in micro-dollars, denoted as µ$ (1 micro-dollar = 10⁻⁶ USD).

7 Related Work

In this section, we briefly survey some related work from the ORAM and DORAM literature. Goldreich and Ostrovsky [14] first introduced ORAM in a general client-server context and proposed the classic “square-root ORAM” protocol, so called because it introduces overhead square root in the database size. The ORAM problem they considered involves a client who wishes to perform some computation over a memory of size n held by an untrusted server, while hiding its access patterns to the memory. Over the subsequent two decades, many works [2, 6, 7, 15, 16, 20, 25, 26, 31, 32] addressed the same basic problem but with progressively lower communication overhead between the client and the server.

Beyond the original client-server model, ORAM can be useful in the context of secure computation. In such a setting, the client operations are implemented as a circuit. Wang, Chan, and Shi’s work [30] introduced a tree-based ORAM that optimizes the circuit size. Gentry, Goldman, Halevi, Julia, Raykova, and Wichs [11] improve upon the Tree ORAM [25]. 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 secure computation ORAM that is a variant of the binary tree ORAM by Shi et al. [25]. Zahur, Wang, Raykova, Gascon, Doerner, Evans, and Katz [33] revisited square-root ORAM. Their work showed that relaxing the asymptotic bounds of access complexity would produce smaller circuits. They proposed an ORAM scheme with cost O(√n · lg n) 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, Kushile-vitz, and Ostrovsky [5]. However, they use 3-party DPFs that have size O(√n). Hamlin and Varia [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(√n · 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 out in a complicated data structure, which is different from DUORAM, where the memory is laid out in an array. 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 pre-processing phase, before the target point or message is known, both of which vastly reduce the online cost of the protocol.

Acknowledgements

We thank the anonymous reviewers and shepherd for their helpful comments in improving this paper. This research was undertaken, in part, thanks to funding from the Canada Research Chairs program, Cisco Research, and an NSERC Discovery Grant. This work benefited from the use of the CrySP RIPPLE Facility at the University of Waterloo.

References


This section presents two methods to generate multiplication triples that were discussed in Section 2.2.1. We use XOR-shared AND 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 AND 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 AND 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 \((x_0 \wedge Y_1) \oplus T_1 = (x_0 \wedge Y_1) \oplus T\).
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 \((x_1 \wedge Y_0) \oplus T_0 = (x_1 \wedge Y_0) \oplus T\).

**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.**

\[
\text{read}_0 + \text{read}_1 = (D_0 + \tilde{D}_1, t^{(0)}_1) - (\zeta_0, t^{(0)}_0 - t^{(0)}_1) + y_0 + (D_1 + \tilde{D}_0, t^{(1)}_0) - (\zeta_1, t^{(1)}_0 - t^{(1)}_1) + y_1
\]

\[
= (D_0 + (D_1 + \zeta_1), t^{(0)}_0 - t^{(0)}_1) + y_0 + (D_1 + (D_0 + \zeta_0), t^{(1)}_0 - t^{(1)}_1) + y_1
\]

\[
= (D, t^{(0)}_0) + (\zeta_1, t^{(0)}_1 - t^{(0)}_1) + (\zeta_0, t^{(1)}_1 - t^{(1)}_1) + y_0 + (D, t^{(1)}_0) - (\zeta_1, t^{(1)}_0 - t^{(1)}_1) + y_1
\]

\[
= (D, e^{(1)}_0) + (\zeta_1, e^{(0)}_1) + (\zeta_0, e^{(0)}_1)
\]

\[
- (\zeta_0, t^{(0)}_0 - t^{(0)}_1) + y_0 + (D, e^{(1)}_1) + (\zeta_1, t^{(1)}_1 - t^{(1)}_1) + y_1
\]

\[
= (D, e^{(1)}_1) + (\zeta_1, e^{(1)}_1) + (\zeta_0, e^{(1)}_1) - (\zeta_1, e^{(1)}_1) - (\zeta_0, e^{(1)}_1)
\]

\[
= (D, e^{(1)}_1)
\]

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

**Proof of Lemma 3.**

\[
\tilde{D}_0 = \tilde{D}_0 + v^{(2)}_1 - v^{(1)}_1
\]

\[
= (D_0 + \zeta_0) + v^{(2)}_1 - v^{(1)}_1
\]

\[
= (D_0 + \zeta_0) + (M \cdot e^{(1)}_1 - v^{(2)}_0) - (M \cdot e^{(1)}_1 - v^{(1)}_0)
\]

\[
= (D_0 + v^{(1)}_0 + (\zeta_0 - v^{(2)}_0)
\]

and similarly for \(\tilde{D}_1\).

Finally, we will prove the correctness of the 2P-DUORAM READ protocol.
Proof of Lemma 4.

(read0 + read1) = c0 - r0 + c1 - r1
= D1[1] + r1 - r0 + D0[1] + r0 - r1
= D1[1] + D0[1]
= D[1]

\[ \square \]

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, \( \hat{i} \). Represent \( \hat{i} \) as a binary bit vector \( \hat{i}^* \). The parties \( P_0 \) and \( P_1 \) start with XOR shares \( \hat{i}^*_0 \) and \( \hat{i}^*_1 \) of \( \hat{i}^* \), and create random seeds \( v_0^\dagger \in \{0,1\}^\lambda \) and \( v_1^\dagger \in \{0,1\}^\lambda \) respectively as use of the roots of binary trees. They use a length-PRG to construct the remainder of the trees, as outlined below. We denote by \( v_{\hat{i},\ell} \) the nodes at level \( \ell \), and by \( t_{\hat{i},\ell} \) the flags at level \( \ell \), 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 lsb(\( v_0^\dagger \)) \( \leftarrow 0 \) and lsb(\( v_1^\dagger \)) \( \leftarrow 1 \). Set \( t_0[0][0] \leftarrow \) lsb(\( v_0^\dagger \)) and \( t_1[0][0] \leftarrow \) lsb(\( v_1^\dagger \)).

For each layer \( \ell \) 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 at this level. The left and right children of node \( i \) at this layer are denoted as \( v_{\hat{i},\ell}^{(i)} \) and \( v_{\hat{i},\ell}^{(i+1)} \) respectively. Therefore, we have \( v_{\hat{i},\ell+1}[2 \cdot i] = v_{\hat{i},\ell}^{(i)} \) and \( v_{\hat{i},\ell+1}[2 \cdot i + 1] = v_{\hat{i},\ell}^{(i+1)} \).

2. For \( b \in \{0,1\} \), \( P_b \) computes \( L_b \leftarrow \bigoplus_{j=0}^{2^{\ell-1}} v_{\hat{i},\ell}^{(i)} \) and \( R_b \leftarrow \bigoplus_{j=0}^{2^{\ell-1}} v_{\hat{i},\ell}^{(i+1)} \).

3. \( P_0 \) and \( P_1 \) use an MPC AND protocol (using the AND triple generation described in Appendix A) to compute the correction word \( cw^{(i,i+1)} \leftarrow \left( (\hat{i}^*_0[\ell] \oplus \hat{i}^*_1[\ell]) \land (L_0 \oplus L_1) \right) \land \left( (\hat{i}^*_0[\ell] \oplus \hat{i}^*_1[\ell]) \land (R_0 \oplus R_1) \right) \).

4. \( P_b \) computes \( cw_b^{(i,i)} \leftarrow \) lsb(\( L_b \)) \( \oplus \hat{i}^*_b[\ell] \) and \( cw_b^{(i,i+1)} \leftarrow \) lsb(\( R_b \)) \( \oplus \hat{i}^*_b[\ell] \), and exchanges those values with the other party; the parties then both compute \( cw_\ell_L \leftarrow cw_b^{(i,i)} \oplus cw_b^{(i,i+1)} \oplus 1 \) and \( cw_\ell_R \leftarrow cw_b^{(i,i)} \oplus cw_b^{(i,i+1)} \).

5. \( P_b \) computes \( t_{\hat{i},\ell+1}[2 \cdot i] \leftarrow v_{\hat{i},\ell+1}[2 \cdot i] \oplus (t_{\hat{i},\ell}[i] \cdot cw_\ell_L) \) and \( t_{\hat{i},\ell+1}[2 \cdot i + 1] \leftarrow v_{\hat{i},\ell+1}[2 \cdot i + 1] \oplus (t_{\hat{i},\ell}[i] \cdot cw_\ell_R) \), for all \( i \in [0, 2^\ell] \).

6. \( P_b \) updates \( v_{\hat{i},\ell+1}[2 \cdot i] \leftarrow v_{\hat{i},\ell+1}[2 \cdot i] \oplus (t_{\hat{i},\ell+1}[2 \cdot i] \cdot cw_{\hat{i},\ell}) \) and \( v_{\hat{i},\ell+1}[2 \cdot i + 1] \leftarrow v_{\hat{i},\ell+1}[2 \cdot i + 1] \oplus (t_{\hat{i},\ell+1}[2 \cdot i + 1] \cdot cw_{\hat{i},\ell}) \), for all \( i \in [0, 2^\ell] \).

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 \( (v_0, t_0, f_0) \) and \( P_1 \) holds \( (v_1, t_1, f_1) \) such that \( t_0 \oplus t_1 = e_{\hat{x}_1} \) and \( v_0 + v_1 = -(f_0 + f_1) \cdot e_{\hat{x}_1} \). Their goal is to end up with vectors \( t_0 \) and \( t_1 \) such that \( t_0 + t_1 = e_{\hat{x}_2} \). An important point to note is that, in this case, \( t_0 \) and the pair \( (v_1, f_0) \) 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 \( t_0[i] \leftarrow (t_0[i]) \) for all \( i \), and \( P_1 \) computes \( t_1[i] \leftarrow -(t_1[i]) \) for all \( i \).

2. For \( b \in \{0,1\} \), \( P_b \) computes \( pm_b \leftarrow \Sigma_i t_b[i] \).

3. For \( b \in \{0,1\} \), \( P_b \) selects a random word \( r_b \) to blind \( pm_b \), and the \( P_b \) exchange \( pm_b + r_b \).

4. For \( b \in \{0,1\} \), \( P_b \) updates \( t_b \leftarrow t_b \cdot ((pm_{1-b} + r_{1-b}) + pm_b + r_b) \).

5. The parties use MPC to compute shares \( f_0' \) and \( f_1' \) of \( (f_0 + f_1) \cdot (pm_0 + pm_1) \).

6. The parties then compute \( f_0'' \leftarrow f_0' + r_0 \) and \( f_1'' \leftarrow f_1' + r_1 \) respectively, and exchange them to reconstruct \( f'' = f_0'' + f_1'' \).

7. For \( b \in \{0,1\} \), \( P_b \) updates \( t_b \leftarrow t_b - v_b - (t_b \cdot f'') \).

Lemma 5. After running the above protocol, \( t_0 + t_1 = \hat{t}_0 \oplus \hat{t}_1 = e_{\hat{x}_2} \).

Proof. Denote \( pm = pm_0 + pm_1 \). First observe that \( \hat{t}_0[i] \leftarrow \hat{t}_0[i] \) for \( i \neq r_1 \), so \( pm = \hat{t}_0[r_1] \). Then \( \hat{t}_0 + \hat{t}_1 = (pm \cdot e_{\hat{x}_2}) \). Also note that \( \hat{t}_0 + \hat{t}_1 = (pm \cdot e_{\hat{x}_1}) \). Finally, recall that \( v_0 + v_1 = -(f_0 + f_1) \cdot e_{\hat{x}_1} \). Also \( f = f \cdot pm \) and \( f'' = f + r = f \cdot pm + r \). Therefore, \( (1 + pm \cdot r) \cdot f'' = e_{\hat{x}_2} \cdot ((1 + pm \cdot r)) \).