Multi-party Private Function Evaluation for RAM

Keyu Ji, Bingsheng Zhang, Tianpei Lu, and Kui Ren

Abstract

Private function evaluation (PFE) is a special type of MPC protocols that, in addition to the input privacy, can preserve the function privacy. In this work, we propose a PFE scheme for RAM. In particular, we first design an efficient 4-server distributed ORAM scheme with amortized communication $O(\log n)$ per access (both reading and writing). We then simulate a RISC RAM machine over the MPC platform, hiding (i) the memory access pattern, (ii) the machine state (including registers, program counter, condition flag, etc.), and (iii) the executed instructions. Our scheme can naturally support a simplified TinyRAM instruction set; if a public RAM program $P$ with given inputs $x$ needs to execute $z$ instruction cycles, our PFE scheme is able to securely evaluate $P(x)$ on private $P$ and $x$ within $5z + 1$ online rounds. We prototype and benchmark our system for set intersection, binary search, quicksort, and heapsort algorithms. For instance, to obliviously perform the binary search algorithm on a $10^4$ array takes 9.21s with function privacy.

1 Introduction

In a secure private function evaluation (PFE) protocol, the MPC players want to collaboratively evaluate a private function without revealing their private inputs to each other. The main difference between PFE and the conventional MPC protocols is that in addition to the input privacy, PFE also needs to preserve the function privacy. Such PFE schemes can be very beneficial for many security sensitive applications, such as software diagnostic [1], medical applications [2], and intrusion detection systems [3]. There, the service providers may require confidentiality of their specific algorithms during the MPC evaluation.

In the past decades, how to design an efficient PFE protocol has been extensive studied in the literature. Early works [4–6] deal with PFE problem by secure computation of universal circuit. A universal circuit $F$ is a circuit that takes as input a description of a circuit $f$ (with at most $z$ gates) and an input $x$, runs $f$ on $x$ and outputs $f(x)$. But transforming a boolean circuit to a universal circuit would introduce a poly-logarithmic overhead; it is even worse for arithmetic circuits (e.g. as high as $O(z^5)$ [7]). Recently, many works [8–14] are devoted to reduce the overhead of circuit size. We highlight that those previous works are all limited by the fact that the private function must be represented as a (arithmetic) circuit. Hereby, we ask the following challenging question:

Is it possible to design an efficient PFE MPC platform that can naturally support random-access machines (RAM) programs?

In this work, we answer this question affirmatively.

1.1 Our Approach

We reduce this problem to simulating a RAM machine which can evaluate a secret shared program while hiding its execution pattern. That is, (i) the memory access pattern, (ii) the machine state (including registers, program counter, condition flag, etc.) and (iii) the executed instructions of the RAM machine all need to be protected from the MPC players.

For hiding the data-dependent access pattern of random-access memory, it is natural to use Oblivious RAM (ORAM) technique [15]. ORAM allows a secure CPU to access the untrusted memory without revealing its access patterns. There are many works [16–24] devoted to reduce the asymptotic ORAM overhead as well as practical efficiency. On the other hand, conventional ORAM schemes are typically not MPC-friendly, even those with multiple non-colluding servers [25–28], because they all require a trusted client who knows the queries to perform complex computation. Recent ORAM
Table 1: Round and communication complexities of distributed ORAM schemes: $n$ is the database size; $\ell$ is the record size; $\lambda, \sigma$ denote the cryptographic and statistical security parameters; $d$ denotes the AND depth of the circuit.

<table>
<thead>
<tr>
<th>Scheme</th>
<th>Rounds</th>
<th>Communication</th>
</tr>
</thead>
<tbody>
<tr>
<td>Circuit ORAM [31]</td>
<td>$O(\log n)$</td>
<td>$O(\lambda \log^2 n + \lambda \ell \log n)$</td>
</tr>
<tr>
<td>Sqrt-ORAM [32]</td>
<td>$O(\log n)$</td>
<td>$O(\lambda \sqrt{n \log^3 n})$</td>
</tr>
<tr>
<td>Floram [34]</td>
<td>$O(\log n)$</td>
<td>$O(\sqrt{n \log n})$</td>
</tr>
<tr>
<td>FJKW15 [30]</td>
<td>$O(\log n)$</td>
<td>$O(\lambda \sigma \log^3 n + \ell \log n)$</td>
</tr>
<tr>
<td>JW18 [33]</td>
<td>$O(\log n)$</td>
<td>$O(\lambda \log^2 n + \ell \log n)$</td>
</tr>
<tr>
<td>BKKO20 [35]</td>
<td>6</td>
<td>$O((\lambda + \ell) \sqrt{n})$</td>
</tr>
<tr>
<td>HNO21 [36]</td>
<td>$O(d)$</td>
<td>$O((\lambda + \ell) \log n)$</td>
</tr>
<tr>
<td>Ours (offline+online)</td>
<td>$O(\log n) + 1$</td>
<td>$O(\lambda \log n) + O(\log n + \ell)$</td>
</tr>
</tbody>
</table>

researches focus on designing distributed ORAM schemes [29–36] without trusted clients, which are tailor-made for the MPC scenarios. To the best of our knowledge, Floram [34] is the best practical distributed ORAM scheme in the literature. In Floram, the private data are stored in both the encrypted form (read-only) and the XOR-shared form (write-only). Based on distributed point function (DPF) [37] technique, it takes $O(\log n)$ communication to obliviously fetch an item from the read-only memory or obliviously write an item to the write-only memory, where $n$ is the size of database. But in order to achieve simultaneous read and write capabilities, Floram uses the stash-and-refresh technique, which requires amortized refreshing communication $O(\sqrt{n \ell \log n})$. Therefore, it wouldn’t work well in our simulated RAM machine with frequent and alternating memory reading and writing. To address this problem, we construct a 4-party efficient distributed ORAM in offline/online model, which is inspired by other DPF-based (distributed) ORAM works [27, 28, 35]. In our scheme, the database $x$ is replicated shared among the 4 MPC players, i.e., $x := x^{(1)} + x^{(2)} = x^{(2)} + x^{(4)}$, such that the players $P_1$ and $P_2$ (P3 and P4) hold the same share. Both the oblivious reading and the oblivious writing require $O(\log n)$ communication and 1 online round. Table 1 compares the access between our distributed ORAM scheme and the related works. Furthermore, we have the cheapest initialization with only $4n\ell$-bit communication for building replicated shares.

For hiding the machine state and instruction execution, we propose a MPC platform to simulate a RISC machine which is a simplified version of TinyRAM [38], using our distributed ORAM scheme as a building block. Our system can obliviously evaluate a TinyRAM program without revealing each operator and its operands. In particular, an oblivious instruction cycle consists of four phases: (i) instruction reading, (ii) instruction decoding, (iii) operation evaluation, and (iv) result writing. Each phase is implemented by 4-party computation protocols in the online/offline mode, and the whole oblivious instruction cycle requires 5 online rounds. More specifically, in the instruction reading phase, the MPC players jointly fetch the current instruction in the shared form using distributed ORAM. After that, the MPC players securely decode the instruction to load the private arguments and determine the secret operator by DPF. Both the instruction reading phase and the decoding phase costs one online round. During the operation evaluation phase, we evaluate all possible operations in parallel via MPC to obtain a series of replicated shared results. This phase requires three online rounds. In the result writing phase, the MPC players obliviously select the right result according to the shared operator, and then obliviously write the result into the destination register using our distributed ORAM. Although the oblivious writing needs an online communication round, in practice, we perform this operation in the same round as the instruction reading phase of the next cycle.

Our system can naturally support control flow, loops, subroutines, and recursion, etc. In more concrete terms, we allow the early termination and branching without revealing the current conditional values, and thus avoid the expensive transformation cost from the RAM program into straight-line code for secure computation. For efficiency, our system does not hide the overall running time. That is, if a public RAM program $P$ with given inputs $x$ needs to execute $z$ instruction cycles, our system is able to securely evaluate $P(x)$ on private $P$ and $x$ within $5z + 1$ online rounds. Note that our system is different from the prior CPU emulation works [39, 40] which assume the program are known to parties. Although another work [41] and the basic scheme in [39] can hide the program, their efficiency is much worse than ours as shown in Sec. 5, where we prototype and benchmark our system.
2 Preliminaries

2.1 Notations

Throughout this paper, we use the following notations and terminologies. Let \( \lambda \in \mathbb{Z} \) be the security parameter. Let \( \{a, b\} \) denote the set \( \{a, a + 1, \ldots, b\} \), \([b]\) stands for \([1, b]\) and \(\mathbb{Z}_n\) stands for \([0, n - 1]\). Denote a value \( x \) indexed by a label \( b \) as \( x(b) \), while \( x^r \) means the value of \( x \) power of \( b \). When \( S \) is a set, \( s \leftarrow S \) stands for sampling \( s \) uniformly at random from \( S \). When \( f \) is an algorithm, \( y \leftarrow f(x) \) stands for running \( f \) on input \( x \) and getting the output \( y \).

2.2 Secure Multiparty Computation

Secure multiparty computation (MPC) allows \( n \) mutually suspicious players to jointly evaluate a function \( F \) of their private inputs without revealing additional information beyond the output of \( F \). Our security model for MPC protocols is based on the Universal Composibility (UC) framework [42]. Our model assumes synchronous communication over secure point-to-point channels. Informally, we say that a protocol \( \Pi \) UC-realizes \( F \) with \( t\)-privacy, if it satisfies that for every semi-honest PPT adversary \( \mathcal{A} \) attacking an execution of \( \Pi \) and statically corrupting most \( t \) players, there is a simulator \( \mathcal{S} \) attacking the ideal process that uses \( F \) (by corrupting the same set of players), such that, the executions of \( \Pi \) with \( \mathcal{A} \) and that of \( F \) with \( \mathcal{S} \) are indistinguishable to any network execution environment \( \mathcal{Z} \).

2.3 Secret Sharing

Secret sharing is an essential primitive of MPC protocols. In this paper, we use the following secret sharing schemes and terminologies.

**Arithmetic Secret Sharing.** Denote a \((2, 2)\)-additive secret sharing in \( \mathbb{Z}_n \) by \( \{x\} := \{x^{(1)}, x^{(2)}\} \), where \( x^{(1)} + x^{(2)} = x \pmod{n} \). Denote a \((4, 4)\)-additive secret sharing in \( \mathbb{Z}_n \) by \( \{x\} := \{x^{(1)}, x^{(2)}, x^{(3)}, x^{(4)}\} \), where \( x^{(1)} + x^{(2)} + x^{(3)} + x^{(4)} = x \pmod{n} \). Denote a replicated secret sharing in \( \mathbb{Z}_n \) by \( \{x\}_{\text{rep}} := \{x^{(1)}, x^{(2)}, x^{(3)}, x^{(4)}\} \), where \( x^{(1)} = x^{(2)} = x^{(3)} = x^{(4)} \) and \( x = x^{(1)} + x^{(2)} + x^{(3)} + x^{(4)} \pmod{n} \). Over the ring \( \mathbb{Z}_n \), when \( P_1, P_2 \) share \([x]\) and \( P_1, \ldots, P_d \) share \([y]\), \([x]+[y]\) the operation yields the result \([x+y]\) as follows: using the additional shares of \((0)\), the player \( P_j, j \in \{1, 2\} \) sets \((x+y)^{(j)} := x^{(j)} + 0^{(j)} + y^{(j)} \pmod{n} \), and the player \( P_j, j \in \{3, 4\} \) sets \((x+y)^{(j)} := 0^{(j)} + y^{(j)} \pmod{n} \). Note that the MPC players can use the zero-sharing protocol proposed by [43] to non-interactively construct \((0)\) after the initialization of seeds.

**Yao’s Secret Sharing.** Denote a Yao’s secret sharing of a bit \( x \in \{0, 1\} \) by \( \{x\}^\mathcal{Y} := \{k_0, k_x\} \), where two labels \( k_0, k_1 \in \{0, 1\}^\lambda \) represent 0, 1 respectively, \( k_0 \) is the zero-label held by garbler and \( k_x \) is the real-label held by evaluator. Abusing notation, we denote a Yao’s secret sharing of a value \( x \in \mathbb{Z}_n \) by \( \{x\}^\mathcal{Y} := \{\{x[t]\}^\mathcal{Y}\}_{t \in \mathbb{Z}_t} \), where \( \ell := \lceil \log_2 n \rceil \) and \( x[t] \) stands for the \( t\)-th bit of \( x \). In addition, to enable free-XOR [44] and point-and-permute [45], we make the one-label for a bit defined as an offset from its corresponding zero-label as \( k_{t,1} := k_{t,0} + \Delta \) for \( x[t] \), where the least significant bit (LSB) of the offset \( \Delta \) is set to 1. Therefore, \([x[t]]^\mathcal{Y}\) can be opened by \( k_{t,0}[0] \oplus k_{t,x[t]}[0] \). We name \( k_{t,0}[0] \) as the permute bit of \( x[t] \). Note that, \( \Delta \) is chosen by the garbler and is fixed across a circuit.

**Share Conversion.** Here we describe how to convert the shared values among additive, replicated and Yao’s secret sharing. We use \( \{\mathbf{A}, \mathbf{R}, \mathbf{Y}\} \) to distinguish them respectively.

- **A2R:** To convert \([x]\) into \([x]_{\text{rep}}\) over the ring \( \mathbb{Z}_n \), \( P_1, P_2 \) exchange \( x^{(1)}, x^{(2)} \) while \( P_3, P_4 \) exchange \( x^{(3)}, x^{(4)} \). Then \( P_1, P_2 \) set \( x^{(1)} := x^{(2)} := x^{(1)} + x^{(2)} \pmod{n} \), and \( P_3, P_4 \) set \( x^{(3)} := x^{(4)} := x^{(3)} + x^{(4)} \pmod{n} \).

- **A2Y:** Given \([x]\) over the ring \( \mathbb{Z}_n \), the goal is to generate its equivalent Yao’s secret sharing \([x]_{\mathcal{Y}}^{\mathcal{Y}} := \{\{x[t]\}_{t \in \mathbb{Z}_t}\}^\mathcal{Y} \), where \( \ell := \lceil \log_2 n \rceil \). Suppose \( P_1 \) plays the role of garbler and \( P_3 \) plays the role of evaluator in the scheme. To reduce communication, we assume \( P_1 \) and \( P_2 \) agree on a random seed \( \eta_1 \in \{0, 1\}^\lambda \); \( P_1 \) and \( P_2 \) agree on a random seed \( \eta_2 \in \{0, 1\}^\lambda \). In the first round, if the offset \( \Delta \) is not defined, \( P_1 \) picks a random \( \Delta \) and sends it to \( P_2 \) and \( P_4 \). Next, for \( q \in \{1, 3\} \), \( P_i \) generates the labels \( \{k_{t,0}^{(q)}, k_{t,1}^{(q)}\}_{t \in \mathbb{Z}_t} \) for the additive share \( x^{(q)} \); for \( q \in \{2, 4\} \), \( P_1 \) and \( P_q \) generate the labels \( \{k_{t,0}^{(q)}, k_{t,1}^{(q)}\}_{t \in \mathbb{Z}_t} \) for \( x^{(q)} \) together, where zero-labels are generated by PRF with \( \eta_{q/2} \). After that, \( P_1 \) garbles an adder circuit for \( x = \sum_{i=q}^4 x^{(q)} \) with all labels of \( \{x^{(q)}\}_{q \in \{4\}} \), and obtains \( \{k_{t,0}\}_{t \in \mathbb{Z}_t} \) of...
Table 2: Notations used in our distributed ORAM scheme.

<table>
<thead>
<tr>
<th>x</th>
<th>database, which is a linear array</th>
</tr>
</thead>
<tbody>
<tr>
<td>x_k</td>
<td>k-th record in database x</td>
</tr>
<tr>
<td>n</td>
<td>number of records in database x</td>
</tr>
<tr>
<td>ℓ</td>
<td>bit-length of each record in database x</td>
</tr>
<tr>
<td>K</td>
<td>key of DPF scheme</td>
</tr>
<tr>
<td>β</td>
<td>evaluation result of DPF keys</td>
</tr>
<tr>
<td>̃β</td>
<td>shifted evaluation result of DPF keys</td>
</tr>
<tr>
<td>λ</td>
<td>security parameter</td>
</tr>
<tr>
<td>i</td>
<td>index of the record to be read/written</td>
</tr>
<tr>
<td>y</td>
<td>value to be written in write-0</td>
</tr>
<tr>
<td>δ</td>
<td>number of cyclic-shifting positions of the DPF full-domain evaluation results</td>
</tr>
<tr>
<td>Δv</td>
<td>the difference between y - xᵢ and the random value v</td>
</tr>
<tr>
<td>w, ζ</td>
<td>random masks for re-randomizing shares</td>
</tr>
</tbody>
</table>

the value x. In the second round, P₁ sends the garbled circuit to the evaluator P₃, while each non-evaluator P₄ sends its real-labels \( \{ k_{t,x(t)} \} \) to P₃. In parallel to the two communication rounds above, P₃ runs OT functionality with P₁ to receive the real-labels \( \{ k_{t,x(3)} \} \) of its additive share. Finally, P₃ locally evaluates the garbled circuit to obtain \( \{ k_{t,x(t)} \} \) of the value x.

- **Y2R**: Given the Yao’s secret sharing \( \{ x \}^Y \) of a value \( x \in \mathbb{Z}_n \), the goal is to generate its equivalent \( \{ x \}^{rep} \) over the ring \( \mathbb{Z}_n \). Suppose P₁ is the garbler and P₃ is the evaluator in the scheme. Let \( ℓ := \lceil \log_2 n \rceil \). First, P₁ picks a random \( x^{(1)} \in \mathbb{Z}_n \) and generates a garbled circuit for \( x^{(3)} := x - x^{(1)} \). After that, P₁ sends the garbled circuit, the real-labels \( \{ k_{t,x(t)}^{(1)} \} \) of \( x^{(1)} \), and the permute bits \( \{ k_{t,0,0}^{(3)} \} \) of \( x^{(3)} \) to P₃. Subsequently, P₃ evaluates the garbled circuit to obtain \( \{ k_{t,x(t)}^{(3)} \} \) and calculates \( x^{(3)}[t] := k_{t,0,0}^{(3)} \oplus k_{t,x(t)}^{(3)}[0] \) for \( t \in \mathbb{Z}_ℓ \), and then concatenates \( \{ x^{(3)}[t] \} \) to get the value \( x^{(3)} \in \mathbb{Z}_n \). Finally, P₁ sends \( x^{(1)} \) to P₂ and P₃ sends \( x^{(3)} \) to P₄.

### 2.4 Function Secret Sharing

**Function Secret Sharing** (FSS) is introduced by Boyle et al. [37]. We focus on the two-party schemes of FSS. Given a function family \( \mathcal{F} = \{ f(x) : \mathbb{G}^n \rightarrow \mathbb{G}^\text{out} \} \), a dealer uses the FSS scheme for \( \mathcal{F} \) to split a function \( f(x) \in \mathcal{F} \) into two additive shares \( \{ f(x) \} := \{ f_1(x), f_2(x) \} \), such that \( \forall x \in \mathbb{G}^n, f_1(x) + f_2(x) = f(x) \) (mod \( \mathbb{G}^\text{out} \)).

**Distributed Point Function** (DPF) is an FSS scheme for the point function \( f_{\alpha,\beta}(x) : \mathbb{G}^n \rightarrow \mathbb{G}^\text{out} \) whose range only has one non-zero value \( f_{\alpha,\beta}(\alpha) = \beta \). It consists of algorithms \( \text{Gen} \) and \( \text{Eval} \) defined as follows:

- **Gen**\( (1^λ, f_{\alpha,\beta}) \) is a key generation algorithm that outputs a pair of keys \((\mathcal{K}^{(1)}, \mathcal{K}^{(2)})\). Each key includes a random PRF seed s and \( \lceil \log_2 |\mathbb{G}^n| \rceil + 1 \) correction words. Each key is able to efficiently describe the share of \( f_{\alpha,\beta} \) without revealing \( \alpha, \beta \).

- **Eval**\((b, \mathcal{K}^{(b)}, x)\) is an evaluation algorithm. For \( \forall x \in \mathbb{G}^n, \forall b \in [2], \) it outputs \( \beta_x^{(b)} \in \mathbb{G}^\text{out} \), such that \( \beta_2^{(1)} + \beta_2^{(2)} = f_{\alpha,\beta}(x) \) (mod \( \mathbb{G}^\text{out} \)).

When DPF is used to realize a PIR protocol, the servers need to run \( \text{Eval} \) on every element of the input domain, named full domain evaluation. [46] provides a more efficient scheme for this case, rather than executing \( |\mathbb{G}^n| \) independent invocations of \( \text{Eval} \). We denote it by \( \text{EvalAll}(b, \mathcal{K}^{(b)}) \).

**Distributed Comparison Function** (DCF) is an FSS scheme for the comparison function \( f_{\alpha,\beta}(x) : \mathbb{G}^n \rightarrow \mathbb{G}^\text{out} \), which outputs \( \beta \) if \( 0 \leq x < \alpha \) and outputs 0 if \( x \geq \alpha \). Based on the DCF scheme, [47] provides the **Distributed Interval Containment Function** (DICF) construction to compute interval containment for a secret input and a publicly known interval. Denote the interval containment function as \( f_{p,q}^{\mathcal{I}}(x) : \mathbb{G}^n \rightarrow \mathbb{G}^\text{out} \), which outputs 1 if \( x \in [p,q] \) and outputs 0 otherwise. DICF is an FSS scheme for the offset interval containment function \( f_{p,q,r_{\text{in}},r_{\text{out}}}^{\mathcal{I}}(x) : \mathbb{G}^n \rightarrow \mathbb{G}^\text{out} \) with given random offset \( r_{\text{in}}, r_{\text{out}} \), such that \( f_{p,q,r_{\text{in}},r_{\text{out}}}(x + r_{\text{in}}) - r_{\text{out}} = f_{p,q}^{\mathcal{I}}(x) \). Similar to DPF, DICF also consists of algorithms.
Functionality $F_{\text{ram}}^{n,\ell}$

It interacts with $\mathcal{P} := \{P_1, \ldots, P_4\}$ and the adversary $S$.
It is parameterized with a set $\mathcal{J}$.
Initially, set $\mathcal{J} := \emptyset$.

- Upon receiving $(\text{INIT}, \text{sid}, x^{(i)})$ from $P_i \in \mathcal{P}$:
  - Send notification $(\text{INIT}, \text{sid}, P_i)$ to $S$;
  - Set $\mathcal{J} := \mathcal{J} \cup \{P_i\}$;
  - If $|\mathcal{J}| = 4$:
    * Assert $x^{(1)} = x^{(2)}$ and $x^{(3)} = x^{(4)}$;
    * Record $x := x^{(1)} + x^{(3)}$.

- Upon receiving $(\text{OREAD}, \text{sid}, ssid, j^{(i)})$ from $P_i \in \mathcal{P}$, ignore the request if $|\mathcal{J}| \neq 4$, otherwise:
  - Send notification $(\text{OREAD}, \text{sid}, ssid, P_i)$ to $S$;
  - Once all players have submitted their input, does:
    * Compute $i := \sum_{j=1}^{4} j^{(j)}$ (mod $n$);
    * Upon receiving $(\text{Rand}, \text{sid}, ssid, o^{*})$ from $S$ for the corrupted party $P_k$:
      - Pick random $o^{(1)}, \ldots, o^{(4)} \leftarrow \mathbb{Z}_2^{\ell}$ under the constraint $\sum_{j=1}^{4} o^{(j)} = x_i$ (mod $2^\ell$) and $o^{(k)} = o^{*}$;
    * For $j \in [4]$, send $(\text{OREAD}, \text{sid}, o^{(j)})$ to $P_j \in \mathcal{P}$ via private delayed channel.

- Upon receiving $(\text{OWRITE}, \text{sid}, ssid, i^{(i)}, j, y^{(j)})$ from $P_j \in \mathcal{P}$, ignore the request if $|\mathcal{J}| \neq 4$, otherwise:
  - Send notification $(\text{OWRITE}, \text{sid}, ssid, P_j)$ to $S$;
  - Once all players have submitted their input, does:
    * Compute $i := \sum_{j=1}^{4} j^{(j)}$ (mod $n$);
    * Set $x_i := \sum_{j=1}^{4} j^{(j)}$ (mod $2^\ell$);

Figure 1: The distributed ORAM functionality $F_{\text{ram}}^{n,\ell}$.

(\text{Gen}^{IC}, \text{Eval}^{IC}) as follows:
- $\text{Gen}^{IC}_{p,q,r,m,n}(\cdot)$ generates $(\mathcal{K}^{(1)}, \mathcal{K}^{(2)})$. Each key is able to efficiently describe the share of $f_{p,q,r,m,n}$, with publicly known $p, q$ but without revealing $r_{\text{in}}, r_{\text{out}}$.
- $\text{Eval}^{IC}_{p,q}(b, \mathcal{K}^{(b)}, x + r_{\text{in}})$ outputs $\beta^{(b)}$ for $b \in [2]$, such that $\beta^{(1)} + \beta^{(2)} = r_{\text{out}} = f_{p,q}(x) \pmod{\mathbb{G}^{\text{out}}}$.

In the rest of paper, we focus on the case of $r_{\text{out}} = 0$ and thus omit $r_{\text{out}}$ in the offset interval containment function and the parameters of the DICF key generation algorithm.

Definition 1. Let $T \subset \{2\}$. We say a two-party FSS scheme $(\text{Gen}, \text{Eval})$ is $T$-secure for function family $\mathcal{F} = \{f : \mathbb{G}^{\text{in}} \rightarrow \mathbb{G}^{\text{out}}\}$, if for all non-uniform PPT adversaries $A$, it holds that

$$\text{Adv}(1^\lambda, A) = \text{Pr} \left[ \frac{(f_1, f_2, \phi) \leftarrow A(1^\lambda); b \leftarrow \{1, 2\}; (\mathcal{K}^{(1)}, \mathcal{K}^{(2)}) \leftarrow \text{Gen}(1^\lambda, f_b); b^* \leftarrow A((\mathcal{K}^{(b)})_{b \in T}, \phi) : f_1, f_2 \in \mathcal{F} \land b = b^*}{\frac{1}{2} - 2} \right]$$

is negligible in $\lambda$.

3 Distributed ORAM Scheme

In this section, we propose a new distributed ORAM scheme in the pre-processing (a.k.a. online/offline) mode. Our distributed ORAM scheme is a 1-private 4-party computation protocol in which the notations used are summarized in Table 2. The functionality $F_{\text{ram}}^{n,\ell}$ of our distributed ORAM is described in Fig. 1. As shown in Fig. 2, our initialization protocol $\Pi_{\text{init}}^{n,\ell}$ only requires $4n\ell$-bit communication. For both oblivious reading and writing, the overall communication is $O(\log n)$ per access; besides, each access only takes 1 round in the online phase. Our scheme allows unlimited number of reading and writing operations without costly refreshing as in Floram [34].

Intuition. Conventionally, in a DPF-based two-server PIR protocol [37], the client holds an index $i \in \mathbb{Z}_n$ and both servers hold the same data $x \in (\mathbb{Z}_{2^n})^n$. During the PIR protocol, the client generates a pair of DPF keys for the point function $f_{i,1}(x)$ and then distributes them to the two servers. The servers then jointly evaluate and return shares of $x_i := \sum_{j=0}^{n-1} f_{i,1}(j) \cdot x_j$ to the client. While in
distributed ORAM, both the index and the data need to be protected from the MPC players using either encryption or secret sharing.

To address the former issue for the oblivious reading, we let a third player (an non-evaluator of this DPF), say $P_3$, generate a pair of DPF keys $(\mathcal{K}_1^{(1)}, \mathcal{K}_2^{(2)})$ of the point function $f_{\varphi,1}(x)$ for a random $\varphi \in \mathbb{Z}_n$ in the offline phase. $P_3$ then sends $\mathcal{K}_1$ and $\mathcal{K}_2$ to $P_1$ and $P_2$, respectively. In online phase, the index $i$ is additively secret shared among MPC players. The MPC Players jointly compute $\delta := i - \varphi \pmod n$ in the additively shared form, and send their shares of $\delta$ to the evaluators, i.e., $P_1$ and $P_2$. The evaluators then reconstruct $\delta$, perform full domain evaluation of DPF $f_\varphi$ and cyclic-shifts the output array to the right $\delta$ positions (i.e., construct the sharing of $f_{(\varphi+\delta),1}(x)$), then use the shifted array to obtain shared $x_i$. To address the latter issue, we introduce a fourth player $P_4$ and construct the replicated secret sharing of the data $x$ among four players (cf. Fig. 2). Note that the client of the initialization could be regarded as any one of the MPC players. After that, $P_1$ and $P_2$ hold the same shares of $(x)^{(r_3)}$, so they can perform DPF evaluation aided by $P_3$ on their same shares instead of the plaintext; similarly for $P_3$ and $P_4$ aided by $P_1$. Finally, $P_1$ and $P_2$ obtain the secret sharing of $[x_i]_4$ while $P_3$ and $P_4$ obtain the secret sharing of $[x_i]_4$, such that $x_i^{(1)} + x_i^{(3)} = x_i$.

Our scheme can directly perform oblivious writing on the reading memory, i.e., the replicated shared data. At a high level, if the MPC players want to obliviously write $y$ into the $i$-th position, they need to first fetch the existing $x_i$ using the aforementioned oblivious reading protocol. (Note that this step can be omitted if $x_i$ is already known in the context, cf. Sec. 4, below.) More specifically, in the offline phase, we let $P_1$ and $P_3$ jointly generate a pair of DPF keys $(\mathcal{K}_1^{(1)}, \mathcal{K}_2^{(2)})$ for the point function $f_{r_1,1}(x)$ via MPC, where $r_1 \in \mathbb{Z}_n$ is secret; meanwhile, we let $P_2$ and $P_4$ jointly generate another pair of DPF keys $(\mathcal{K}_1^{(3)}, \mathcal{K}_2^{(4)})$ for the point function $f_{r_2,v}(x)$ via MPC, where $r_2 \in \mathbb{Z}_n$ and $v \in \mathbb{Z}_2$ are secret. After that, $P_1$ and $P_3$ exchange $\mathcal{K}_1^{(1)}$ and $\mathcal{K}_1^{(2)}$; $P_2$ and $P_4$ exchange $\mathcal{K}_2^{(3)}$ and $\mathcal{K}_2^{(4)}$. In the online phase, $\delta_{w,1} := i - r_1 \pmod n$, $\delta_{w,2} := i - r_2 \pmod n$ and $\Delta v := v - x_i - v \pmod 2^4$ are opened. For $j \in \mathbb{Z}_n$, the MPC players then can jointly update $x_j := x_j + \Delta v \cdot f_{(r_1+\delta_{w,1}),1}(j) + f_{(r_2+\delta_{w,2}),v}(j)$.

### 3.1 Oblivious Reading

The oblivious reading protocol of our distributed ORAM scheme is denoted as the o-read protocol. As mentioned before, it is realized by a DPF-based PIR protocol with replicated shared database $x$ and additively shared selection index $i$, and only requires logarithmic communication in both offline and online phase. After o-read, $x_i$ is additively shared among the 4 MPC players.

**Protocol description.** Our o-read protocol is designed in the online/offline model. To reduce the protocol communication, we assume that $P_1$ and $P_3$ agree on a random seed $\eta_1 \in \{0,1\}^\lambda$; $P_1$ and $P_2$ agree on a random seed $\eta_2 \in \{0,1\}^\lambda$; $P_2$ and $P_4$ agree on a random seed $\eta_3 \in \{0,1\}^\lambda$; $P_3$ and $P_4$ agree on a random seed $\eta_4 \in \{0,1\}^\lambda$.

In the offline phase, $P_3$ invokes DPF.Gen to generate the DPF keys $(\mathcal{K}_1^{(1)}, \mathcal{K}_2^{(2)})$ for the point function $f_{\varphi_1,1}(x) : \mathbb{Z}_n \rightarrow \mathbb{Z}_2^\phi$, where $\varphi_1 \in \mathbb{Z}_n$ is randomly picked; $P_1$ invokes DPF.Gen to generate DPF keys $(\mathcal{K}_1^{(3)}, \mathcal{K}_2^{(4)})$ for the point function $f_{\varphi_2,1}(x) : \mathbb{Z}_n \rightarrow \mathbb{Z}_2^\phi$, where $\varphi_2 \in \mathbb{Z}_n$ is randomly picked. Subsequently, $P_3$ sends $\mathcal{K}_1^{(1)}$ to $P_1$ and $\mathcal{K}_2^{(2)}$ to $P_2$; $P_1$ sends $\mathcal{K}_1^{(3)}$ to $P_3$ and $\mathcal{K}_2^{(4)}$ to $P_2$.

In the online phase, the MPC players compute $(\delta_{r,1}) := (i) - [\varphi_1]$ and $(\delta_{r,2}) := (i) - [\varphi_2]$, and then open $\delta_{r,1}$ to $P_1, P_2$ while opening $\delta_{r,2}$ to $P_3, P_4$. After that, for $j \in \{2\}$, the player $P_j$ first invokes DPF.Eval to perform full domain evaluation of the DPF key received in the offline phase. Next, $P_j$ cyclic-shifts the evaluation result $\{\beta^{(j)}_{\varphi_1,k}\}_{k \in \mathbb{Z}_n}$ to the right $\delta_{r,1}$ positions, denoted as $\{\beta^{(j)}_{\varphi_1,k}\}_{k \in \mathbb{Z}_n}$. It is easy to see,
Protocol $\Pi_{\text{read}}^{\alpha,\ell}$

Initialization:
- $P_1$ and $P_3$ agree on a random seed $\eta_1 \leftarrow \{0,1\}^\lambda$;
- $P_1$ and $P_2$ agree on a random seed $\eta_2 \leftarrow \{0,1\}^\lambda$;
- $P_2$ and $P_4$ agree on a random seed $\eta_3 \leftarrow \{0,1\}^\lambda$;
- $P_3$ and $P_4$ agree on a random seed $\eta_4 \leftarrow \{0,1\}^\lambda$.

Offline phase:
- Upon initialization, $P_1$ does:
  - Generate $\varphi_2 \leftarrow Z_n$, set $\varphi_2^{(1)} := \varphi_2 - \text{PRF}_{\eta_2}^{n}(\text{sid}, 0)$;
  - Set $(K_{\varphi_2}^{(1)}, K_{\varphi_2}^{(2)}) \leftarrow \text{DPF.Gen}(1^\lambda, \varphi_2, 1)$;
  - Send $(\text{sid}, K_{\varphi_2}^{(1)})$ to $P_3$, $(\text{sid}, K_{\varphi_2}^{(2)})$ to $P_4$;
- Upon initialization, $P_3$ does:
  - Generate $\varphi_1 \leftarrow Z_n$, set $\varphi_1^{(1)} := \varphi_1 - \text{PRF}_{\eta_4}^{n}(\text{sid}, 0)$;
  - Set $(K_{\varphi_1}^{(1)}, K_{\varphi_1}^{(2)}) \leftarrow \text{DPF.Gen}(1^\lambda, \varphi_1, 1)$;
  - Send $(\text{sid}, K_{\varphi_1}^{(1)})$ to $P_1$, $(\text{sid}, K_{\varphi_1}^{(2)})$ to $P_2$;
- Upon initialization, $P_2$ sets $\varphi_2^{(2)} := \text{PRF}_{\eta_2}^{n}(\text{sid}, 0)$;
- Upon initialization, $P_4$ sets $\varphi_1^{(2)} := \text{PRF}_{\eta_4}^{n}(\text{sid}, 0)$.

Online phase:
- With the private input $(\text{OREAD}, \text{sid}, \text{ssid}, i^{(j)})$, the player $P_j, j \in \{1, 2\}$ does:
  - Set $w_{1,j} \leftarrow \text{PRF}_{\eta_2}^{n}(\text{sid}, 1)$, $w_{2,j} \leftarrow \text{PRF}_{\eta_2}^{n}(\text{sid}, 2)$;
  - Set $i^{(j)} := i^{(j)} + w_{1,j}$ (mod $n$);
  - Set $\delta^{(j)} := i^{(j)} - \varphi_2^{(j)} + w_{2,j}$ (mod $n$);
  - Send $(\text{sid}, \text{ssid}, i^{(j)})$ to $P_{3-j}$, $(\text{sid}, \text{ssid}, i^{(j)})$ to $P_3$ and $P_5$;
- With the private input $(\text{OREAD}, \text{ssid}, i^{(j)})$, the player $P_j, j \in \{3, 4\}$ does:
  - Set $w_{1,j-2} \leftarrow \text{PRF}_{\eta_2}^{n}(\text{sid}, 1)$;
  - Set $w_{2,j-2} \leftarrow \text{PRF}_{\eta_2}^{n}(\text{sid}, 2)$;
  - Set $\delta^{(j)} := i^{(j)} - \varphi_1^{(j-2)} - w_{1,j-2}$ (mod $n$);
  - Set $\delta^{(j)} := i^{(j)} - w_{2,j-2}$ (mod $n$);
  - Send $(\text{ssid}, \text{ssid}, \delta^{(j)})$ to $P_1$ and $P_2$, $(\text{sid}, \text{ssid}, \delta^{(j)})$ to $P_{7-j}$;
- Upon receiving $(\text{sid}, \text{ssid}, \delta^{(3-j)})$ from $P_{3-j}$, $(\text{ssid}, \text{ssid}, \delta^{(3-j)})$ from $P_3$, and $(\text{ssid}, \text{ssid}, \delta^{(4)}(j))$ from $P_4$, $P_{j}, j \in \{1, 2\}$ does:
  - Set $\delta^{(j)} := \delta^{(j)} + \delta^{(2)} + \delta^{(3)} + \delta^{(4)}$ (mod $n$);
  - Set $(\beta^j_{\varphi_1,k})_{k \in Z_n} \leftarrow \text{DPF.EvalAll}(j, K_{\varphi_1}^{(j)})$;
  - Set $\bar{\beta}^j_{\varphi_1,k} := \beta^j_{\varphi_1,k} \mod 2^\delta$ for $k \in Z_n$;
  - Set $\zeta_j \leftarrow \text{PRF}_{\eta_2}^{n}(\text{sid}, 1)$, $\zeta_j \leftarrow \text{PRF}_{\eta_2}^{n}(\text{sid})$
  - Return $o^{(j)} := \sum_{k=0}^{n-1} (x^{(j)}_k, \bar{\beta}^j_{\varphi_2,k} + \zeta_j + (-1)^j \cdot \zeta_4)$.
- Upon receiving $(\text{sid}, \text{ssid}, \delta^{(1)}(j))$ from $P_1$, $(\text{ssid}, \text{ssid}, \delta^{(2)}(j))$ from $P_2$, and $(\text{ssid}, \text{ssid}, \delta^{(4)}(j))$ from $P_{7-j}$, $P_{j}, j \in \{3, 4\}$ does:
  - Set $\delta^{(j)} := \delta^{(1)} + \delta^{(2)} + \delta^{(3)} + \delta^{(4)}$ (mod $n$);
  - Set $(\beta^j_{\varphi_2,k})_{k \in Z_n} \leftarrow \text{DPF.EvalAll}(j-2, K_{\varphi_2}^{(j-2)})$;
  - Set $\bar{\beta}^j_{\varphi_2,k} := \beta^j_{\varphi_2,k} \mod 2^\delta$ for $k \in Z_n$;
  - Set $\zeta_{j-2} \leftarrow \text{PRF}_{\eta_2}^{n}(\text{sid})$, $\zeta_4 \leftarrow \text{PRF}_{\eta_2}^{n}(\text{sid})$
  - Return $o^{(j)} := \sum_{k=0}^{n-1} (x^{(j)}_k, \bar{\beta}^j_{\varphi_2,k} - \zeta_{j-2} + (-1)^j \cdot \zeta_4)$.

Figure 3: O-read protocol $\Pi_{\text{read}}^{\alpha,\ell}$.

the only non-zero value is $[\bar{\beta}_{\varphi_1,j}] = 1$ in the shifted array. Similarly, $P_3$ and $P_4$ use their DPF keys $(K_{\varphi_1}^{(1)}, K_{\varphi_2}^{(2)})$ and the corresponding $\delta_{r,2}$ for cyclic-shifting to obtain $(\bar{\beta}_{\varphi_2,k})_{k \in Z_n}$. Consequently, we have $x_i = \sum_{j=1}^{n-1} \delta^{(j)}$ (mod $2^\delta$) where for $j \in [4]$:

$$\delta^{(j)} := \sum_{k=0}^{n-1} (x^{(j)}_k, \bar{\beta}^j_{\varphi_2,k}) \pmod{2^\delta}$$
It interacts with $P := \{P_1, P_2\}$ and the adversary $S$. It is parameterized with the DPF key generation algorithm DPF.Gen.

- Upon receiving $(\text{KEYGen}, sid, r(j), v(j))$ from $P_j$, $j \in [2]$;
  - Send notification $(\text{KEYGen}, \text{sid}, P_j)$ to $S$;
  - Once both players have submitted their input, does:
    * Compute $r := r(1) + r(2) \pmod n$, $v := v(1) + v(2) \pmod {2^\ell}$;
    * Generate $(K(1), K(2)) \leftarrow \text{DPF.Gen}(1^n, f_{\alpha, \beta})$, for the point function $f_{\alpha, \beta}(x) : \mathbb{Z}_n \rightarrow \mathbb{Z}_{2^\ell}$;
    * For $j \in [2]$, send $(\text{KEYGen}, \text{sid}, K(j))$ to $P_j \in P$ via private delayed channel.

Figure 4: Distributed DPF key generation functionality $\mathcal{F}^{n,\ell}_{\text{dGen}}[\text{DPF.Gen}]$.}

when $q := 2 + (-1)^j$. Finally, we re-randomize $d(j)$ to ensure the uniform distribution.

### 3.2 Oblivious Writing

The oblivious writing protocol of our distributed ORAM scheme is denoted as $o$-write protocol. The $o$-write protocol allows the 4 MPC players to jointly update $y$ to the $i$-th position of database $x$ without revealing $i$ and $y$. Analogously, our $o$-write protocol is designed in the online/offline mode, and it can be directly applied to the replicated shared database.

For the sake of readability, we use $\mathcal{F}^{n,\ell}_{\text{dGen}}[\text{DPF.Gen}]$ in $o$-write protocol description for distributed DPF key generation, and its actual 2PC protocol $\Pi^{n,\ell}_{\text{dGen}}$ can be found in [34].

**Protocol description.** As depicted in Fig. 5, our $o$-write protocol assumes that $P_1$ and $P_3$ agree on a random seed $\eta_1 \in \{0,1\}^\lambda$; $P_1$ and $P_2$ agree on a random seed $\eta_2 \in \{0,1\}^\lambda$; $P_2$ and $P_4$ agree on a random seed $\eta_4 \in \{0,1\}^\lambda$.

In the offline phase, $P_1$ and $P_3$ invoke $\mathcal{F}^{n,\ell}_{\text{dGen}}[\text{DPF.Gen}]$ to jointly generate a pair of DPF keys $(\mathcal{K}^{(1)}_1, \mathcal{K}^{(2)}_1)$ to describe a point function $f_{r_1,1}(x)$, where random $r_1 \leftarrow \mathbb{Z}_n$ is additively shared between $P_1$ and $P_2$. After the execution, $P_1$ obtains the share $\mathcal{K}^{(1)}_1$ of $f_{r_1,1}(x)$ and sends it to $P_2$; $P_3$ obtains the share $\mathcal{K}^{(2)}_1$ and sends it to $P_4$. Meanwhile, $P_2$ and $P_4$ also jointly generate a pair of DPF keys $(\mathcal{K}^{(1)}_2, \mathcal{K}^{(2)}_2)$ to express a point function $f_{r_2,v}(x)$, where random $r_2 \leftarrow \mathbb{Z}_n$ and $v \leftarrow \mathbb{Z}_{2^\ell}$ are additively shared between $P_2$ and $P_4$. Then $P_2$ obtains the share $\mathcal{K}^{(1)}_2$ of $f_{r_2,v}(x)$ and sends it to $P_1$; $P_4$ obtains the share $\mathcal{K}^{(2)}_2$ and sends it to $P_3$.

In the online phase, four MPC players jointly compute and open $(\delta_{w,1}) := (i) - \lfloor r_1 \rfloor \pmod n$, $(\delta_{w,2}) := (i) - \lfloor r_2 \rfloor \pmod n$, and $(\Delta v) := (y) - (x_i) - \lfloor v \rfloor \pmod {2^\ell}$. Each party then locally performs full domain evaluation of obtained DPF keys, and cyclic-shifts the output arrays of shared $f_{r_1,1}$, $f_{r_2,v}$ to the right $\delta_{w,1}$, $\delta_{w,2}$ positions respectively, followed by adding the (scaled) shifted array to the shares of data $x$. More specifically, $P_j$ full domain evaluates $\mathcal{K}^{(1,j/2)}_1$, $\mathcal{K}^{(j/2)}_1$ to get $\{\beta^{(j/2)}_{1,k}\}_{k \in \mathbb{Z}_n}$, $\{\beta^{(j/2)}_{v,k}\}_{k \in \mathbb{Z}_n}$. After that, $P_j$ cyclic-shifts $\{\beta^{(j/2)}_{1,k}\}_{k \in \mathbb{Z}_n}$ to the right $\delta_{w,1}$ positions, $\{\beta^{(j/2)}_{v,k}\}_{k \in \mathbb{Z}_n}$ to the right $\delta_{w,2}$ positions, and denotes the shifted arrays as $\{\tilde{\beta}^{(j/2)}_{1,k}\}_{k \in \mathbb{Z}_n}$ and $\{\tilde{\beta}^{(j/2)}_{v,k}\}_{k \in \mathbb{Z}_n}$ respectively. $P_j$ then updates the shares of database as $x^{(j)}_k := x^{(j)}_k + \Delta v \cdot \tilde{\beta}^{(j/2)}_{1,k} + \tilde{\beta}^{(j/2)}_{v,k} \pmod {2^\ell}$, $k \in \mathbb{Z}_n$.

Note that $P_1$ and $P_2$ still have identical shares of $x$ after the updating; similarly for $P_3$ and $P_4$. That is, four parties obliviously write new value $y$ at the $i$-th position and maintain the database in the replicated shared form.

### 3.3 Security

We show the security of $\Pi^{n,\ell}_{\text{init}}$, $\Pi^{n,\ell}_{\text{read}}$ and $\Pi^{n,\ell}_{\text{write}}$ with the following theorem, and its proof can be found in Appendix B.

**Theorem 1.** Let $\text{DPF}^{\mathbb{Z}_n,\mathbb{Z}_{2^\ell}}$ be a secure function secret sharing scheme for point function $f_{\alpha,\beta}(x) : \mathbb{Z}_n \rightarrow \mathbb{Z}_{2^\ell}$ with adversarial advantage $\text{Adv}^{\mathbb{Z}_n,\mathbb{Z}_{2^\ell}}_{\text{DPF}}(1^n, \mathcal{A})$. Let $\text{PRF}^{\mathbb{Z}_n} : \{0,1\}^n \times \{0,1\}^n \rightarrow \mathbb{Z}_n$ be
Initialization:
- \( P_1 \) and \( P_3 \) agree on a random seed \( \eta_1 \leftarrow \{0,1\}^\lambda \);
- \( P_1 \) and \( P_2 \) agree on a random seed \( \eta_2 \leftarrow \{0,1\}^\lambda \);
- \( P_2 \) and \( P_4 \) agree on a random seed \( \eta_3 \leftarrow \{0,1\}^\lambda \);
- \( P_3 \) and \( P_4 \) agree on a random seed \( \eta_4 \leftarrow \{0,1\}^\lambda \).

Offline phase:
- \( P_3 \) sets \( r_1^{(1)} \leftarrow Z_n \), \( P_1 \) sets \( r_2^{(1)} \leftarrow Z_n \);
- \( P_3 \) and \( P_5 \) request \( F_{\text{CR}}^{\eta_0,\ell}[\text{DPF}.\text{Gen}] \):
  - \( P_1 \) sends (\( \text{KEYGEN}, \text{sid}, r_1^{(1)}, 0 \)) and gets \( \mathcal{K}_1^{(1)} \);
  - \( P_3 \) sends (\( \text{KEYGEN}, \text{sid}, r_1^{(2)}, 1 \)) and gets \( \mathcal{K}_1^{(2)} \);
- \( P_1 \) sends (\( \text{sid}, \mathcal{K}_1^{(1)} \)) to \( P_2 \), \( P_3 \) sends (\( \text{sid}, \mathcal{K}_1^{(2)} \)) to \( P_4 \);
- \( P_2 \) sets \( r_2^{(1)} \leftarrow Z_n \), \( v^{(1)} \leftarrow Z_{2r} \);
- \( P_4 \) sets \( r_2^{(2)} \leftarrow Z_n \), \( v^{(2)} \leftarrow Z_{2r} \);
- \( P_2 \) and \( P_4 \) request \( F_{\text{CR}}^{\eta_0,\ell}[\text{DPF}.\text{Gen}] \):
  - \( P_2 \) sends (\( \text{KEYGEN}, \text{sid}, r_2^{(1)}, v^{(1)} \)) and gets \( \mathcal{K}_1^{(1)} \);
  - \( P_4 \) sends (\( \text{KEYGEN}, \text{sid}, r_2^{(2)}, v^{(2)} \)) and gets \( \mathcal{K}_1^{(2)} \);
- \( P_2 \) sends (\( \text{sid}, \mathcal{K}_1^{(1)} \)) to \( P_1 \), \( P_4 \) sends (\( \text{sid}, \mathcal{K}_1^{(2)} \)) to \( P_3 \);

Online phase:
- With the private input (\( \text{OWRITE}, \text{sid}, \text{ssid}, i^{(j)}, x_i^{(j)}, y^{(j)} \)), the player \( P_j, j \in \{1,3\} \) sets:
  - \( w_{k,q}^{(j)} \leftarrow \text{PRF}_{\text{sym}}^{\eta_0}(\text{sid}, k), k \in [2], q \in \{1, j + 1\} \);
  - \( w_{3,q}^{(j)} \leftarrow \text{PRF}_{\text{sym}}^{\eta_0}(\text{sid}, 3), q \in \{1, j + 1\} \);
  - \( \delta_{w,1}^{(j)} := i^{(j)} - r_1^{(j/2)} + (j - 2) \cdot w_{1,1}^{(j)} + w_{1,j+1}^{(j)} \);
  - \( \delta_{w,2}^{(j)} := i^{(j)} + (j - 2) \cdot w_{2,1}^{(j)} + w_{2,j+1}^{(j)} \);
  - \( \Delta v^{(j)} := y_i^{(j)} - x_i^{(j)} + (j - 2) \cdot w_{3,1}^{(j)} + w_{3,j+1}^{(j)} \);
- With the private input (\( \text{OWRITE}, \text{sid}, \text{ssid}, i^{(j)}, x_i^{(j)}, y^{(j)} \)), the player \( P_j, j \in \{2,4\} \) sets:
  - \( w_{k,q}^{(j)} \leftarrow \text{PRF}_{\text{sym}}^{\eta_0}(\text{sid}, k), q \in \{3, j\}, k \in [2] \);
  - \( w_{3,q}^{(j)} \leftarrow \text{PRF}_{\text{sym}}^{\eta_0}(\text{sid}, 3), q \in \{3, j\} \);
  - \( \delta_{w,1}^{(j)} := i^{(j)} + (j - 3) \cdot w_{1,3}^{(j)} - w_{1,j}^{(j)} \);
  - \( \delta_{w,2}^{(j)} := i^{(j)} - r_2^{(j/2)} + (j - 3) \cdot w_{2,3}^{(j)} - w_{2,j}^{(j)} \);
  - \( \Delta v^{(j)} := y_i^{(j)} - x_i^{(j)} - v^{(j/2)} + (j - 3) \cdot w_{3,3}^{(j)} - w_{3,j}^{(j)} \);
- \( P_j, j \in [4] \) sends (\( \text{sid}, \text{ssid}, \delta_{w,1}^{(j)}, \delta_{w,2}^{(j)}, \Delta v^{(j)} \)) to others.
- Upon receiving all (\( \text{sid}, \text{ssid}, \delta_{w,1,k}^{(j)}, \delta_{w,2,k}^{(j)}, \Delta v^{(k)} \)) from \( P_k \) for \( k \in [4]/j \), party \( P_j, j \in [4] \) sets:
  - \( \delta_{w,k} := \sum_{q=1}^{4} \delta_{w,k}^{(q)} \mod n \), for \( k \in [2] \);
  - \( \Delta v := \sum_{q=1}^{4} \Delta v^{(q)} \mod m \);
  - \( \langle \beta_{1,k}^{(j/2)}, h \rangle_{Z_n} \leftarrow \text{DPF}.\text{EvalAll}([j/2], \mathcal{K}_1^{(j/2)}) \);
  - \( \langle g_{v,k}^{(j/2)}, h \rangle_{Z_n} \leftarrow \text{DPF}.\text{EvalAll}([j/2], \mathcal{K}_v^{(j/2)}) \);
  - for \( k := 0 \) to \( n - 1 \):
    - \( \beta_{1,k}^{(j/2)} := \beta_{1,k}^{(j/2)} \mod n \);
    - \( \beta_{v,k}^{(j/2)} := \beta_{v,k}^{(j/2)} \mod n \);
    - \( x_k^{(j)} := x_k^{(j)} + \Delta v \cdot \beta_{1,k}^{(j/2)} + \beta_{v,k}^{(j/2)} \).

Figure 5: O-write protocol \( \Pi_{\text{write}}^{\lambda,\ell} \).
Table 3: Our instruction set.

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Operator</th>
<th>Operands</th>
<th>Effects for registers</th>
<th>Flag</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000</td>
<td>and</td>
<td>i,j, A</td>
<td>compute bitwise AND of i_j and a and store result in i-th register</td>
<td>result is 0'</td>
</tr>
<tr>
<td>00001</td>
<td>or</td>
<td>i,j, A</td>
<td>compute bitwise OR of i_j and a and store result in i-th register</td>
<td>result is 0'</td>
</tr>
<tr>
<td>00010</td>
<td>xor</td>
<td>i,j, A</td>
<td>compute bitwise XOR of i_j and a and store result in i-th register</td>
<td>result is 0'</td>
</tr>
<tr>
<td>00011</td>
<td>not</td>
<td>i</td>
<td>compute bitwise NOT of a and store result in i-th register</td>
<td>result is 0'</td>
</tr>
<tr>
<td>00100</td>
<td>shl</td>
<td>i,j, A</td>
<td>shift i_j by a bits to the left and store result in i-th register</td>
<td>MSB of i_j</td>
</tr>
<tr>
<td>00101</td>
<td>shr</td>
<td>i,j, A</td>
<td>shift i_j by a bits to the right and store result in i-th register</td>
<td>LSB of i_j</td>
</tr>
<tr>
<td>00110</td>
<td>add</td>
<td>i,j, A</td>
<td>compute i_j + a and store result in i-th register</td>
<td>overflow</td>
</tr>
<tr>
<td>00111</td>
<td>sub</td>
<td>i,j, A</td>
<td>compute i_j - a and store result in i-th register</td>
<td>borrow</td>
</tr>
<tr>
<td>01000</td>
<td>mul</td>
<td>i,j, A</td>
<td>compute i_j * a and store the least significant bits of result in i-th register</td>
<td>overflow</td>
</tr>
<tr>
<td>01001</td>
<td>umul</td>
<td>i,j, A</td>
<td>compute i_j * a and store the most significant bits of result in i-th register</td>
<td>overflow</td>
</tr>
<tr>
<td>01010</td>
<td>smul</td>
<td>i,j, A</td>
<td>compute signed i_j * a and store the most significant bits of result in i-th register</td>
<td>overflow</td>
</tr>
<tr>
<td>01011</td>
<td>cmpl</td>
<td>i</td>
<td>if i = 0, store a in i-th register</td>
<td>result is 0</td>
</tr>
<tr>
<td>01100</td>
<td>cmpl</td>
<td>i</td>
<td>if i = 1, store a in i-th register</td>
<td>result is 0</td>
</tr>
<tr>
<td>01101</td>
<td>cmpe</td>
<td>i</td>
<td>if i &lt; 0, store a in i-th register</td>
<td>result is 0</td>
</tr>
<tr>
<td>01110</td>
<td>cmpg</td>
<td>i</td>
<td>if i &gt; 0, store a in i-th register</td>
<td>result is 0</td>
</tr>
<tr>
<td>01111</td>
<td>cmpps</td>
<td>i</td>
<td>if i = 0, store a in i-th register</td>
<td>result is 0</td>
</tr>
<tr>
<td>10000</td>
<td>mov</td>
<td>i</td>
<td>store a in i-th register</td>
<td>result is 0</td>
</tr>
<tr>
<td>10001</td>
<td>cmov</td>
<td>i</td>
<td>store a in i-th register</td>
<td>result is 0</td>
</tr>
<tr>
<td>10100</td>
<td>store</td>
<td>i</td>
<td>store a in memory that is aligned to the a-th word</td>
<td>result is 0</td>
</tr>
<tr>
<td>10101</td>
<td>load</td>
<td>i</td>
<td>store into i-th register the word in memory that is aligned to the a-th word</td>
<td>result is 0</td>
</tr>
<tr>
<td>10100</td>
<td>read</td>
<td>i</td>
<td>store the next word of a-th tape in i-th register</td>
<td>result is 0</td>
</tr>
<tr>
<td>10101</td>
<td>jmp</td>
<td>A</td>
<td>store a in pc</td>
<td>result is 0</td>
</tr>
<tr>
<td>10110</td>
<td>cjmp</td>
<td>A</td>
<td>store a in pc</td>
<td>result is 0</td>
</tr>
<tr>
<td>10111</td>
<td>cnjmp</td>
<td>A</td>
<td>store a in pc</td>
<td>result is 0</td>
</tr>
<tr>
<td>11000</td>
<td>ans</td>
<td>A</td>
<td>stall or halt (and return a)</td>
<td>result is 0</td>
</tr>
</tbody>
</table>

A secure pseudorandom function with adversarial advantage \( \text{Adv}_{\text{PRF}}^{2^{3\ell}} (1^{\lambda}, \mathcal{A}) \). Let \( \text{PRF}^{2^{3\ell}} : \{0,1\}^\lambda \times \{0,1\}^\ell \rightarrow \mathbb{Z}_{2^{3\ell}} \) be a secure pseudorandom function with adversarial advantage \( \text{Adv}_{\text{PRF}}^{2^{3\ell}} (1^{\lambda}, \mathcal{A}) \). The series of protocols \( \Pi_{\text{ram}}^{\ell} := \left\{ \Pi_{\text{init}}^{\ell}, \Pi_{\text{read}}^{\ell}, \Pi_{\text{write}}^{\ell} \right\} \) UC-realizes \( \mathcal{F}_{\text{ram}}^{\ell} \) as described in Fig. 1 in the \( \mathcal{F}_{\text{Gen}}^{\lambda} \)-hybrid model against semi-honest adversaries who can statically corrupted up to 1 player with distinguishing advantage

\[
11 \cdot \text{Adv}_{\text{PRF}}^{2^{3\ell}} (1^{\lambda}, \mathcal{A}) + 4 \cdot \text{Adv}_{\text{PRF}}^{2^{3\ell}} (1^{\lambda}, \mathcal{A}) + 3 \cdot \text{Adv}_{\text{GBPR}}^{2^{4\ell}} (1^{\lambda}, \mathcal{A})
\]

4 MPC for TinyRAM

Our construction follows the RISC framework with random-access memory, whose word size is denoted by \( \ell \) and required to be a power of 2 and divisible by 8. It can be seen as a simplified version of TinyRAM [38]. The state of this machine consists of the following.

- The program counter, denoted as pc \( \in \{0,1\}^\ell \).
- The condition flag, denoted as flg \( \in \{0,1\}^\ell \).
- \( m \) general purpose registers, each storing one word.
- Memory, which is a linear array \( (M_i)_{i \in \mathbb{Z}_{2^\ell}} \) of \( 2^\ell \) words.
- Two input tapes: \( \text{tape}_0 \) is used for the primary input, and \( \text{tape}_1 \) is used for an auxiliary input. Each tape has a read counter, denoted as \( r_{\text{co}}, r_{\text{ci}} \in \{0,1\}^\ell \) and input length, denoted as \( \text{num}_0, \text{num}_1 \in \{0,1\}^\ell \).

During a program execution, we assume that the input tapes are read-only and the \( \text{num}_0 \)-th (or \( \text{num}_1 \)-th) word in \( \text{tape}_0 \) (or \( \text{tape}_1 \)) is set to 0. At each step, the machine executes an instruction that changes its state. The instruction set of our RISC includes 25 instructions (a subset of the TinyRAM’s) as summarized in Table 3. Let \( i \in \mathbb{Z}_m \) denote a register index and \( \text{ri} \) denotes the \( \ell \)-bit string currently stored in the \( i \)-th register; \( A \) is either an immediate value or a register index, and \( \alpha \) denotes its value (i.e., the immediate value itself or the \( \ell \)-bit string currently stored in \( A \)-th register). To distinguish between unsigned and signed integers represented by the same \( \ell \)-bit string \( x \), we let \( x_u \) stand for the unsigned integer encoded by \( x \), and \( x_s \) stand for the signed integer. Each instruction is specified via an operator and up to three operands. Generally, the first operand is the destination register where the result shall be stored, and the other operands are the arguments to the instruction. In practice, each instruction is encoded into a \( 2\ell \)-bit string, and it consists of following 6 fields:

\[
\begin{array}{|c|c|c|c|}
\hline
\text{Opcode} & \text{Operator} & \text{Operands} & \text{Effects for registers} \\
\hline
\end{array}
\]
Figure 6: Oblivious RAM-MPC Architecture.

- Field 1 stores the instruction operator, which is identified by an opcode $\text{op}$ with $5 = \lceil \log_2 25 \rceil$ bits.
- Field 2 stores an indicator, which consists of a bit and a (dummy) register index. It is set to $1||A[\ell - \lceil \log_2 m \rceil, \ell]$ for all bits if $A$ is a register index and $0||A[\ell - \lceil \log_2 m \rceil, \ell]$ if $A$ is an immediate value.
- Field 3 stores a register index with $\lceil \log_2 m \rceil$ bits.
- Field 4 stores a register index with $\lceil \log_2 m \rceil$ bits.
- Field 5 consists of $\ell - 6 - 3\lceil \log_2 m \rceil$ bits of 0’s to pad the instruction.
- Field 6 stores an immediate value or a register index, which takes $\ell$ bits.

Therefore, our encoding scheme requires $6 + 3\lceil \log_2 m \rceil \leq \ell$. For instance, if we take $m, \ell := 32$, a valid encoding of the instruction “mull 8 15 523”, where 523 is an immediate value, is as follows:

\[
\begin{array}{cccccccc}
0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 \\
\text{mull} & A \text{ is imm.} & r := 8 & j := 15 & \text{padding} & A := 523 \\
\end{array}
\]

We construct a series of MPC protocols to simulate the RISC machine as described above, where programs can be executed obliviously in the RAM mode. That is, our system can naturally support control flow, loops, subroutines, and recursion, etc. In our simulated RISC machine, each word is represented by two’s complement. If a word $x$ is secret, the MPC players share the unsigned integer value it represents. Thus we will omit the subscript $s$ or $u$ that identify signed or unsigned of the word value. Our registers, memory and tapes are shared among four parties in the replicated secret sharing and supports random-access as described in Sec. 3, while flag, $\text{pc}$, $\{rc_0, rc_1\}$ and $\{num_0, num_1\}$ are (4, 4)-additively shared. Note that the initial content of above all are specified by the program owner. In addition, before “loading” a private program into our oblivious memory, the program owner should unify the representation of each instruction. For example, the instruction “jmp $A$” is transformed to “ jmp $i’ j’ A’”, where $i’, j’ \leftarrow Z_m$ are dummy data.

We will formally describe the details of the machine execution in Sec. 4.1, Sec. 4.2, Sec. 4.3 and Sec. 4.4, below; briefly, each step is an oblivious instruction cycle following the architecture as shown in Fig. 6: (i) parities first obliviously fetch the $\text{pc}$-th instruction; (ii) securely decode the instruction to load arguments and determine the operator type (in the shared form); (iii) perform all possible operations via MPC; (iv) obliviously select the right result based on the shared operator type and write it into the destination register and condition flag.
4.1 Instruction Fetching

Instruction fetching is the first phase of each instruction cycle. Since the private program instructions are stored in the replicated-shared form while the program counter are (4, 4)-additively shared, we utilize our o-read protocol to obliviously fetch the \( \text{pc} \)-th instruction and spend only 1 online round. More specifically, four MPC players begin by invoking \( \Pi_{\text{read}}^{n,4} \) with the secret shared program counter \( \langle \text{pc} \rangle \) over the private memory, \(^1\) and then locally increase the program counter as \( \langle \text{pc} \rangle := \langle \text{pc} \rangle + 2 \mod n \).\(^2\) Note that, if needed, \( \text{pc} \) will be corrected during the evaluation of jump operations.

After this phase, parties shares the opcode \( \langle \text{op} \rangle \), indicator \( \langle d \rangle \) and operands \( \{ \langle i \rangle, \langle j \rangle, \langle A \rangle \} \) of the current instruction.

4.2 Instruction Decoding

Our instruction decoding phase consists of (i) arguments loading from registers and (ii) operator determination. While presented in two steps, we note that they can be performed at the same time and require 1 online round.

4.2.1 Arguments Loading

We load \( \langle ri \rangle, \langle rj \rangle \text{ and } \langle a \rangle \) by our o-read protocol in parallel as follows. For \( \langle ri \rangle \) and \( \langle rj \rangle \), four MPC players simply invoke \( \Pi_{\text{read}}^{m,\ell} \) with the input \( \langle i \rangle \) and \( \langle j \rangle \) respectively over the private registers. For \( \langle a \rangle \), parties first convert \( \langle A \rangle \) to the replicated share form, and then jointly invoke \( \Pi_{\text{read}}^{(2^{\lceil \log_2 m \rceil} + m),\ell} \) with the indicator \( \langle d \rangle \) over the replicated shared array in the form of

\[
\mathbf{v} := A, \ldots, A \underbrace{||(rk)_{k \in \mathbb{Z}_m}}_{2^{\lceil \log_2 m \rceil}}.
\]

If \( A \) is an immediate value, the MSB of \( d \) is 0, so that the \( d \)-th position is in the interval \([0, 2^{\lceil \log_2 m \rceil})\) of \( \mathbf{v} \), where all items are \( A \); if \( A \) is a register index, the MSB of \( d \) is 1 followed by the target register index, so that the \( d \)-th position of \( \mathbf{v} \) corresponds to the value of the \( A \)-th register. In practice, the share conversion for \( A \) is performed in the same online communication round of \( \Pi_{\text{read}}^{(2^{\lceil \log_2 m \rceil} + m),\ell} \).

4.2.2 Operator Determination

We determine the operator type of current instruction using the DPF scheme. After this step, \( \langle \text{op} \rangle \) is converted to a unit vector of 25 elements in the shared form, where the \( \text{op} \)-th element is 1 and all the other elements are 0’s. Namely, the MPC players share the point function \( f_{\text{op},1}(x) : \mathbb{Z}_{25} \to \mathbb{Z}_{25} \). More specifically, to construct the shares of \( f_{\text{op},1}(x) \) without revealing \( \text{op} \), \( P_3 \) generates a pair of DPF keys for the point function \( f_{\psi_1,1}(x) : \mathbb{Z}_{25} \to \mathbb{Z}_{25} \), where \( \psi_1 \in \mathbb{Z}_{25} \) is randomly picked, and then distributes them to \( P_1, P_2; P_1 \) also generates a pair of DPF keys for the point function \( f_{\psi_2,1}(x) : \mathbb{Z}_{25} \to \mathbb{Z}_{25} \), where \( \psi_2 \in \mathbb{Z}_{25} \) is randomly picked, and then distributes them to \( P_0, P_4 \). Meanwhile, four MPC players open \( \langle \delta_1 \rangle := \langle \text{op} \rangle + \psi_1 \mod 2^{5} \) to \( P_1, P_2 \), \( \langle \delta_2 \rangle := \langle \text{op} \rangle + \psi_2 \mod 2^{5} \) to \( P_3, P_4 \). After that, \( P_1 \) and \( P_2 \) share the point function \( f_{\delta_1-\psi_1,1}(x) \) by evaluating \( \llbracket f_{\psi_1,1}(x) \rrbracket \) with the input \( \delta_1 - x \) on the \( x \)-th position; similarly, \( P_3 \) and \( P_4 \) share the point function \( f_{\delta_2-\psi_2,1}(x) \). Note that \( \delta_1 - \psi_1 = \delta_2 - \psi_2 = \text{op} \).

4.3 Operation Evaluation

Let \( \text{op} \in \mathbb{Z}_{25} \) denote our 25 supported operations while \( \text{op} \) stands for the right opcode of current instruction. We construct a series of 4PC protocols to obliviously evaluate 25 operations. Except for answer, each \( \text{op} \) will produce results after execution, usually \( \{ \text{res}_\text{op}, \text{flag}_\text{op} \} \).\(^3\) To facilitate the selection step in the next phase described in Sec. \( 4.4 \) below, we make all operation results shared in the replicated secret sharing. Note that each oblivious instruction cycle requires executing all operations in parallel to hide \( \text{op} \). Therefore, in the evaluation of some operations, we can use the (intermediate) result of

---

\(^1\)The * symbol in \( \Pi_{\text{read}}^{n,4} \) indicates that parties simultaneously but separately fetch each instruction filed using the same DPF key.

\(^2\)Each instruction takes two words in the memory.

\(^3\)The instruction \( \text{ans} \) is used to signify the program has finished, thus \( \text{res}_\text{ans} \) and \( \text{flag}_\text{ans} \) are not required.
Figure 7: Operations computation: Except for the answer, each $\tilde{op}$ has corresponding $\text{res}_{\tilde{op}}$ and $\text{flag}_{\tilde{op}}$. For ease of representation, the $\text{res}_{\tilde{op}}$ and $\text{flag}_{\tilde{op}}$ whose values must be $r_i$ and flag are omitted in the figure.
others to reduce the communication and computation. In addition, since some operations’ results without \( \text{res}_{\text{op}} \) or \( \text{flag}_{\text{op}} \) (i.e., these operations do not modify registers or condition flag), we convert \( \langle r_i \rangle \), \( \langle \text{flag} \rangle \) into \( \langle r_i \rangle^{\text{rep}} \), \( \langle \text{flag} \rangle^{\text{rep}} \) at the same round as instruction decoding, and set \( \text{res}_{\text{op}} := \langle r_i \rangle^{\text{rep}} \) (or \( \text{flag}_{\text{op}} := \langle \text{flag} \rangle^{\text{rep}} \) if \( \text{res}_{\text{op}} \) (or \( \text{flag}_{\text{op}} \)) is undefined.

We utilize Yao’s garbled circuits (GC) with free-XOR \([48]\) and half-gates \([44]\) techniques to build the low-latency evaluation protocol of bitwise operations and, or, xor, not, shl, shr. For other operations, our private evaluation protocol is based on the arithmetic secret sharing with DICF and DPF schemes. A detail description of our basic operation evaluation protocols appears in the Appendix A.

It is easy to see that performing all operations in parallel costs 3 online rounds by generating FSS keys and truncation pairs in the offline phase, except for the multiplication operations which need 4 online rounds in our basic solutions. We now show how to compress the multiplication operations to 3 online rounds. Notice that the last A2R step of \( \text{flag}_{\text{mul}} \) (and \( \text{flag}_{\text{mulh}} \)) computation causes the additional communication round. To avoid that, we let \( P_1 \) and \( P_3 \) jointly generate DICF keys \((K^{(1)}, K^{(2)})\) for the offset interval containment function \( f_{[0, 2^c-1, 1]}(x) : \mathbb{Z}_{2^c} \rightarrow \mathbb{Z}_{2^c} \) via MPC, and send \( K^{(1)}, K^{(2)} \) to \( P_2, P_4 \) respectively. We refer to \([47]\) for the detailed description of the distributed DICF key generation protocol. Since then four parties can directly evaluate the DICF keys to obtain \( \langle \text{flag}_{\text{mul}} \rangle^{\text{rep}} \) in online phase. Hence we can finish the operation evaluation phase in 3 online rounds, as summarized in Fig. 7.

### 4.4 Result Writing

The result writing phase is for updating the condition flag and registers. We begin by selecting the right results based on the generated \( \langle f_{\delta_1 - \psi_1, 1}(x) \rangle \) and \( \langle f_{\delta_2 - \psi_2, 1}(x) \rangle \), and then obliviously write them into \( \text{flag} \) and \( i \)-th register by o-write protocol.

Note that after previous phases, the MPC players already obtain \( \langle r_i \rangle^{\text{rep}} \), \( \langle \text{flag} \rangle^{\text{rep}} \), \{\( \langle \text{res}_k \rangle^{\text{rep}} \): \( k \in \mathbb{Z}_{2^c} \) \} and \{\( \langle \text{flag}_k \rangle^{\text{rep}} \): \( k \in \mathbb{Z}_{2^c} \) \} in the replicated secret sharing. Since \( P_1, P_2 \) share the point function \( f_{\delta_1 - \psi_1, 1}(x) \), and \( P_3, P_4 \) share the point function \( f_{\delta_2 - \psi_2, 1}(x) \), parties can jointly compute

\[
\langle \text{res} \rangle := \sum_{k=0}^{2^c} \text{res}_k^{(1)} \cdot \langle f_{\delta_1 - \psi_1, 1}(k) \rangle + \text{res}_k^{(3)} \cdot \langle f_{\delta_2 - \psi_2, 1}(k) \rangle ,
\]

\[
\langle \text{flag} \rangle := \sum_{k=0}^{2^c} \text{flag}_k^{(1)} \cdot \langle f_{\delta_1 - \psi_1, 1}(k) \rangle + \text{flag}_k^{(3)} \cdot \langle f_{\delta_2 - \psi_2, 1}(k) \rangle
\]

over the ring \( \mathbb{Z}_{2^c} \) without communication. MPC players then re-randomize the shares of \( \langle \text{res} \rangle \) and \( \langle \text{flag} \rangle \) to ensure their uniform distribution. It is obvious that the condition flag has been updated. To obliviously write \( \langle \text{res} \rangle \) into \( i \)-th register, the MPC players jointly invoke the o-write protocol with the index \( \langle i \rangle \), the old value \( \langle r_i \rangle \) and the new value \( \langle \text{res} \rangle \). In practice, we perform the o-write in the same round as the instruction fetching phase of the next oblivious instruction cycle.

### 4.5 Efficiency and Security

For efficiency, our system does not hide the overall run-time of the private RAM program. Therefore, if a RAM program with given inputs needs \( z \) cpu cycles, our system only requires \( 5z + 1 \) online rounds to obliviously evaluate it without the leaks of program and inputs other than run-time. If in some applications, this leakage is sensitive, we could hide the actual run-time by padding random cycles w.r.t. differential privacy.

In terms of the security of our overall system, since all the MPC primitives are proven in the UC framework, by the composition theorem, our PFE system is UC secure against one semi-honest adversary with static corruption. In particular, our PFE system utilizes the proposed distributed ORAM (cf. Sec. 3), 4-party addition/multiplication and share conversion protocol, garbled circuit, and FSS schemes from [47] as building blocks. The security of our distributed ORAM is proven in Sec. 3.3, and the rest protocols are all well-known MPC protocols whose security proof can be found in the literature accordingly.
Figure 8: Run-time (in different log scales) in LAN/WAN (bandwidth/RTT) setting. Where ours refers to our distributed ORAM protocol; floram refers to [34]; sqrt refers to [32]; circuit refers to [31].

5 Implementation and Benchmarks

Our distributed ORAM scheme and PFE for TinyRAM scheme are implemented in C++. We implement the DPF and DICF schemes with the full domain evaluation from [46] and interval containment
Table 4: Performance of our PFE scheme for different algorithms with different input sizes

<table>
<thead>
<tr>
<th>Input Size</th>
<th>32</th>
<th>64</th>
<th>128</th>
<th>256</th>
<th>1024</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Set Intersection</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td># of cycles</td>
<td>427</td>
<td>907</td>
<td>1603</td>
<td>3643</td>
<td>13747</td>
</tr>
<tr>
<td>Offline time (s)</td>
<td>12.97</td>
<td>27.27</td>
<td>50.01</td>
<td>110.70</td>
<td>420.72</td>
</tr>
<tr>
<td>Fetch time (s)</td>
<td>0.16</td>
<td>0.39</td>
<td>0.72</td>
<td>1.36</td>
<td>5.58</td>
</tr>
<tr>
<td>Decode time (s)</td>
<td>0.32</td>
<td>0.67</td>
<td>1.26</td>
<td>2.86</td>
<td>10.65</td>
</tr>
<tr>
<td>Eval. time (s)</td>
<td>8.37</td>
<td>17.46</td>
<td>30.47</td>
<td>70.68</td>
<td>269.10</td>
</tr>
<tr>
<td>Write time (s)</td>
<td>0.75</td>
<td>1.53</td>
<td>2.94</td>
<td>7.19</td>
<td>23.61</td>
</tr>
<tr>
<td>Total time (s)</td>
<td>22.54</td>
<td>47.32</td>
<td>85.39</td>
<td>192.79</td>
<td>729.66</td>
</tr>
</tbody>
</table>

| **Binary Search** |     |     |     |     |      |
| # of cycles | 12 | 64 | 75 | 96 | 111 |
| Offline time (s) | 0.39 | 2.03 | 2.27 | 2.91 | 3.24 |
| Fetch time (s) | 0.004 | 0.014 | 0.038 | 0.044 | 0.078 |
| Decode time (s) | 0.007 | 0.044 | 0.07 | 0.093 | 0.097 |
| Eval. time (s) | 0.20 | 1.25 | 1.49 | 1.87 | 2.21 |
| Write time (s) | 0.016 | 0.11 | 0.16 | 0.21 | 0.19 |
| Total time (s) | 0.62 | 3.45 | 4.06 | 5.13 | 5.81 |

| **Quick Sort** |     |     |     |     |      |
| # of cycles | 2860 | 6190 | 14656 | 29370 | 60090 |
| Offline time (s) | 90.52 | 198.31 | 479.70 | 918.13 | 1976.2 |
| Fetch time (s) | 1.23 | 2.67 | 6.03 | 12.45 | 25.33 |
| Decode time (s) | 2.28 | 4.81 | 10.93 | 20.27 | 44.71 |
| Eval. time (s) | 55.49 | 118.73 | 281.22 | 534.13 | 1151.9 |
| Write time (s) | 5.39 | 11.18 | 26.33 | 49.46 | 108.54 |
| Total time (s) | 154.91 | 335.66 | 804.19 | 1534.4 | 3306.5 |

gate from [47]. While performing the full domain evaluation, we prune the unnecessary branches for efficiency. Furthermore, for distributed ORAM, we follow the Tree Trimming approach in [34] to reduce the number of layers for FSS evaluation. AES-128 is chosen for PRF function, and we implement it by Intel’s AES-NI. EMP-toolkits [49] is used for GC evaluation. Our benchmarks are executed on a desktop with Intel(R) Core i7 8700 CPU @ 3.2 GHz running Ubuntu 18.04.2 LTS; with 6 CPUs, 32 GB Memory and 1TB SSD. Our local AES computations are implemented.

5.1 Distributed ORAM

We perform initialization, reading and writing benchmarks in 2 network environments for our distributed ORAM: local-area network (LAN, RTT: 1ms, bandwidth: 1Gbps) and wide-area network network (WAN, RTT: 6ms, bandwidth: 100Mbps). For the purpose of comparison, we also perform the same benchmarks for sqrt ORAM [32], circuit ORAM [31], and floram [34]. We make use of their implementations provided by the original authors of these works, which are identical to the ones previously reported by [34]. Note that, the writing operation in both our scheme and floram includes reading and updating two steps. In our experiment, the element size is 32 bits, and the number of ORAM elements is up to $2^{29}$. Our results are reported in Fig.8.

**Initialization.** Our construction outperforms all the other schemes in terms of initialization because it only requires building a replicated secret sharing of the database. Circuit ORAM has the slowest initialization process whose run-time is several order of magnitude longer than others, and thus it is impractical for large database sizes.

**Reading.** As we expected, the performance of our oblivious reading has a clear advantage over others for large database sizes. Compared with floram, which is also based on DPF technique, we have much lower bandwidth and fewer communication rounds because our o-read avoids distributed DPF key generation. Therefore, our scheme shows increasing advantage along with lower bandwidth and higher latency. Beyond roughly $2^{26}$ elements, our computation overhead of the full-domain evaluation of DPF
keys becomes the dominant factor in run-time. In this region, our performance is degrading to floram. However, since the bottleneck is at computation, our scheme can be further improved by using multiple cores.

**Writing.** The total run-time of our oblivious writing is comparable to floram. It is due to the fact that the additional cost of refreshing of floram is not noticeable on average. Compared with sqrt ORAM and circuit ORAM, our scheme shows significant performance improvement in LAN, although it is slower than circuit ORAM (and sqrt ORAM with less than $2^{23}$ elements) in WAN. In addition, our oblivious writing allows to move a lot of computation and communication burden to the offline phase. Thus, for large database sizes, our online run-time is superior to other solutions, including floram. More specifically, at $2^{22}$ elements, it has at least 6-fold concrete performance advantage over others in WAN. This property is very attractive for practical applications.

### 5.2 PFE for TinyRAM

To demonstrate the power of our PFE scheme, we implement a variety of algorithms using the simplified TinyRAM instruction set (cf. Table 3 at Supplemental Material), including set intersection, binary search and quick sort. Following the Von-Neumann architecture, both data and program stored in the same memory space. More specifically, for set intersection, similarly to [39] and [41], we use two ordered arrays of equal length as input, and count how many elements are shared in the two arrays. Denote the size of one input array as the input size of set intersection. For binary search and quick sort, there is only one input array. In addition, quick sort is implemented by a recursive rather than a for-loop structure in our experiment, so an extra $2 \log n$-word memory space is needed for the recursive stack. In this case, we reserve the last $2 \log n$-word space of memory as the function stack, and use the 16-th register as the stack register.

Note that the distributed DICF key generation leads expensive computation cost when the word size $\ell$ is large. Therefore, we use our basic scheme with 6z online rounds instead of the more intricate scheme with $5z + 1$ online rounds in the benchmark, where the word size $\ell$ is set to 32 bits. All experiments of our PFE scheme are carried out in the LAN setting. We record a detailed run-time profile as shown in the Table 4. Since those algorithms execute the different amount of instruction cycles depending on the different random input, our results are averages from 10 samples for each input size. As expected, in our PFE scheme, almost half of the overhead is moved to the offline stage, and the operation evaluation phase dominates the online run-time. The average time per oblivious cycle is almost stable for input size varied between $2^5$ and $2^{10}$, whose trend is same as our distributed ORAM. Therefore, we can predict that only beyond $2^{19}$ input size, the cost of per oblivious cycle may increase significantly.

Furthermore, we compare with [39] and [41] on set intersection. Our running instructions is compiled from the example provided by [39], as follows:

\[
\begin{align*}
I_0 &: \text{read } %0,0 \\
I_1 &: \text{read } %1,0 \\
I_2 &: \text{read } %2,0 \\
I_3 &: \text{read } %3,0 \\
I_4 &: \text{cmpa } %1,%0 \\
I_5 &: \text{cnjmp } I_{22} \\
I_6 &: \text{cmpe } %3,%2 \\
I_7 &: \text{cnjmp } I_{22} \\
I_8 &: \text{load } %30,%0 \\
I_9 &: \text{load } %31,%2 \\
I_{10} &: \text{cmp } %31,%30 \\
I_{11} &: \text{cnjmp } I_6 \\
I_{12} &: \text{add } %0,%0,1 \\
I_{13} &: \text{add } %2,%2,1 \\
I_{14} &: \text{add } %29,%29,1 \\
I_{15} &: \text{jmp } I_4 \\
I_{16} &: \text{cmpe } %30,%31 \\
I_{17} &: \text{cnjmp } I_{20} \\
I_{18} &: \text{add } %2,%2,1 \\
I_{19} &: \text{jmp } I_4 \\
I_{20} &: \text{add } %0,%0,1 \\
I_{21} &: \text{jmp } I_4 \\
I_{22} &: \text{ans } %29
\end{align*}
\]

where,

- $I_i$ stands for the memory address of $i$-th instruction.
- $%a$ denotes the $a$-th register.
Table 5: Run-time (s) of PFE schemes for set intersection

<table>
<thead>
<tr>
<th>Input Size</th>
<th>64</th>
<th>256</th>
<th>1024</th>
<th>4096</th>
</tr>
</thead>
<tbody>
<tr>
<td>WGMK16 [39]</td>
<td>58.35</td>
<td>324.09</td>
<td>3068.19</td>
<td>-</td>
</tr>
<tr>
<td>Marcel17 [41]</td>
<td>6.43</td>
<td>44.12</td>
<td>1346.82</td>
<td>-</td>
</tr>
<tr>
<td>Ours</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>best</td>
<td>online</td>
<td>18.10</td>
<td>68.34</td>
<td>275.65</td>
</tr>
<tr>
<td></td>
<td>total</td>
<td>42.53</td>
<td>161.03</td>
<td>649.46</td>
</tr>
<tr>
<td>worst</td>
<td>online</td>
<td>32.56</td>
<td>135.83</td>
<td>547.16</td>
</tr>
<tr>
<td></td>
<td>total</td>
<td>77.99</td>
<td>331.69</td>
<td>1303.22</td>
</tr>
</tbody>
</table>

Table 6: A feasible solution for 2-collusion tolerance, where $p_1 := 3$, $p_2 := 5$, $n := 7$.

<table>
<thead>
<tr>
<th></th>
<th>$P_1$</th>
<th>$P_2$</th>
<th>$P_3$</th>
<th>$P_4$</th>
<th>$P_5$</th>
<th>$P_6$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x^{(1)}$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>$x^{(2)}$</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>$x^{(3)}$</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$x^{(4)}$</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>$x^{(5)}$</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Let the input size be $m$. Since neither [39] or [41] provides test data, to avoid the unfairness from the input, we record our run-time for the best computational complexity with $12m + 7$ cycles and the worst computational complexity with $24m - 5$ cycles. In the former case, two input arrays are identical; in the latter case, there is no share element in two input arrays, and the $i$-th element of each array is smaller than the $(i+1)$-th element of both arrays. As shown in Table 5, we achieve an overwhelming performance advantage on large data sets. At the input size $2^{10}$, even if our scheme uses the worst input case, its total run-time is comparable to [41] and has a roughly 2-fold advantage over [39].

6 Notes on Scalability

Throughout this paper, our scheme assumes a scenario of 4 MPC parties with 1 corrupted. In this section, we will discuss the scaling security of our scheme.

It is easy to see, the main barrier to scaling security is our distributed ORAM, whose setting is now restricted by 2-party DPF and our replicated secret sharing scheme as defined in Section 2. Fortunately, besides the efficient 2-party DPF, Boyle et al. also present the nontrivial construction of $p$-party DPF for $p \geq 3$ in [37]. For $t$ collision tolerance, assume our scaling scheme uses $p_1$-party DPF for $o$-read and $p_2$-party DPF for $o$-write. It is obvious to require $p_1, p_2 > t$. Accordingly, the replicated secret sharing scheme should satisfy the following conditions: (a) the secret is split into $p_2$ shares, (b) each share is holed by $p_1$ different parties, and (c) any $t$ parties cannot reconstruct the secret.$^4$

Naively, we can set $p_1 := p_2 := t + 1$, and introduce $(t + 1)^2$ MPC parties to achieve $t$-privacy. In this solution, for $i \in [t + 1]$, parties in the subset $\mathcal{P}_i := \{P_{(t+1)i-t}, \ldots, P_{(t+1)i}\}$ hold the same share, denoted as $x^{(i)}$, by abusing notations, of the secret $x$ such that $\sum_{i=1}^{t+1} x^{(i)} = x$. In o-read operation, parties in $\mathcal{P}_i$ jointly evaluate the $p_1$-party DPF for $f_{\phi_{1}}$ to obliviously extract the $i$-th share of target element, and thus a total of $p_2$ different $p_1$-party DPFs are required. In o-write operation, parties in $\mathcal{P}_i$ evaluate the $i$-th keys of the $p_2$-party DPFs for $f_{r_{1i}}, f_{r_{1v}}$ to obliviously and identically update the $i$-th share of target element, and thus only 2 $p_2$-party DPFs are required.

Furthermore, adjusting the value of $p_1, p_2$, we can scale down the number of MPC parties, denoted by $n$. Take the scenario of 2 corrupted parties as an example. Table 6 shows a feasible solution with $n < (2 + 1)^2$, where the element $s_{i,j} = 0$ in the $i$-th row and the $j$-th column indicates that $x^{(i)}$ is holed by $P_j$. Its $o$-read and $o$-write operations are similar to above. In fact, scaling our scheme for

$^4$Note that, it is not required that any $t + 1$ parties can reconstruct the secret.
\( t \)-collusion is an optimization problem, which can be formally represented as follows:

\[
\begin{align*}
&\begin{cases}
  p_1, p_2 > t; \\
  n > p_1, p_2;
\end{cases} \\
&\sum_{i=1}^{p_2} \sum_{j=1}^{n} s_{i,j} = (n - p_1) \cdot p_2; \\
&\sum_{j=1}^{n} s_{i,j} = n - p_1, \text{ for } i \in [p_2]; \\
&\sum_{i=1}^{p_2} \prod_{j \in Q} s_{i,j} > 0, \text{ for } Q \in \{U | U \subset [n], |U| = t\}.
\end{align*}
\]

7 Related Works

7.1 Distributed ORAM

ORAM-based MPC protocols enable dramatic efficiency improvements in processing large, secret databases. Unfortunately, the traditional ORAM and multi-server ORAM has a main obstacle to efficiency in MPC context: a trusted client, which leads to expensive overhead when instantiating it using generic MPC. Therefore, a number of researches focus on designing distributed ORAM schemes that are tailor-made for the MPC scenario. [31] proposes the well-known MPC-friendly circuit ORAM, which achieves asymptotic circuit-complexity \( O(\lambda \log^3 n + \lambda \ell \log n) \) per access. Its key idea is to complete the eviction algorithm within a single scan of the current eviction path, while being as aggressive as possible. [32] revisits the classical square-root ORAM and proposed a distributed variant with a recursive position map instead of MPC hash function. Although its amortized complexity \( O(\lambda \ell \sqrt{n} \log n) \) is worse than 2PC circuit ORAM, it has better practical performance. 2PC Floram [34] needs fewer MPC operations, compared with the prior distributed ORAM designs. However, to achieve simultaneous read and write capabilities, [34] requires a refresh operation with \( O(n\ell) \) communication cost after every \( O(\sqrt{n\ell}/\lambda \log n) \) writes to achieve simultaneous read and write capabilities. [35] extends the Floram of [34] to 3PC distributed ORAM with constant round complexity. They use 3PC DPF technology to get rid of the refresh operation, but 3PC DPF itself causes \( O((\lambda + \ell)\sqrt{n}) \) communication and leads worse practical performance. The bandwidth of another 3PC distributed ORAM scheme [33] is competitive to Floram. Nevertheless, they require heavy initialization phase as well as 2PC circuit ORAM. [36] achieves the best asymptotic communication complexity \( O((\lambda + \ell) \log n) \) in the literature. But they use costly SISO-PRP technique and require linear round complexity in AND gates depth.

7.2 CPU emulation via MPC

Several CPU emulation schemes (e.g. [39, 40]) have been proposed in an attempt to improve the performance of privacy-preserving RAM program evaluation. However, their practical design and implementation hold only for private input but public program setting. For instance, in the inefficient basic scheme of [39], they use garbled universal circuits to securely execute all possible instructions at each step, and that hides the program; in its optimized system, they pad branches and map instructions to steps to reduce the number of possible instructions at each step, and that maintains the branching privacy but incurs the risk of the program leakage. Furthermore, [41] aims to private function evaluation. But their solution for RAM-model computation comes with polylogarithmic overhead \( O(z \log^3 n) \) while our protocol only requires \( O(z \log n) \).

8 Conclusion

We present a 4-party PFE system for RAM program. We simulate a RISC machine in MPC context that can naturally support a simplified version of TinyRAM [38]. As a building block, we design a DPF-based distributed ORAM scheme with \( O(\log n) \) communication per access, which may be of independent interests. In the future, we will upgrade our PFE for malicious security and securely reduce the number of memory operations’ evaluation aided by our private registers.

A Detailed Operation Evaluation

In this section, we show how to obliviously evaluate each operation via MPC in detail.
A.1 Bitwise Operations

In our system, data are shared over the ring $\mathbb{Z}_{2^\ell}$. The low-latency bitwise operations, denoted as $D := \{\text{and}, \text{or}, \text{xor}, \text{not}, \text{shl}, \text{shr}\}$, are built using Yao’s garbled circuits (GC) with free-XOR and half-gates techniques, and result in two rounds of share conversion communication overhead. In the first round, the MPC players collaborate with the evaluator to convert their additive shares into GC’s keys. Then the evaluator locally evaluates the aggregated additive shares and calculates the objective boolean function locally. In the second round, all parties reconstruct the replicated secret shares of the results by opening GC output label. More specifically, it works as follows.

$\text{res}_{\text{op}\in D}$. In our protocol $P_1$ plays the GC garbler and $P_3$ plays the GC evaluator. All parties perform A2Y to convert $(rj)$ and $(a)$ to shared Yao’s select wire keys $\{k_{rj[i]}{q}\}_{i\in \mathbb{Z}_t}$, $\{k_{a[i]}{q}\}_{i\in \mathbb{Z}_t}$ for $q \in [4]$, and send to $P_3$. Instead of an adder circuit, $P_1$ picks $\text{res}_{\text{op}}^{(1)}$ and generates the circuit with the function $\text{res}_{\text{op}}^{(3)} := F_{\text{op}}(\sum_{q=1}^4 rj{q}, \sum_{q=1}^4 a{q}) - \text{res}_{\text{op}}^{(1)}$. $P_1$ sends the select wire keys of $\text{res}_{\text{op}}^{(1)}$ to $P_3$. After evaluating the GC circuit locally, $P_3$ holds the Yao’s share $\{k_{\text{res}[i],t}{q}\}_{i\in \mathbb{Z}_t}$ and $P_1$ holds another Yao’s share $\{k_{0,0,i}{q}\}_{i\in \mathbb{Z}_t}$. Finally, $P_1$ sends the permute bits of $\{k_{0,0,i}\}_{i\in \mathbb{Z}_t}$ to both $P_3$ and $P_4$, sends $\text{res}_{\text{op}}^{(1)}$ to $P_2$, $P_3$ and $P_4$ then xor and concatenate permute bits to obtain $\text{res}_{\text{op}}^{(3)}$.

$\text{flag}_{\text{op}\in D}$. $P_1$ picks $\text{flag}_{\text{op}}^{(1)}$ and generates the GC circuit for the function

$$\text{flag}_{\text{op}}^{(3)} := F_{\text{flag}}(F_{\text{op}}(\sum_{q=1}^4 rj{q}, \sum_{q=1}^4 a{q})) - \text{flag}_{\text{op}}^{(1)},$$

where $F_{\text{flag}}$ is set to the most/least significant bit of the variable for shl/shr operations, and $(\text{res}_{\text{op}} = 0)$ for the other operations. Then all parties execute $(\text{res}_{\text{op}})_{\text{rep}}$ calculation process to get $(\text{flag}_{\text{op}})_{\text{rep}}$.

A.2 Arithmetic Operations

These are various unsigned and signed integer operations, including addition (add), subtraction (sub) and multiplication (mul, umulh and smulh). In each case, the corresponding $\text{flag}_{\text{op}}$ is set to 1 if an arithmetic overflow occurs and to 0 otherwise.

add&sub: For $\text{op} \in \{\text{add, sub}\}$, we compute $(rj + a)$ or $(rj - a)$ over the ring $\mathbb{Z}_{2^\ell}$ and store the result to $(\text{res}_{\text{op}})_{\text{rep}}$ after the A2R share conversion. To determine $\text{flag}_{\text{op}}$, we first convert the shared $rj$ and $a$ from $\mathbb{Z}_{2^\ell}$ to $\mathbb{Z}_{2^\ell +1}$ using the DICF scheme, and then compare the addition or subtraction result over the ring $\mathbb{Z}_{2^\ell +1}$ with $2^{\ell - 1}$. $\text{flag}_{\text{op}}$ is set to 1 if the result is greater than $2^{\ell-1}$ and to 0 otherwise. More specifically, it works as follows.

$\text{res}_{\text{add}}$ (or $\text{res}_{\text{sub}}$). The MPC players add their local shares to obtain $\text{res}_{\text{add}}$ in a shared form. That is, $\text{res}_{\text{add}}^{(q)} := rj{q} + a{q}$ (mod $2^\ell$) for $q \in [4]$. After that, parties spend a communication round to convert $(\text{res}_{\text{add}})$ into $(\text{res}_{\text{add}})_{\text{rep}}$.

$\text{flag}_{\text{add}}$ (or $\text{flag}_{\text{sub}}$). When shares are individually converted from $\mathbb{Z}_{2^\ell}$ to $\mathbb{Z}_{2^\ell +1}$, the extra carry-in value $2^\ell$ may appear. Thus the MPC players need to check whether $\sum_{q=1}^4 rj{q}, \sum_{q=1}^4 a{q} \in [0, 2^\ell - 1]$ over the ring $\mathbb{Z}_{2^\ell +1}$, and then correct the individually converted shares based on the results of interval containment computation. In detail, $P_1$ generates DICF keys of the offset interval containment functions $f_{0,2^\ell-1,r}^\ell(x) : \mathbb{Z}_{2^\ell +1} \rightarrow \mathbb{Z}_{2^\ell +1}$, where $r \leftarrow \mathbb{Z}_{2^\ell +1}$ is random offset, and then distributes DICF keys to $P_2, P_3$. Similarly, $P_2$ generates DICF keys of $f_{0,2^\ell-1,l}^\ell(x)$ and distributes keys to $P_1, P_3$. After that, $P_1, P_3$ hold $\|f_{0,2^\ell-1,r}^\ell(x)\|$, and $P_2, P_3$ hold $\|f_{0,2^\ell-1,l}^\ell(x)\|$. Parties then open $(rj') := (rj) + r$ (mod $2^{\ell+1}$) to $P_1, P_3$, $(a') := (a) + v$ (mod $2^{\ell+1}$) to $P_2, P_4$, and jointly compute

$$\langle rj \rangle := (rj) - (1 - \langle f_{0,2^\ell-1,r}^\ell(x) \rangle) \cdot 2^\ell \pmod{2^{\ell+1}},$$

$$\langle a \rangle := (a) - (1 - \langle f_{0,2^\ell-1,l}^\ell(x) \rangle) \cdot 2^\ell \pmod{2^{\ell+1}},$$

by evaluating DICF keys in local.

Later, the MPC players jointly compare the $2^\ell - 1$ with the addition (or subtraction) result of $rj'$ and $a'$ over $\mathbb{Z}_{2^\ell +1}$. More specifically, we let $P_1$ generate DICF keys of $f_{0,2^\ell-1,l}^\ell(x) : \mathbb{Z}_{2^\ell +1} \rightarrow \mathbb{Z}_{2^\ell}$, where $t \leftarrow \mathbb{Z}_{2^\ell +1}$ is random offset, and then distribute keys to $P_2, P_4$. The four MPC players open
\textbf{Protocol }\Pi_{mul}'\textbf{.}

- Upon receiving \((\text{Mul}, \text{sid}, r_j(q), a(q))\):
  - \(P_\ell \in (1, 2)\) sends \((\text{sid}, r_j(q))\) to \(P_{3-\ell}\), \((\text{sid}, a(q))\) to \(P_{q+2}\);
  - \(P_\ell \in (3, 4)\) sends \((\text{sid}, r_j(q))\) to \(P_{7-\ell}\), \((\text{sid}, a(q))\) to \(P_{q-2}\);

- Upon receiving \((\text{sid}, r_j(q)), (\text{sid}, a(q+2))\), \(P_\ell \in (1, 2)\) sets:
  - \(\text{res}^{(q)} := (r_j(3-q) + r_j(q)) \cdot (a(q) + a(q+2)) \pmod{2^\ell}\);

- Upon receiving \((\text{sid}, r_j(q)), (\text{sid}, a(q-3))\), \(P_\ell \in (3, 4)\) sets:
  - \(\text{res}^{(q)} := (r_j(7-q) + r_j(q)) \cdot (a(q) + a(q-3)) \pmod{2^\ell}\);

\begin{align*}
\langle y_t \rangle := \langle \hat{r}_j \rangle + \langle \hat{a} \rangle + t \pmod{2^\ell+1} & \to P_2, P_4. \quad \text{Upon receiving the DICF keys and } y_t, P_2 \text{ and } P_4 \text{ jointly compute } \\
[\text{flag}_{\text{add}}] := 1 - [f^{IC}_{0,2^\ell-1,t}(y_t)] \pmod{2^\ell} & \text{by evaluating DICF keys in local. Finally, to construct } \langle \text{flag}_{\text{add}} \rangle^{\text{rep}}\text{ among four parties, } P_2 \text{ sends its share of } \text{flag}_{\text{add}} \text{ to } P_1 \text{ and } P_3 \text{ sends its share to } P_3.
\end{align*}

\textbf{mul:} This operation computes \((r_j \cdot a)\) over the ring \(\mathbb{Z}_{2^\ell}\) and stores the result to \(\text{res}_{\text{mul}}\) in the replicated-shared form. Analogous to the flag_{add} determination, to check whether \(r_j \cdot a \notin \mathbb{Z}_{2^\ell}\), we first convert the shared \(r_j\) and \(a\) from \(\mathbb{Z}_{2^\ell}\) to \(\mathbb{Z}_{2^2\ell}\), then compare the multiplication result over the ring \(\mathbb{Z}_{2^2\ell}\) with \(2^\ell - 1\). \text{flag}_{\text{mul}}\ is set to 1 if the result is greater than \(2^\ell - 1\) and to 0 otherwise. More specifically, it works as follows.

\textbf{res}_{\text{mul}}: The MPC players jointly invoke our 4-party multiplication protocol \(\Pi_{mul}'\) (cf. Fig. 9) to compute \(\langle \text{res}_{\text{mul}} \rangle := \langle r_j \rangle \cdot \langle a \rangle \pmod{2^\ell}\), and then convert \(\langle \text{res}_{\text{mul}} \rangle\) to \(\langle \text{res}_{\text{mul}} \rangle^{\text{rep}}\).

\textbf{flag}_{\text{mul}}: First, parties convert \(\langle r_j \rangle, \langle a \rangle\) from \(\mathbb{Z}_{2^\ell}\) to \(\mathbb{Z}_{2^2\ell}\). This step is similar to flag_{add} computation, and the main difference is that there are extra two carry-in bits in this case. That is, for each value, parties require two functions of \(f_{0,2^\ell-1}^{IC}(x) : \mathbb{Z}_{2^\ell+1} \to \mathbb{Z}_{2^{2\ell}}\) and \(f_{0,2^2\ell-1}^{IC}(x) : \mathbb{Z}_{2^{2\ell}+2} \to \mathbb{Z}_{2^{2\ell}}\) to detect the values of \(\ell\)-th bit and \(\ell + 1\)-th bit, respectively. Denote the converted shares as \(\langle \hat{r}_j \rangle\) and \(\langle \hat{a} \rangle\). Parties then jointly compute \(\langle y \rangle := \langle r_j \rangle \cdot \langle a \rangle \pmod{2^{2\ell}}\) and compare \(\langle y \rangle\) with \(2^{2\ell}\) to get \(\langle \text{flag}_{\text{mul}} \rangle^{\text{rep}}\) as shown in Fig. 10.

\textbf{umulh:} This operation computes \((r_j \cdot a)\) over the ring \(\mathbb{Z}_{2^2\ell}\) and stores the most significant \(\ell\) bits of result to \(\text{res}_{\text{umulh}}\) in the replicated-shared form. It requires a truncation and several share conversions among \(\mathbb{Z}_{2^2\ell}\) and \(\mathbb{Z}_{2^\ell}\). Besides, \text{flag}_{\text{umulh}}\ is determined by whether \(r_j \cdot a \geq 2^\ell\), i.e., \text{flag}_{\text{umulh}} = \text{flag}_{\text{mul}}.\) More specifically, it works as follows.

\textbf{res}_{\text{umulh}}: The MPC players first convert \(\langle r_j \rangle, \langle a \rangle\) from \(\mathbb{Z}_{2^2\ell}\) to \(\mathbb{Z}_{2^\ell}\), and multiply them over the ring \(\mathbb{Z}_{2^\ell}\). This step to obtain \(\langle r_j \rangle \cdot \langle a \rangle \in \mathbb{Z}_{2^\ell}\) is the same as in \text{flag}_{\text{mul}}.\) In other words, parties can straightforwardly use the multiplication result \(\langle y \rangle\) from \text{flag}_{\text{mul}} computation. After that, parties jointly truncate \(\langle y \rangle\) to get the shared value of \(y\)'s most significant \(\ell\) bits over the ring \(\mathbb{Z}_{2^\ell}\). More specifically, similar to the share conversion, when shares are individually truncated (i.e., divided by \(2^\ell\)), two carry-in bits may be lost. To correct this error, parties begin by generating a truncation pair \(\langle t \rangle^{\text{rep}}, \langle t' \rangle := \langle t \cdot 2^\ell \rangle\) over the ring \(\mathbb{Z}_{2^\ell}\), and then open \(\langle y_t \rangle := \langle y \rangle - \langle t' \rangle \pmod{2^{2\ell}}\). After that, parties can jointly compute \(\langle y_t \rangle^{\text{rep}} := \langle t \rangle^{\text{rep}} + y_t >> \ell \pmod{2^{2\ell}}\), hence, \(y_t = y >> \ell\). Finally, \(\langle \text{res}_{\text{umulh}} \rangle^{\text{rep}}\) can be obtained by individually removing the most significant \(\ell\) bits of \(\langle y_t \rangle^{\text{rep}}\).

\textbf{flag}_{\text{umulh}}: For \(q \in [4]\), the player \(P_q\) sets \(\text{res}_{\text{umulh}}^{(q)} := \text{res}_{\text{mul}}^{(q)}\).

\textbf{smulh:} This operation computes signed \((r_j \cdot a)\) over the ring \(\mathbb{Z}_{2^2\ell}\) and stores the most significant \(\ell\) bits of result to \(\text{res}_{\text{smulh}}\) in the replicated-shared form. Note that, \(r_j\) and \(a\) are two's complement representation of the integers \(r_j\) and \(a\). To increase their bit-length to \(2^\ell\), we should securely add the sign bits of \(r_j\) and \(a\) to their left. \text{flag}_{\text{smulh}}\ is set to 1 if the result is in \([2^{\ell-1}, 2^{\ell-1} - 1]\), and is set to 0 otherwise. More specifically, it works as follows.

\textbf{res}_{\text{smulh}}: The MPC players first check whether \(r_j, a \in [0, 2^{\ell-1}]\) to extract their sign bits \([s_r]\) and \([s_a]\) respectively. In detail, \(P_1\) and \(P_2\) generate DICF keys for functions \(f_{0,2^{\ell-1},v}^{IC}(x), f_{0,2^{\ell-1},r}^{IC}(x) : \mathbb{Z}_{2^\ell} \to \mathbb{Z}_{2^\ell}\).
Offline phase:
• Upon initialization, $P_1$ does:
  - Set $v_k \leftarrow \mathbb{Z}_{2^{t+k}}, v^{(1)}_k := v_k - \text{PRF}_{\text{PRF}}^{2^{t+k}}(\text{sid}, 0), k \in [2]$;
  - Set $(K^{(1)}, K^{(2)}_k) \leftarrow \text{DICF.Gen}^{\text{IC}}_{0, k, 2^t-1}(1, f^{\text{IC}}_{R, 0, k, 2^t-1, v_k}), k \in [2]$;
  - Generate $t \leftarrow \mathbb{Z}_{2^{t}}$;
  - Set $(K^{(1)}_t, K^{(2)}_t) \leftarrow \text{DICF.Gen}^{\text{IC}}_{0, k, 2^t-1}(1, f^{\text{IC}}_{R, 0, k, 2^t-1, t})$;
  - Send $(\text{sid}, (K^{(1)}_t, K^{(2)}_t))$ to $P_2$;
  - Send $(\text{sid}, (K^{(2)}_t, K^{(2)}_t))$ to $P_3$;
• Upon initialization, $P_2$ does:
  - Set $r_k \leftarrow \mathbb{Z}_{2^{t+k}}, r^{(1)}_k := r_k - \text{PRF}_{\text{PRF}}^{2^{t+k}}(\text{sid}, 0), k \in [2]$;
  - Set $(K^{(1)}_r, K^{(2)}_{r, k}) \leftarrow \text{DICF.Gen}^{\text{IC}}_{0, k, 2^t-1}(1, f^{\text{IC}}_{R, 0, k, 2^t-1, r_k}), k \in [2]$;
  - Send $(\text{sid}, (K^{(1)}_r, K^{(2)}_{r, k}))$ to $P_1$, $(\text{sid}, (K^{(2)}_r, K^{(2)}_{r, k}))$ to $P_3$;
• Upon initialization, $P_3$ sets $v^{(2)}_k := \text{PRF}_{\text{PRF}}^{2^{t+k}}(\text{sid}, 0), k \in [2]$;
• Upon initialization, $P_4$ sets $r^{(2)}_k := \text{PRF}_{\text{PRF}}^{2^{t+k}}(\text{sid}, 0), k \in [2]$;

Online phase:
• Upon receiving $(\text{FLAGMULL}, \text{sid}, r^{(q)}_j, a^{(q)})$ from the environment $Z$, player $P_q$, $q \in \{1, 2\}$ does:
  - For $k \in [2]$:
    * Set $w_{1,q,k} \leftarrow \text{PRF}_{\text{PRF}}^{2^{t+k}}(\text{sid}, 1)$;
    * Set $w_{2,q,k} \leftarrow \text{PRF}_{\text{PRF}}^{2^{t+k}}(\text{sid}, 2)$;
    * Set $r^{(q)}_j := r^{(q)}_j + w_{1,q,k}$ (mod $2^{t+k}$);
    * Set $a^{(q)}_k := a^{(q)} + v^{(1)}_k + w_{2,q,k}$ (mod $2^{t+k}$);
    - Send $(\text{sid}, r^{(q)}_j, a^{(q)}_k)$ to $P_1$, $(\text{sid}, a^{(q)}_k, a^{(q)}_2)$ to $P_2, P_3$;
• Upon receiving $(\text{FLAGMULL}, \text{sid}, r^{(q)}_j, a^{(q)})$ from the environment $Z$, player $P_q$, $q \in \{3, 4\}$ does:
  - For $k \in [2]$:
    * Set $w_{1,q-2,k} \leftarrow \text{PRF}_{\text{PRF}}^{2^{t+k}}(\text{sid}, 1)$;
    * Set $w_{2,q-2,k} \leftarrow \text{PRF}_{\text{PRF}}^{2^{t+k}}(\text{sid}, 2)$;
    * Set $r^{(q)}_j := r^{(q)}_j + r^{(q)} - w_{1,q,k}$ (mod $2^{t}$);
    * Set $a^{(q)}_k := a^{(q)} - w_{2,q-2,k}$ (mod $2^{t}$);
    - Send $(\text{sid}, r^{(q)}_j, a^{(q)}_k)$ to $P_1$, $(\text{sid}, a^{(q)}_k, a^{(q)}_2)$ to $P_2, P_3$;
• Upon receiving $(\text{sid}, r^{(3-q)}_j, r^{(3-q)}_j, r^{(3-q)}_j)$ from $P_{3-q}$, $(\text{sid}, r^{(3)}_j, r^{(3)}_j, r^{(3)}_j)$ from $P_3$, and $(\text{sid}, r^{(4)}_j, r^{(4)}_j)$ from $P_4, P_q$, $q \in \{1, 3\}$ does:
  - Set $r^{(q)}_j := \sum_{q=1}^{4} r^{(q)}_j$ (mod $2^{t+k}$) for $k \in [2]$;
  - Set $\beta_{r,j}^{(q)} \leftarrow \text{DICF.Eval}^{\text{IC}}_{0, k, 2^t-1}(q, K^{(1)}_{r,k}, r^{(q)}_j), k \in [2]$;
  - Set $r^{(q)}_j := r^{(q)}_j + \sum_{k=1}^{2} \beta_{r,j}^{(q)} \cdot k \cdot 2^t$ (mod $2^{2t}$);
  - Set $a^{(q)} := a^{(q)} - \{q/2\} \cdot 2^t$ (mod $2^{2t}$);
  - Invoke $\Pi^{\text{null}}_{\text{flag}}$ with $(\text{MUL}, \text{sid}, r^{(q)}_j, a^{(q)}_k)$ to get $y^{(q)}_j$;
  - If $q = 1$, $y^{(q)}_j := y^{(q)} + t$ (mod $2^{2t}$);
  - Send $(\text{sid}, y^{(q)}_j)$ to $P_2$ and $P_4$;
• Upon receiving $(\text{sid}, a^{(1)}_1, a^{(2)}_2)$ from $P_1$, $(\text{sid}, a^{(2)}_1, a^{(2)}_2)$ from $P_2$, $(\text{sid}, a^{(7-q)}_1, a^{(7-q)}_2)$ from $P_3$, and $(\text{sid}, a^{(7)}_1, a^{(7)}_2)$ from $P_4, P_q$, $q \in \{2, 4\}$ does:
  - Set $a^{(q)}_{r,k} := \sum_{q=1}^{4} a^{(q)}_{r,k}$ (mod $2^{2t}$) for $k \in [2]$;
  - $\beta^{(q)}_{r,k} \leftarrow \text{DICF.Eval}^{\text{IC}}_{0, k, 2^t-1}(q - 2, K^{(1)}_{r,k}, a^{(q)}_{r,k}), k \in [2]$;
  - Set $r^{(q)}_j := r^{(q)}_j - \{q/2\} \cdot 2^t$ (mod $2^{2t}$);
  - Set $a^{(q)} := a^{(q)} + \sum_{k=1}^{2} \beta^{(q)}_{r,k} \cdot k \cdot 2^t$ (mod $2^{2t}$);
  - Invoke $\Pi^{\text{null}}_{\text{flag}}$ with $(\text{MUL}, \text{sid}, r^{(q)}_j, a^{(q)}_k)$ to get $y^{(q)}_j$;
  - Send $(\text{sid}, y^{(q)}_j)$ to $P_6-q$;
• Upon receiving $(\text{sid}, y^{(k)}_j)$ from $P_{\in [4]/q}, P_{\in [2, 4]}$ does:
  - Set $y^{(q)}_j := y^{(1)}_j + y^{(2)}_j + y^{(3)}_j + y^{(4)}_j$ (mod $2^{2t}$);
  - Set $\text{flag}_{\text{null}} \leftarrow \text{DICF.Eval}^{\text{IC}}_{0, k, 2^t-1}(q/2, K^{(2)}_{j, q, 2^t-1}), y_j$;
  - Send $\text{flag}_{\text{null}}$ to $P_{q-1}$.

Figure 10: null overflow detection protocol $\Pi^{\text{null}}_{\text{flag}}$. 
specifically, where \( v, r \leftarrow \mathbb{Z}_{2^t} \) are random offsets. \( P_1 \) (and \( P_2 \)) then distribute their DICF keys to \( P_2, P_3 \) (and \( P_1, P_3 \)). Parties then open \( \langle r' \rangle := (r_j) + r \mod (2^2) \) to \( P_1, P_3 \), \( \langle a' \rangle := (a) + v \mod (2^2) \) to \( P_2, P_3 \), and jointly compute
\[
\langle s_r \rangle := 1 - \left\lfloor \frac{r'_{t-1}}{2^t} \right\rfloor (\mod 2^t),
\langle s_a \rangle := 1 - \left\lfloor \frac{a'_{t-1}}{2^t} \right\rfloor (\mod 2^t)
\]
by locally evaluating DICF keys. Meanwhile, four parties convert \( \langle r_j \rangle \) and \( \langle a \rangle \) from \( \mathbb{Z}_{2^t} \) to \( \mathbb{Z}_{2^{2t}} \) as unsigned integers, i.e., they can copy the converted shares of \( \langle r_j \rangle \) and \( \langle a \rangle \) from flag_{smulh} computation. After that, the MPC players obtain the sign extension values
\[
\langle r_j \rangle := (r_j) + (2^{2t} - 2^t) \cdot \langle s_r \rangle (\mod 2^t),
\langle a \rangle := (a) + (2^{2t} - 2^t) \cdot \langle s_a \rangle (\mod 2^t)
\]
without communication. Next, parties compute \( \langle y_s \rangle := \langle r_j \rangle \cdot \langle a \rangle (\mod 2^t) \), and then truncate the most significant \( \ell \) bits of \( \langle y_s \rangle \) to get \( \langle \text{res}_{\text{smulh}} \rangle \) by the same method as in flag_{smulh} computation. flag_{smulh} The MPC players detect whether \( y_s \in [2^t, 2^{2t} - 2^t - 1] \) based on DICF schemes. More specifically, \( P_1 \) generates DICF keys of the functions \( \ell_{t-1, 2^{2t}-2^t-1} : \mathbb{Z}_{2^{2t}} \rightarrow \mathbb{Z}_{2^t} \) where \( t \leftarrow \mathbb{Z}_{2^t} \) is random offset, and distributes them to \( P_2, P_4 \). Parties then open \( \langle y_t \rangle := \langle y_s \rangle + t \mod (2^t) \) to \( P_2, P_4 \). Upon receiving \( y_t, P_2 \) and \( P_4 \) jointly compute
\[
\langle \text{flag}_{\text{smulh}} \rangle := 1 - \left\lfloor \ell_{t-1, 2^{2t}-2^t-1} (\langle y_t \rangle) \right\rfloor (\mod 2^t)
\]
by locally evaluating DICF keys. Finally, to construct \( \langle \text{flag}_{\text{add}} \rangle \) among four parties, \( P_2 \) sends its share of \( \text{flag}_{\text{smulh}} \) to \( P_1 \) and \( P_4 \) sends its share to \( P_3 \).

A.3 Comparison Operations

We support the comparison of unsigned and signed integer, including cmpa, cmpae, cmpge and cmpg operations. Each comparison result is assigned to the corresponding flag_{op} and does not modify any register. As mentioned before, we set \( \langle \text{res}_{\text{op}} \rangle := \langle \text{res} \rangle \) to hide the operation.

**cmpa:** This operation checks if \( r_j \) is equal to \( a \). We use the DPF scheme to obliviously detect whether its equivalent expression \( r_j - a = 0 \) holds. More specifically, \( P_1 \) generates a pair of DPF keys for the point function \( f_{r,1} : \mathbb{Z}_{2^t} \rightarrow \mathbb{Z}_{2^t} \), where \( r \leftarrow \mathbb{Z}_{2^t} \) is randomly picked. \( P_1 \) then distributes the DPF keys to \( P_2, P_4 \). Parties jointly compute and open \( \delta := (r_j) - (a) + r \mod (2^t) \) to \( P_2 \) and \( P_4 \). After that, \( P_2 \) and \( P_4 \) compute \( \langle \text{flag}_{\text{cmpa}} \rangle := \langle f_{r,1} (\delta) \rangle \), and send their shares to \( P_1, P_3 \) respectively to construct \( \langle \text{flag}_{\text{cmpa}} \rangle \).

**cmpae:** This operation checks if \( r_j \) is above or equal to \( a \). It follows that \( \langle \text{flag}_{\text{cmpae}} \rangle := 1 - \langle \text{flag}_{\text{cmps}} \rangle (\mod 2^t) \).

**cmpg:** This operation checks if \( r_j \) is above to \( a \), whose comparison result is directly derived from the results of cmpa and cmpae by \( \langle \text{flag}_{\text{cmpg}} \rangle := \langle \text{flag}_{\text{cmpa}} \rangle - \langle \text{flag}_{\text{cmps}} \rangle (\mod 2^t) \).

**cmpge:** This operation checks if \( r_j \) is greater than \( a \) or equal to \( a \). We combine the sign bits of \( r_j, a \) and \( y := r_j - a \mod (2^t) \) to compute flag_{cmpge}. More specifically, in the same way as flag_{smulh} computation, the MPC players obliviously detect whether \( r_j, a, y \in [0, 2^t-1] \) by DICF scheme to obtain their sign bits \( \langle s_r \rangle, \langle s_a \rangle \) and \( \langle s_y \rangle \) over the ring \( \mathbb{Z}_{2^t} \). After that, four parties jointly compute
\[
\langle \text{flag}_{\text{cmpge}} \rangle := (1 - \langle s_a \rangle) \cdot (1 - \langle s_y \rangle - \langle s_a \rangle + 2 \cdot \langle s_r \rangle) \cdot \langle s_a \rangle \\
+ \langle s_a \rangle \cdot (\langle s_r \rangle + \langle s_a \rangle - 2 \cdot \langle s_r \rangle \cdot \langle s_a \rangle) (\mod 2^t)
\]
and then convert \( \langle \text{flag}_{\text{cmpge}} \rangle \) to \( \langle \text{flag}_{\text{cmpg}} \rangle \).

A.4 Move Operations

These are standard move operation mov and the conditional move operation cmov. Since both mov and cmov do not modify the condition flag, we set \( \langle \text{res}_{\text{mov}} \rangle := \langle \text{flag}_{\text{cmov}} \rangle := \langle \text{flag} \rangle \).

**mov:** This operation stores \( \langle a \rangle \) to \( \langle \text{res}_{\text{mov}} \rangle \). We only need to convert \( \langle a \rangle \) into \langle a \rangle \) and then copy it to \( \langle \text{res}_{\text{mov}} \rangle \).
**cmov:** If \( \text{flag} = 1 \), this operation stores \( \langle a \rangle \) to \( \langle \text{res}_{\text{cmov}} \rangle_{\text{rep}} \); otherwise, set \( \langle \text{res}_{\text{cmov}} \rangle_{\text{rep}} := \langle \text{ri} \rangle_{\text{rep}} \). The MPC players first jointly compute \( \langle \text{res}_{\text{cmov}} \rangle := \langle \text{flag} \rangle \cdot \langle a \rangle + \langle 1 - \langle \text{flag} \rangle \rangle \cdot \langle \text{ri} \rangle \pmod{2^t} \) and then convert \( \langle \text{res}_{\text{cmov}} \rangle \) into \( \langle \text{res}_{\text{cmov}} \rangle_{\text{rep}} \).

### A.5 Memory Operations

We support \text{load}, \text{store} to randomly access the memory, and \text{read} to access input tapes. Note that our memory operations only have register and immediate addressing modes.

**load:** This operation loads the \( a \)-th word in the memory into \( \langle \text{res}_{\text{load}} \rangle_{\text{rep}} \), and it does not modify the condition flag. To obliviously fetch \( M_a \) from the memory, we utilize our o-read protocol (cf. Fig. 3). More specifically, it works as follows.

\[ \langle \text{res}_{\text{load}} \rangle \text{.} \]

The MPC players invoke \( \Pi_{\text{read}}^{(2^t), t} \) to get \( \langle M_a \rangle \), and then convert \( \langle M_a \rangle \) to replicated secret sharing as \( \langle \text{res}_{\text{load}} \rangle_{\text{rep}} \).

**flagload:** Since this operation do not modify the condition flag, four MPC players directly set \( \langle \text{flag}_{\text{load}} \rangle_{\text{rep}} := \langle \text{flag} \rangle_{\text{rep}} \).

**store:** This operation stores \( \langle \text{ri} \rangle \) into the memory that is aligned to the \( a \)-th word. We first obliviously fetch \( \langle M_a \rangle \) and compute the new value \( y \) based on the operator type of current instruction. In practice, the MPC players copy the shares of \( \langle M_a \rangle \) obtained in \( \langle \text{res}_{\text{load}} \rangle \) computation, and utilize \( \lceil f(\delta_i, \psi_i) \rceil \) generated in the instruction fetching phase to compute

\[ \langle y \rangle := \langle f(\delta_i, \psi_i) \rangle \cdot \langle \text{read} \rangle \cdot \langle \text{ri} \rangle + \langle 1 - f(\delta_i, \psi_i) \rangle \cdot \langle M_a \rangle \pmod{2^t} \].

Namely, \( y \) is set to \( M_a \) if \( \text{op} = \text{store} \) and to \( \text{ri} \) otherwise. After that, we update the \( a \)-th word in the memory to the value of \( \langle y \rangle \) by our o-write protocol (cf. Fig. 5).

\[ \langle \text{res}_{\text{store}} \rangle \text{,} \langle \text{flag}_{\text{store}} \rangle \text{.} \]

This operation does not modify any register or the condition flag so that we set \( \langle \text{res}_{\text{store}} \rangle_{\text{rep}} := \langle \text{ri} \rangle_{\text{rep}} \) and \( \langle \text{flag}_{\text{store}} \rangle_{\text{rep}} := \langle \text{flag} \rangle_{\text{rep}} \).

**read:** This operation reads the next word (i.e. \( \langle \text{rc}_a \rangle \)-th word) on the \( a \)-th tape, then stores it into \( \langle \text{res}_{\text{read}} \rangle \) and sets \( \langle \text{flag} \rangle := 0 \); if the \( a \)-th tape does not have remaining words, it stores 0 into \( \langle \text{res}_{\text{read}} \rangle \) and sets \( \langle \text{flag} \rangle := 1 \). We use our o-read protocol to obliviously fetch the next word in \( \text{tape}_0 \) and \( \text{tape}_1 \). After that, we select the right word according to \( a \). Since the \( \text{num}_a \)-th word of \( \text{tape}_a \) is 0, we keep \( \text{rc}_a = \text{num}_a \) when the input words are consumed to ensure the fetched word is 0. To check whether each tape has any remaining input words, we let four parties hold two \( (4, 4) \)-additively shared words \( t_0, t_1 \). Initialize them to 1 if \( \text{tape}_0, \text{tape}_1 \) have input words and to 0 otherwise. Once the input words in \( \text{tape}_0 \) (or \( \text{tape}_1 \)) are consumed, \( t_0 \) (or \( t_1 \)) will set to 0. And then \( t_a \) is the value of \( \langle \text{flag}_{\text{read}} \rangle \). More specifically, it works as follows.

\[ \langle \text{res}_{\text{read}} \rangle \text{.} \]

First of all, the MPC players invoke o-read protocol twice to obliviously fetch \( \langle \text{rc}_0 \rangle \)-th word \( \langle x_0 \rangle \) in \( \text{tape}_0 \) and \( \langle \text{rc}_1 \rangle \)-th word \( \langle x_1 \rangle \) in \( \text{tape}_1 \). Meanwhile, \( P_1 \) generates two pairs of DPF keys for the point functions \( f_{r_0,1}(x), f_{r_1,1}(x) : \mathbb{Z}_{2^t} \rightarrow \mathbb{Z}_{2^t} \), where \( r_0, r_1 \leftarrow \mathbb{Z}_{2^t} \) are randomly picked, and distributes these keys to \( P_3 \) and \( P_4 \). The MPC players open \( \langle \delta_0 \rangle := \langle \text{rc}_0 \rangle - \langle \text{num}_0 \rangle - 1 + r_0 \pmod{2^t} \) and \( \langle \delta_1 \rangle := \langle \text{rc}_1 \rangle - \langle \text{num}_1 \rangle - 1 + r_1 \pmod{2^t} \) to \( P_3 \) and \( P_4 \). After that, parties can jointly compute

\[ \langle k \rangle := \lceil f(\delta_i, \psi_i) \rceil \cdot \langle \text{read} \rangle \cdot \langle a \rangle \pmod{2^t} ,
\]

\[ \langle t_0' \rangle := \langle t_0 \rangle - \lceil f(\delta_i, \psi_i) \rceil \cdot \langle a \rangle \pmod{2^t} ,
\]

\[ \langle t_1' \rangle := \langle t_1 \rangle - \lceil f(\delta_i, \psi_i) \rceil \cdot \langle a \rangle \pmod{2^t} \]

by local DPF key evaluation. If \( \text{op} = \text{read} \), \( a \) is set to 0 or 1 and \( k \) takes the value of \( a \); if \( \text{op} \neq \text{read} \), \( k \) takes the value of 0. And \( t_0' \) is set to 1 if \( \text{tape}_0 \) has remaining words (i.e. \( \text{rc}_0 < \text{num}_0 \)) and set to 0 otherwise; similarly for \( t_1' \). Parties then calculate the operation result and increases in read counters as

\[ \langle \text{res}_{\text{read}} \rangle := \langle 1 - \langle k \rangle \rangle \cdot \langle x_0 \rangle + \langle k \rangle \cdot \langle x_1 \rangle \pmod{2^t} ,
\]

\[ \langle \delta_0 \rangle := \langle 1 - \langle k \rangle \rangle \cdot \lceil f(\delta_i, \psi_i) \rceil \cdot \langle \text{read} \rangle \pmod{2^t} ,
\]

\[ \langle \delta_1 \rangle := \langle k \rangle \cdot \lceil f(\delta_i, \psi_i) \rceil \cdot \langle \text{read} \rangle \pmod{2^t} \]

Finally, parties convert \( \langle \text{res}_{\text{read}} \rangle \) into \( \langle \text{res}_{\text{read}} \rangle_{\text{rep}} \).
flag\textsubscript{read}. With the intermediate result in res\textsubscript{read} computation, parties jointly compute \( \langle \text{flag}_{\text{read}} \rangle := (1 - \langle k \rangle) \cdot \langle t_0 \rangle + \langle k \rangle \cdot \langle t_1 \rangle \pmod{2^k} \), and then convert \( \langle \text{flag}_{\text{read}} \rangle \) into \( \langle \text{flag}_{\text{read}} \rangle_{\text{rep}} \). In addition, to obliviously update \( t_0, t_1 \) and the read counters accordingly, parties need to compute
\[
\langle t_0 \rangle := (1 - \langle f(\delta_1 - \psi_1),1 (\text{read}) \rangle) \langle t_0 \rangle + \langle f(\delta_1 - \psi_1),1 (\text{read}) \rangle \langle t_1 \rangle,
\]
\[
\langle t_1 \rangle := (1 - \langle f(\delta_1 - \psi_1),1 (\text{read}) \rangle) \langle t_1 \rangle + \langle f(\delta_1 - \psi_1),1 (\text{read}) \rangle \langle t_0 \rangle,
\]
\[
\langle r_c \rangle := \langle r_c \rangle + \langle \delta_0 \rangle (\text{flag}_{\text{read}}), \quad \langle r_c \rangle := \langle r_c \rangle + \langle \delta_1 \rangle (\text{flag}_{\text{read}})
\]
over the ring \( \mathbb{Z}_{2^t} \). Namely, in the case of \( \bar{\phi} = \text{read} \), \( r_c \) will be increased if \( r_c < \text{num}_{\text{a}} \) and keep \( r_{c_a} = \text{num}_{\text{a}} \) otherwise.

A.6 Jump Operations

These jump operations may (conditional) correct the incremented \( \text{pc} \) described in Sec. 4.1 of our main text, and they do not modify any register or the condition flag, i.e., \( \langle \text{res}_{\phi} \rangle_{\text{rep}} := \langle \text{ri} \rangle_{\text{rep}} \) and \( \langle \text{flag}_{\phi} \rangle_{\text{rep}} := \langle \text{flag} \rangle_{\text{rep}} \) for \( \phi \in \{\text{jmp, cjmp, cnjmp}\} \).

\textbf{jmp}: This instruction is a standard jump operation that stores \( \langle a \rangle \) in \( \langle \text{pc} \rangle \). The MPC players only need to compute
\[
\langle \text{pc} \rangle := \langle f(\delta_1 - \psi_1),1 (\text{jmp}) \rangle : \langle a \rangle + (1 - \langle f(\delta_1 - \psi_1),1 (\text{jmp}) \rangle) : \langle \text{pc} \rangle
\]
over the ring \( \mathbb{Z}_{2^t} \).

\textbf{cjmp}: This instruction stores \( \langle a \rangle \) in \( \langle \text{pc} \rangle \) if flag = 1. The MPC players should jointly compute
\[
\langle \text{pc} \rangle := \langle f(\delta_1 - \psi_1),1 (\text{cjmp}) \rangle : \langle \langle a \rangle : \langle \text{flag} \rangle + \langle \text{pc} \rangle \cdot (1 - \langle \text{flag} \rangle) \rangle + (1 - \langle f(\delta_1 - \psi_1),1 (\text{cjmp}) \rangle) : \langle \text{pc} \rangle \pmod{2^t}.
\]

\textbf{cnjmp}: This operation stores \( \langle a \rangle \) in \( \langle \text{pc} \rangle \) if flag = 0. Similar to \text{cjmp}, it requires four parties to jointly compute
\[
\langle \text{pc} \rangle := \langle f(\delta_1 - \psi_1),1 (\text{cnjmp}) \rangle : \langle \langle a \rangle : (1 - \langle \text{flag} \rangle) + \langle \text{pc} \rangle \cdot (\langle \text{flag} \rangle) \rangle + (1 - \langle f(\delta_1 - \psi_1),1 (\text{cnjmp}) \rangle) : \langle \text{pc} \rangle \pmod{2^t}.
\]

A.7 Answer Operation

\textbf{ans}: This instruction leads to our RISC stall or stop, and does not require the following result writing phase. More specifically, \( P_1 \) and \( P_2 \) open \( f(\delta_1 - \psi_1),1 (\text{ans}) \) between each other. If \( f(\delta_1 - \psi_1),1 (\text{ans}) = 1 \), \( P_1 \) and \( P_2 \) stop computation then return \( a^{(1)} \) and \( a^{(2)} \); similarly for \( P_3 \) and \( P_4 \).

B Proof of Theorem 1

\textit{Proof.} To prove Thm. 1, we construct a PPT simulator \( S \) such that no non-uniform PPT environment \( Z \) can distinguish between (i) the real execution \( \text{Exec}_{\Pi_{\text{ram}}^{\lambda}, A, Z} \) where the parties \( P := \{P_1, \ldots, P_4\} \) run protocols \( \Pi_{\text{init}}^{\lambda}, \Pi_{\text{read}}^{\lambda} \) and \( \Pi_{\text{write}}^{\lambda} \) in the real world and the corrupted parties are controlled by a dummy adversary \( A \) who simply forwards messages from/to \( Z \), and (ii) the ideal execution \( \text{Exec}_{\mathcal{F}_{\text{ram}}^{\lambda}, S, Z} \) where the parties \( P_1, \ldots, P_4 \) interact with functionality \( \mathcal{F}_{\text{ram}}^{\lambda} \) in the ideal world, and corrupted parties are controlled by the simulator \( S \). We consider following cases.

\textbf{Case 1:} \( P_1 \) (or \( P_2 \)) is corrupted.

\textbf{Simulator.} The simulator \( S \) internally runs \( A \), forwarding messages to/from the environment \( Z \). \( S \) simulates the interface of \( \mathcal{F}_{\text{dGen}}^{\lambda}[\text{DPF.Gen}] \) as well as honest parties \( P_2, P_3, P_4 \). In addition, \( S \) simulates the following interactions with \( A \).

\begin{itemize}
  \item Upon initialization, \( S \) acts as the honest party \( P_3 \) to do:
    \begin{itemize}
      \item Set \( \varphi_1, \varphi_1^{(2)} :\rightarrow \mathbb{Z}_n, \quad \varphi_1^{(1)} := \varphi_1 - \varphi_1^{(2)}, \quad r_1^{(2)} :\rightarrow \mathbb{Z}_n; \)
      \item Set \( \kappa_{\varphi_1}^{(1)}, \kappa_{\varphi_1}^{(2)} :\rightarrow \text{DPF.Gen}(1^\lambda, f_{\varphi_1,1}); \)
      \item Send \( (\text{sid}, \kappa_{\varphi_1}^{(1)}) \) to \( P_1 \), \( (\text{sid}, \kappa_{\varphi_1}^{(2)}) \) to \( P_2 \); \)
      \item Send \( (\text{KEYGEN}, \text{sid}, r_1^{(2)}, 1) \) to \( \mathcal{F}_{\text{dGen}}^{\lambda}[\text{DPF.Gen}]; \)
    \end{itemize}
\end{itemize}
We assume that the parties distinguishability is proven through a series of hybrid worlds $H$. Upon receiving $(\text{ORAND, sid, ssid, } P_j)$ for an honest party $P_j$, $j \in \{1, 2\}$ from the external $F^{n,\ell}_{\text{ram}}$, the simulator $S$ does:
- Set $r^{(1)}_1 := w_{1 \cdot 1}$ and $r^{(2)}_1 := 0$;
- Send $(\text{sid, ssid, } \delta^{(1)}(j))$ to $P_{3-j}$, $(\text{sid, ssid, } \delta^{(2)}(j))$ to $P_3$ on behalf of $P_j$;
- Upon receiving $(\text{ORWRITE, sid, ssid, } P_j)$ for an honest party $P_j$, $j \in \{1, 3\}$ from the external $F^{n,\ell}_{\text{ram}}$, the simulator $S$ does:
  - Extract $i^{(1)} := \delta^{(1)}(j) - \text{PRF}_{\text{ram}}(\text{sid, } i^{(1)}) \pmod{n}$;
  - Send $(\text{READ, sid, } i^{(1)})$ to the external $F^{n,\ell}_{\text{ram}}$;
  - Compute $\delta^{(1)}(j) := \delta^{(1)}(j) \cdot \delta^{(2)}(j) + \delta^{(3)}(j) \cdot \delta^{(4)}(j) \pmod{n}$;
  - Set $(\text{READ, sid, } i^{(1)})$ to $P_1$ and $P_2$, $(\text{sid, ssid, } \delta^{(3)}(j))$ to $P_{7-j}$ on behalf of $P_j$;
- Upon receiving $(\text{sid, ssid, } \delta^{(1)}(j))$ from the corrupted $P_1$ to $P_2$ and $(\text{sid, ssid, } \delta^{(2)}(j))$ from the corrupted $P_1$ to $P_3, P_4$, the simulator $S$ does:
  - Set $\delta^{(1)}(j) := (j - 2) \cdot w_{1 \cdot j} + w_{1 \cdot j+1}$;
  - Set $\delta^{(2)}(j) := (j - 2) \cdot w_{1 \cdot j+1} + w_{2 \cdot j+1}$;
  - Set $\Delta^{(1)}(j) := (j - 2) \cdot w_{2 \cdot j} + w_{3 \cdot j+1}$;
- Send $(\delta^{(1)}(j), \delta^{(2)}(j), \Delta^{(1)}(j))$ to $P_q, q \in \{4/\} \pmod{n}$ on behalf of $P_j$.

Indistinguishability. We assume that the parties $P_1, ..., P_4$ communicate with each other via the secure channel functionality $F_{\text{sc}}$ (omitted in the protocol description for simplicity). The indistinguishability is proven through a series of hybrid worlds $H_0, ..., H_3$.

Hybrid $H_0$: It is the real protocol execution $\text{Exec}_{\text{PQ}, \text{A}, \text{Z}}$.

Hybrid $H_1$: $H_1$ is the same as $H_0$ except that in $H_1$, $\{w_{1 \cdot q}\}_{q \in \{2\}}$, $\varphi^{(2)}_r$ and $\{w_{1 \cdot q}\}_{q \in \{4\}}$ are picked uniformly random from $\mathbb{Z}_m$ instead of calculating from $\text{PRF}_{\text{ram}}$; $\{w_{1 \cdot q}\}_{q \in \{4\}}$ is picked uniformly random from $\mathbb{Z}_m$, instead of calculating from $\text{PRF}_{\text{ram}}$.

Claim 1. If $\text{PRF}_{\text{ram}} : \{0, 1\}^\lambda \times \{0, 1\}^n \rightarrow \mathbb{Z}_m$ is a secure pseudorandom function with adversarial advantage $\text{Adv}_{\text{PRF}_{\text{ram}}}(1^\lambda, \mathcal{A})$ and $\text{PRF}_{\text{ram}} : \{0, 1\}^\lambda \times \{0, 1\}^n \rightarrow \mathbb{Z}_m$ is a secure pseudorandom function.
with adversarial advantage $\text{Adv}_{\text{PRF}_Z^{\cdot,\cdot}}(1^\lambda, A)$, then $\mathcal{H}_1$ and $\mathcal{H}_0$ are indistinguishable with advantage $\epsilon_1 := 11 \cdot \text{Adv}_{\text{PRF}_Z^{\cdot,\cdot}}(1^\lambda, A) + 4 \cdot \text{Adv}_{\text{PRF}_Z^{\cdot,\cdot}}(1^\lambda, A)$.

Proof. We have changed 11 PRF outputs and 4 PRF outputs to uniformly random strings; therefore, the overall advantage is $11 \cdot \text{Adv}_{\text{PRF}_Z^{\cdot,\cdot}}(1^\lambda, A) + 4 \cdot \text{Adv}_{\text{PRF}_Z^{\cdot,\cdot}}(1^\lambda, A)$ by hybrid argument via reduction.

**Hybrid $\mathcal{H}_2$:** $\mathcal{H}_2$ is the same as $\mathcal{H}_1$ except that in $\mathcal{H}_2$:

- Upon receiving $(\text{OREAD, sid, ssid, } P_j)$ for $j \in \{1, 2\}$:
  - Set $\delta_{r,1}^{(j)} := w_i^{(j)}$ and $\delta_{r,2}^{(j)} := 0$.
- Upon receiving $(\text{OREAD, sid, ssid, } P_j)$ for $j \in \{3, 4\}$:
  - Set $\delta_{r,1}^{(j)} := -\varphi_1^{(j-2)} - w_i^{(j)}$ and $\delta_{r,2}^{(j)} := 0$.
- Upon receiving $(\text{OWRITE, sid, ssid, } P_j)$ for $j \in \{1, 3\}$:
  - Set $\delta_{r,1}^{(j)} := -r_1^{(j/2)} + (j-2) \cdot w_{i,1} + w_{i,j+1}^1$.
  - Set $\delta_{r,2}^{(j)} := (j-2) \cdot w_{2,1}^2 + w_{2,j+1}^2$.
- Upon receiving $(\text{OWRITE, sid, ssid, } P_j)$ for $j \in \{2, 4\}$:
  - Set $\delta_{r,1}^{(j)} := (j-3) \cdot w_{i,3} + w_{i,j}^i, \delta_{r,2}^{(j)} := (j-3) \cdot w_{2,3}^i - w_{2,j}^i$.

Instead of

- Upon receiving $(\text{OREAD, sid, ssid, } P_j)$ for $j \in \{1, 2\}$:
  - Set $\delta_{r,1}^{(j)} := i(i) + w_{i,1}$ and $\delta_{r,2}^{(j)} := i(i) - \varphi_2 + w_{2,j}$.
- Upon receiving $(\text{OREAD, sid, ssid, } P_j)$ for $j \in \{3, 4\}$:
  - Set $\delta_{r,1}^{(j)} := i(i) - \varphi_1^{(j-2)} - w_{i,j-2}$ and $\delta_{r,2}^{(j)} := i(i) - w_{2,j-2}$.
- Upon receiving $(\text{OWRITE, sid, ssid, } P_j)$ for $j \in \{1, 3\}$:
  - Set $\delta_{r,1}^{(j)} := i(i) - r_1^{(j/2)} + (j-2) \cdot w_{i,1} + w_{i,j+1}$.
  - Set $\delta_{r,2}^{(j)} := i(i) + (j-2) \cdot w_{2,1}^2 + w_{2,j+1}$.
- Upon receiving $(\text{OWRITE, sid, ssid, } P_j)$ for $j \in \{2, 4\}$:
  - Set $\delta_{r,1}^{(j)} := i(i) + (j-3) \cdot w_{i,3} - w_{i,j}$.
  - Set $\delta_{r,2}^{(j)} := i(i) - r_2^{(j/2)} + (j-3) \cdot w_{2,3}^i - w_{2,j}^i$.

**Claim 2.** If $\text{DPF}_{Z_n, Z^{\cdot,\cdot}} := (\text{Gen, Eval})$ is a secure function secret sharing scheme for point function $f_{\alpha, \beta}(x) : Z_n \rightarrow Z^{\cdot,\cdot}$ with adversarial advantage $\text{Adv}_{\text{DPF}_{Z_n, Z^{\cdot,\cdot}}}(1^\lambda, A)$, then $\mathcal{H}_2$ and $\mathcal{H}_1$ are indistinguishable with advantage $\epsilon_2 := 3 \cdot \text{Adv}_{\text{DPF}_{Z_n, Z^{\cdot,\cdot}}}(1^\lambda, A)$.

Proof. Note that $P_1$ only sees $\{\delta_{r,1}^{(j)}, \delta_{r,2}^{(j)} : j \in [4]\}$; therefore, the modification of $\{\delta_{r,1}^{(j)}, \delta_{r,2}^{(j)} : j \in [4]\}$ is oblivious to the corrupted party $P_1$. In the hybrid $\mathcal{H}_1$, we have

- $\delta_{r,1}^{(1)} := i(i) + w_{1,1}, \delta_{r,1}^{(2)} := i(i) + w_{1,2}$.
- $\delta_{r,1}^{(3)} := i(i) - \varphi_1 - w_{1,1}, \delta_{r,1}^{(4)} := i(i) - \varphi_2 - w_{1,2}$.
- $\delta_{r,1}^{(5)} := i(i) - r_1^{(j/2)} + w_{i,j}^i + w_{i,1} + w_{i,2}$.
- $\delta_{r,1}^{(6)} := i(i) - r_2^{(j/2)} + w_{i,j}^i + w_{i,1} + w_{i,2}$.
- $\delta_{r,1}^{(7)} := i(i) - r_1^{(j/2)} + w_{i,j}^i + w_{i,1} + w_{i,2}$.
- $\delta_{r,1}^{(8)} := i(i) - r_2^{(j/2)} + w_{i,j}^i + w_{i,1} + w_{i,2}$.
- $\delta_{r,1}^{(9)} := i(i) - r_1^{(j/2)} + w_{i,j}^i + w_{i,1} + w_{i,2}$.

It is straightforward that the distribution of $\{\delta_{r,1}^{(j)} : j \in [4]\}$ is uniformly random under the condition $\delta_{r,1} := \sum_{k=1}^{3} i(i) = i(i) - \varphi_1$ used to generate the DPF keys $(K_1^{(1)}, K_2^{(2)})$; similarly for $\{\delta_{r,1}^{(j)} : j \in [4]\}$ and $\{\delta_{r,2}^{(j)} : j \in [4]\}$. Whereas $\delta_{r,1} := -\varphi_1, \delta_{w,1} := -r_1$ and $\delta_{w,2} := -r_2$ in the hybrid $\mathcal{H}_2$, we can show that if there exists an adversary $A$ who can distinguish the view of $\mathcal{H}_2$ from the view of $\mathcal{H}_1$ then we can construct an adversary $B$ who uses $A$ in a blackbox fashion can break at least one of the three $\text{DPF}_{Z_n, Z^{\cdot,\cdot}} := (\text{Gen, Eval})$ with the same advantage. Therefore, $\mathcal{H}_2$ and $\mathcal{H}_1$ are indistinguishable.

**Hybrid $\mathcal{H}_3$:** $\mathcal{H}_3$ is the same as $\mathcal{H}_2$ except that in $\mathcal{H}_2$:

- Upon receiving $(\text{OWRITE, sid, ssid, } P_j)$ for $j \in \{1, 3\}$:
  - Set $\Delta_{r,1}^{(j)} := (j-2) \cdot w_{i,j+1}^i + w_{i,j}^i$.
• Upon receiving (OWRITE, sid, ssid, P_j) for j \in \{2, 4\}:
  - Set \Delta v^{(j)} := -v^{(j/2)} + (j - 3) \cdot w_{3,3}^{'} - w_{3,j}^{'};
  instead of
• Upon receiving (OWRITE, sid, ssid, P_j) for j \in \{1, 3\}:
  - Set \Delta v^{(j)} := y^{(j)}_i - x^{(j)} + (j - 2) \cdot w_{3,1}^{'} + w_{3,j+1};
• Upon receiving (OWRITE, sid, ssid, P_j) for j \in \{2, 4\}:
  - Set \Delta v^{(j)} := y^{(j)}_i - x^{(j)} - v^{(j/2)} + (j - 3) \cdot w_{3,3}^{'} - w_{3,j}^{'};

Claim 3. \mathcal{H}_3 and \mathcal{H}_2 are perfectly indistinguishable.

Proof. Since \{w_{3,j}^{'}\}_{j \in [4]} \cup \{y^{(j)}_i\}_{j \in [4]} \cup \{x^{(j)}\}_{j \in [4]} are uniformly random in \mathbb{Z}_2^{\tau}, the distribution of \\{\Delta v^{(j)}\}_{j \in [4]} \cup \{y^{(j)}_i - x^{(j)}\}_{j \in [4]} are identical. Therefore, \mathcal{H}_3 and \mathcal{H}_2 are perfectly indistinguishable. ∎

The adversary’s view of \mathcal{H}_2 is identical to the simulated view \text{Exec}_{\mathcal{F}^{n,\ell}_{\text{ram}},\mathcal{S},\mathcal{Z}}. Therefore, the overall distinguishing advantage is
\[
\text{Adv}_{\text{prf}_{\mathcal{S}}}(1^\lambda, A) + 4 \cdot \text{Adv}_{\text{prf}_{\mathcal{S}}}(1^\lambda, A) + 3 \cdot \text{Adv}_{\mathcal{F}^{n,\ell}_{\text{ram}},\mathcal{Z}}(1^\lambda, A).
\]

Case 2: P_3 (or P_4) is corrupted.

Simulator. The simulator \mathcal{S} internally runs \mathcal{A}, forwarding messages to/from the environment \mathcal{Z}. \mathcal{S} simulates the interface of \mathcal{F}^{n,\ell}_{\text{ram}}[\mathcal{DFP}.\text{Gen}] as well as honest parties P_1, P_2, P_3. In addition, \mathcal{S} simulates the following interactions with \mathcal{A}.

• Upon initialization, \mathcal{S} acts as the honest party P_1 to do:
  - Set \varphi_2, \varphi_2^{(2)} \leftarrow \mathbb{Z}_n, \varphi_2^{(1)} := \varphi_2 - \varphi_2^{(2)}, r_1^{(1)} \leftarrow \mathbb{Z}_n;
  - Set K_{\varphi_2^{(1)}}, K_{\varphi_2^{(2)}} \leftarrow \text{DFP}.\text{Gen}(1^\lambda, f_{\varphi_2,1});
  - Send (sid, K_{\varphi_2^{(1)}}) to P_3, (sid, K_{\varphi_2^{(2)}}) to P_4;
  - Send (\text{KeyGen}, sid, r_1^{(1)}, 0) to \mathcal{F}^{n,\ell}_{\text{ram}}[\text{DFP}.\text{Gen}] then get (sid, K_1^{(1)}) and send (sid, K_1^{(1)}) to P_2;

• Upon initialization, the simulator \mathcal{S} acts as the honest party P_j, j \in \{2, 4\} to do:
  - Set r_2^{(j/2)} \leftarrow \mathbb{Z}_n, v^{(j/2)} \leftarrow \mathbb{Z}_2^{\tau};
  - Send (\text{KeyGen}, sid, r_2^{(j/2)}, v^{(j/2)}) to \mathcal{F}^{n,\ell}_{\text{ram}}[\text{DFP}.\text{Gen}], then get (sid, K_v^{(j/2)}) and send (sid, K_v^{(j/2)}) to P_{j-1};

• The simulator \mathcal{S} does:
  - Pick random w_{1,1}^{}, w_{2,2}^{'} \leftarrow \mathbb{Z}_n;
  - Pick random w_{k,q}^{'} \leftarrow \mathbb{Z}_n, for k \in [2] and q \in [4];
  - Pick random w_{q}^{'} \leftarrow \mathbb{Z}_2^{\tau}, for q \in [4];

• Upon receiving (OREAD, sid, ssid, P_j) for an honest party P_j, j \in \{1, 2\} from the external \mathcal{F}^{n,\ell}_{\text{ram}}, the simulator \mathcal{S} does:
  - Set \delta_{r,1}^{(j)} := 0 and \delta_{r,2}^{(j)} := w_{2, j} - \varphi_2^{(j)};
  - Send (sid, ssid, \delta_{r,1}^{(j)}) to P_{3-j}, (sid, ssid, \delta_{r,2}^{(j)}) to P_{3} on behalf of P_{j};

• Upon receiving (OREAD, sid, ssid, P_j) for an honest party P_j, j \in \{3, 4\} from the external \mathcal{F}^{n,\ell}_{\text{ram}}, the simulator \mathcal{S} does:
  - Set \delta_{r,1}^{(j)} := 0 and \delta_{r,2}^{(j)} := -w_{2,j-2};
  - Send (sid, ssid, \delta_{r,1}^{(j)}) to P_1 and P_2, (sid, ssid, \delta_{r,2}^{(j)}) to P_{7-j} on behalf of P_{j};

• Upon receiving (sid, ssid, \delta_{r,1}^{(3)}) from the corrupted P_3 to P_4 and (sid, ssid, \delta_{r,1}^{(3)}) from the corrupted P_3 to P_1, P_2, the simulator \mathcal{S} does:
  - Extract i^{(3)} := \delta_{r,2}^{(3)} - \text{PRF}^{\mathcal{Z}_{\mathcal{S}}}_{\mathcal{S}}(\text{sid}, 2) \mod n);
  - Send (OREAD, sid, i^{(3)}) to the external \mathcal{F}^{n,\ell}_{\text{ram}};
  - Compute \delta_{r,2} := \delta_{r,1}^{(2)} + \delta_{r,2}^{(3)} + \delta_{r,2}^{(4)} \mod n);
  - Set (\beta^{(1)}_{\varphi_2,k})_{k \in \mathbb{Z}_n} \leftarrow \text{DFP}.\text{EvalAll}(1, K_{\varphi_2^{(1)}});
  - Set (\beta^{(1)}_{\varphi_2,k}) := \beta^{(1)}_{\varphi_2,k} + \delta_2 \mod n, for k \in \mathbb{Z}_n;
  - Set \zeta_1 := \text{PRF}^{\mathcal{Z}_{\mathcal{S}}}_{\mathcal{S}}(\text{sid}), \zeta_4 := \text{PRF}^{\mathcal{Z}_{\mathcal{S}}}_{\mathcal{S}}(\text{sid});
  - Compute o^{(3)} := \sum_{k=0}^{n-1} x^{(1)}_k \cdot \beta^{(1)}_{\varphi_2,k} - \zeta_1 - \zeta_4 \mod 2^{\tau};
  - Send (\text{RAND}, sid, ssid, o^{(3)}) to the external \mathcal{F}^{n,\ell}_{\text{ram}};
• Upon receiving \((\text{ORWRITE}, \text{sid}, \text{ssid}, P_j)\) for an honest party \(P_j, j \in \{1, 3\}\) from the external \(\mathcal{F}_{\text{ram}}^n\), the simulator \(S\) does:
  - Set \(\delta_{w,1}^{(1)} := -r_1^{(1/2)} + (j - 2) \cdot w_{1,1} + w_{1,j+1}';
  - Set \(\delta_{w,2}^{(1)} := (j - 2) \cdot w_{2,1} + w_{2,j+1}';
  - Set \(\Delta v^{(1)} := (j - 2) \cdot w_{3,1} + w_{3,j+1}';
  - Send (\text{sid}, \text{ssid}, \delta_{w,1}^{(1)}, \delta_{w,2}^{(1)}, \Delta v^{(1)})\) to \(P_q, q \in [4]/j\) on behalf of \(P_j\);

• Upon receiving \((\text{ORWRITE}, \text{sid}, \text{ssid}, P_j)\) for an honest party \(P_j, j \in \{2, 4\}\) from the external \(\mathcal{F}_{\text{ram}}^n\), the simulator \(S\) does:
  - Set \(\delta_{w,1}^{(j)} := (j - 3) \cdot w_{1,1} - w_{1,j}';
  - Set \(\delta_{w,2}^{(j)} := -r_2^{(j/2)} + (j - 3) \cdot w_{2,1} - w_{2,j}';
  - Set \(\Delta v^{(j)} := -r_{3/2}^{(j/2)} + (j - 3) \cdot w_{3,1} - w_{3,j}';
  - Send (\text{sid}, \text{ssid}, \delta_{w,1}^{(j)}, \delta_{w,2}^{(j)}, \Delta v^{(j)})\) to \(P_q, q \in [4]/j\) on behalf of \(P_j\);

Indistinguishability. We assume that the parties \(P_1, \ldots, P_4\) communicate with each other via the secure channel functionality \(\mathcal{F}_{\text{sc}}\) (omitted in the protocol description for simplicity). The indistinguishability is proven through a series of hybrid worlds \(H_0, \ldots, H_3\).

Hybrid \(H_0\): It is the real protocol execution \(\mathcal{E}_{\text{ram},A,z}\).

Hybrid \(H_1\): \(H_1\) is the same as \(H_0\) except that in \(H_1, \{w_{2,q}\}_{q \in [2]}, \varphi^{(2)}_{1,\ell,2}\) and \(\{w_{k,q}\}_{k \in [2],q \in [4]}\) are picked uniformly random from \(\mathbb{Z}_n\) instead of calculating from \(\text{PRF}^{\gt_2}\); \(\{w_{3,q}\}_{q \in [4]}\) is picked uniformly random from \(\mathbb{Z}_{2^2}\) instead of calculating from \(\text{PRF}^{\gt_2}\).

Claim 4. If \(\text{PRF}^{\gt_2} : \{0,1\}^\lambda \times \{0,1\}^n \rightarrow \mathbb{Z}_n\) is a secure pseudorandom function with adversarial advantage \(\text{Adv}_{\text{PRF}^{\gt_2}}(\lambda, A)\) and \(\text{PRF}^{\gt_2} : \{0,1\}^\lambda \times \{0,1\}^n \rightarrow \mathbb{Z}_{2^2}\) is a secure pseudorandom function with adversarial advantage \(\text{Adv}_{\text{PRF}^{\gt_2}}(\lambda, A)\), then \(H_1\) and \(H_0\) are indistinguishable with advantage \(c_1 := 11 \cdot \text{Adv}_{\text{PRF}^{\gt_2}}(\lambda, A) + 4 \cdot \text{Adv}_{\text{PRF}^{\gt_2}}(\lambda, A)\).

Proof. We have changed 11 \(\text{PRF}^{\gt_2}\) outputs and 4 \(\text{PRF}^{\gt_2}\) outputs to uniformly random strings; therefore, the overall advantage is \(11 \cdot \text{Adv}_{\text{PRF}^{\gt_2}}(\lambda, A) + 4 \cdot \text{Adv}_{\text{PRF}^{\gt_2}}(\lambda, A)\) by hybrid argument via reduction.

Hybrid \(H_2\): \(H_2\) is the same as \(H_1\) except that in \(H_2\):

• Upon receiving \((\text{ORREAD}, \text{sid}, \text{ssid}, P_j)\) for \(j \in \{1, 2\}\):
  - Set \(\delta_{r,1}^{(j)} := 0\) and \(\delta_{r,2}^{(j)} := -r_2^{(j/2)}\);
• Upon receiving \((\text{ORREAD}, \text{sid}, \text{ssid}, P_j)\) for \(j \in \{3, 4\}\):
  - Set \(\delta_{r,1}^{(j)} := 0\) and \(\delta_{r,2}^{(j)} := -w_{2,j-2}\);
• Upon receiving \((\text{ORWRITE}, \text{sid}, \text{ssid}, P_j)\) for \(j \in \{1, 3\}\):
  - Set \(\delta_{w,1}^{(j)} := -r_1^{(j/2)} + (j - 2) \cdot w_{1,1} + w_{1,j+1}';
  - Set \(\delta_{w,2}^{(j)} := (j - 2) \cdot w_{2,1} + w_{2,j+1}';
• Upon receiving \((\text{ORWRITE}, \text{sid}, \text{ssid}, P_j)\) for \(j \in \{2, 4\}\):
  - Set \(\delta_{w,1}^{(j)} := (j - 3) \cdot w_{1,1} - w_{1,j}';
  - Set \(\delta_{w,2}^{(j)} := (j - 3) \cdot w_{2,1} - w_{2,j}';

instead of

• Upon receiving \((\text{ORREAD}, \text{sid}, \text{ssid}, P_j)\) for \(j \in \{1, 2\}\):
  - Set \(\delta_{r,1}^{(j)} := i^{(j)} + w_{1,j}\) and \(\delta_{r,2}^{(j)} := i^{(j)} - \varphi^{(j)}_{2} + w_{2,j}\);
• Upon receiving \((\text{ORREAD}, \text{sid}, \text{ssid}, P_j)\) for \(j \in \{3, 4\}\):
  - Set \(\delta_{r,1}^{(j)} := i^{(j)} - \varphi^{(j-2)}_{1} - w_{1,j-2}\) and \(\delta_{r,2}^{(j)} := i^{(j)} - w_{2,j-2}\);
• Upon receiving \((\text{ORWRITE}, \text{sid}, \text{ssid}, P_j)\) for \(j \in \{1, 3\}\):
  - Set \(\delta_{w,1}^{(j)} := i^{(j)} - r_1^{(j/2)} + (j - 2) \cdot w_{1,1} + w_{1,j+1}';
  - Set \(\delta_{w,2}^{(j)} := i^{(j)} + (j - 2) \cdot w_{2,1} + w_{2,j+1}';
• Upon receiving \((\text{ORWRITE}, \text{sid}, \text{ssid}, P_j)\) for \(j \in \{2, 4\}\):
  - Set \(\delta_{w,1}^{(j)} := i^{(j)} + w_{1,1} - w_{1,j}';
  - Set \(\delta_{w,2}^{(j)} := i^{(j)} - r_2^{(j/2)} + (j - 3) \cdot w_{2,1} - w_{2,j}';

Claim 5. If \( \text{DPF}^{Z_n, Z_d} := (\text{Gen}, \text{Eval}) \) is a secure function secret sharing scheme for point function \( f_{\alpha, \beta}(x) : Z_n \rightarrow Z_d \) with adversarial advantage \( \text{Adv}^{\text{DPF}}_{\alpha, \beta}(1^{\lambda}, A) \), then \( \mathcal{H}_2 \) and \( \mathcal{H}_1 \) are indistinguishable with advantage \( \epsilon_2 := 3 \cdot \text{Adv}^{\text{DPF}}_{\alpha, \beta}(1^{\lambda}, A) \).

Proof. Note that \( P_1 \) only sees \( \{ \delta_r(2) \}, \{ \delta_r(1) \}_{j \in [4]} \); therefore, the modification of \( \{ \delta_r(1) \}_{j \in [4]} \) is oblivious to the corrupted party \( P_1 \). In the hybrid \( \mathcal{H}_1 \), we have

- \( \delta_r(1) := i(1) - \varphi_r(2) + w_{1,1} \), \( \delta_r(2) := i(2) - \varphi_r(2) + w_{2,2} \);
- \( \delta_r(3) := i(1) - w_{2,1} \), \( \delta_r(4) := i(1) - w_{2,2} \);
- \( \delta_r(1) := i(j) - r_{1,1} - w_{1,1}' + w_{1,2}' \), \( \delta_r(2) := i(j) - w_{1,3}' - w_{1,2}' \);
- \( \delta_r(3) := i(j) - r_{1,2}' + w_{1,1}' + w_{1,4}' + w_{2,1}' \), \( \delta_r(4) := i(j) + w_{1,3}' - w_{1,4}' \);
- \( \delta_r(1) := i(j) - w_{2,1}' + w_{2,2}' \), \( \delta_r(2) := i(j) - r_{2,1}' - w_{2,3}' - w_{2,2}' \);
- \( \delta_r(3) := i(j) + w_{2,1}' + w_{2,4}' \), \( \delta_r(4) := i(j) - r_{2,2}' + w_{2,3}' - w_{2,2}' \);

It is straightforward that the distribution of \( \{ \delta_r(2) \}_{j \in [4]} \) is uniformly random under the condition \( \sigma_r := \sum_{k=1}^{4} \delta_r(2) = i - \varphi_r \) where \( \varphi_r \) is used to generate the DPF keys \( (K^{(1)}, K^{(2)}) \); similarly for \( \{ \delta_r(1) \}_{j \in [4]} \) and \( \{ \delta_r(2) \}_{j \in [4]} \). Whereas \( \delta_r := -\varphi_r \), \( \delta_r(1) := i - r_{1} \) and \( \delta_r(2) := i - r_{2} \): in the hybrid \( \mathcal{H}_2 \), we can show that if there exists an adversary \( A \) who can distinguish the view of \( \mathcal{H}_2 \) from the view of \( \mathcal{H}_1 \), then we can construct an adversary \( B \) who uses \( A \) in a blackbox fashion can break at least one of the three \( \text{DPF}^{Z_n, Z_d} := (\text{Gen}, \text{Eval}) \) with the same advantage. Therefore, \( \mathcal{H}_2 \) and \( \mathcal{H}_1 \) are indistinguishable with adversarial advantage \( \epsilon_2 := 3 \cdot \text{Adv}^{\text{DPF}}_{\alpha, \beta}(1^{\lambda}, A) \).

Hybrid \( \mathcal{H}_3 \): \( \mathcal{H}_3 \) is the same as \( \mathcal{H}_2 \) except that in \( \mathcal{H}_2 \):
- Upon receiving \( \text{OWRITE}, \text{sid}, \text{ssid}, P_2 \) for \( j \in [1, 3] \):
  - Set \( \Delta u(j) := (j - 2) \cdot w_{3,1} + w_{3,3,1} + w_{5,1} + w_{5,3,1} + w_{5,3,1} + w_{5,3,1} \);
- Upon receiving \( \text{OWRITE}, \text{sid}, \text{ssid}, P_2 \) for \( j \in [2, 4] \):
  - Set \( \Delta u(j) := -u(j/2) + (j - 3) \cdot w_{3,3} - w_{3,3} \);

Instead of
- Upon receiving \( \text{OWRITE}, \text{sid}, \text{ssid}, P_2 \) for \( j \in [1, 3] \):
  - Set \( \Delta u(j) := y_{i,j} - x(j) + (j - 2) \cdot w_{3,1} + w_{3,3,1} + w_{5,1} + w_{5,3,1} + w_{5,3,1} + w_{5,3,1} \);
- Upon receiving \( \text{OWRITE}, \text{sid}, \text{ssid}, P_2 \) for \( j \in [2, 4] \):
  - Set \( \Delta u(j) := y_{i,j} - x(j) + (j - 2) \cdot w_{3,3} - w_{3,3} \);

Claim 6. \( \mathcal{H}_3 \) and \( \mathcal{H}_2 \) are perfectly indistinguishable.

Proof. Since \( \{ w_{3,3} \}_{j \in [4]} \), \( \{ y_{i,j} \}_{j \in [4]} \) and \( \{ x(j) \}_{j \in [4]} \) are uniformly random in \( Z_2 \), the distribution of \( \{ \Delta u(j) \}_{j \in [4]} \) and \( \{ y_{i,j} - x(j) \}_{j \in [4]} \) are identical. Therefore, \( \mathcal{H}_3 \) and \( \mathcal{H}_2 \) are perfectly indistinguishable.

The adversary’s view of \( \mathcal{H}_2 \) is identical to the simulated view \( \text{Exec}_{\text{msg}, \text{iden}, \text{sid}, Z} \). Therefore, the overall distinguishing advantage is

\[
\epsilon_1 = 11 \cdot \text{Adv}_{\text{PF}}(1^{\lambda}, A) + 4 \cdot \text{Adv}_{\text{PF}}(1^{\lambda}, A) + 3 \cdot \text{Adv}_{\text{DPF}}(1^{\lambda}, A).
\]

This concludes the proof.

References


