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; ℓ is the record size; λ, σ denote the cryptographic and statistical security parameters; d denotes the AND depth of the circuit.

	Rounds	Communication
Circuit ORAM [31]	$O(\log n)$	$O(\lambda \log^3 n + \lambda\ell \log n)$
Sqrt-ORAM [32]	$O(\log n)$	$O(\lambda\ell\sqrt{n \log^3 n})$
Floram [34]	$O(\log n)$	$O(\sqrt{\lambda\ell n \log n})$
FJKW15 [30]	$O(\log n)$	$O(\lambda\sigma \log^3 n + \ell \log n)$
JW18 [33]	$O(\log n)$	$O(\lambda \log^3 n + \ell \log n)$
BKKO20 [35]	6	$O((\lambda + \ell)\sqrt{n})$
HNO21 [36]	$O(d)$	$O((\lambda + \ell) \log n)$
Ours (offline+online)	$O(\log n) + 1$	$O(\lambda \log n) + O(\log n + \ell)$

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 $O(\sqrt{\lambda\ell n \log n})$. 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 $\mathbf{x}$ is replicated shared among the 4 MPC players, i.e., $\mathbf{x} := \mathbf{x}^{(1)} + \mathbf{x}^{(3)} = \mathbf{x}^{(2)} + \mathbf{x}^{(4)}$, such that the players P_1 and P_2 (P_3 and P_4) 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^b 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 a 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 $\mathcal{F}$ of their private inputs without revealing additional information beyond the output of $\mathcal{F}$. 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 Π UC-realizes $\mathcal{F}$ with *t-privacy*, if it satisfies that for every semi-honest PPT adversity $\mathcal{A}$ attacking an execution of Π and statically corrupting most t players, there is a simulator $\mathcal{S}$ attacking the ideal process that uses $\mathcal{F}$ (by corrupting the same set of players), such that, the executions of Π with $\mathcal{A}$ and that of $\mathcal{F}$ 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 $\langle x \rangle := \{x^{(1)}, \ldots, x^{(4)}\}$, where $x^{(1)} + x^{(2)} + x^{(3)} + x^{(4)} = x \pmod{n}$. Denote a replicated secret sharing in $\mathbb{Z}_n$ by $\langle x \rangle^{\mathsf{rep}} := \{x^{(1)}, x^{(2)}, x^{(3)}, x^{(4)}\}$, where $x^{(1)} = x^{(2)}$, $x^{(3)} = x^{(4)}$ and $x = x^{(1)} + x^{(3)} = x^{(2)} + x^{(4)} \pmod{n}$. Over the ring $\mathbb{Z}_n$, when P_1, P_2 share $[\![x]\!]$ and $P_1, \ldots, P_4$ share $\langle y \rangle$, $[\![x]\!] + \langle y \rangle$ the operation yields the result $\langle x + y \rangle$ as follows: using the additional shares of $\langle 0 \rangle$, 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 $\langle 0 \rangle$ after the initialization of seeds.

Yao's Secret Sharing. Denote a Yao's secret sharing of a bit $x \in \{0, 1\}$ by $[\![x]\!]^{\mathsf{Y}} := \{k_0, k_x\}$, 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 $[\![x]\!]^{\mathsf{Y}} := \{[\![x[t]]\!]^{\mathsf{Y}}\}_{t \in \mathbb{Z}_\ell} := \{k_{t,0}, k_{t,x[t]}\}_{t \in \mathbb{Z}_\ell}$, 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 $k_{t,1} := k_{t,0} \oplus \Delta$ for $x[t]$, where the least significant bit (LSB) of the offset Δ is set to 1. Therefore, $[\![x[t]]\!]^{\mathsf{Y}}$ 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, Δ 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 $\langle x \rangle$ into $\langle x \rangle^{\mathsf{rep}}$ 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 $\langle x \rangle$ over the ring $\mathbb{Z}_n$, the goal is to generate its equivalent Yao's secret sharing $[\![x]\!]^{\mathsf{Y}} := \{[\![x[t]]\!]^{\mathsf{Y}}\}_{t \in \mathbb{Z}_\ell}$, 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_4 agree on a random seed $\eta_2 \in \{0, 1\}^{\lambda}$. In the first round, if the offset Δ is not defined, P_1 picks a random Δ and sends it to P_2 and P_4. Next, for $q \in \{1, 3\}$, P_1 generates the labels $\{k_{t,0}^{(q)}, k_{t,1}^{(q)}\}_{t \in \mathbb{Z}_\ell}$ for the additive share $x^{(q)}$; for $q \in \{2, 4\}$, P_1 and P_q generate the labels $\{k_{t,0}^{(q)}, k_{t,1}^{(q)}\}_{t \in \mathbb{Z}_\ell}$ 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 $\{k_{t,0}\}_{t \in \mathbb{Z}_\ell}$ of

Table 2: Notations used in our distributed ORAM scheme.

$\mathbf{x}$	database, which is a linear array
x_k	k-th record in database $\mathbf{x}$
n	number of records in database $\mathbf{x}$
ℓ	bit-length of each record in database $\mathbf{x}$
$\mathcal{K}$	key of DPF scheme
β	evaluation result of DPF keys
$\tilde{\beta}$	shifted evaluation result of DPF keys
λ	security parameter
i	index of the record to be read/written
y	value to be written in o-write
δ	number of cyclic-shifting positions of the DPF full-domain evaluation results
Δv	the difference between $y - x_i$ and the random value v
w, ζ	random masks for re-randomizing shares

the value x. In the second round, P_1 sends the garbled circuit to the evaluator P_3, while each non-evaluator P_q sends its real-labels $\{k^{(q)}_{t,x^{(q)}[t]}\}_{t\in\mathbb{Z}_\ell}$ to P_3. In parallel to the two communication rounds above, P_3 runs OT functionality with P_1 to receive the real-labels $\{k^{(3)}_{t,x^{(3)}[t]}\}_{t\in\mathbb{Z}_\ell}$ of its additive share. Finally, P_3 locally evaluate the garbled circuit to obtain $\{k_{t,x[t]}\}_{t\in\mathbb{Z}_\ell}$ of the value x.

- **Y2R:** Given the Yao's secret sharing $[\![x]\!]^{\mathsf{Y}}$ of a value $x \in \mathbb{Z}_n$, the goal is to generate its equivalent $\langle x\rangle^{\mathsf{rep}}$ over the ring $\mathbb{Z}_n$. Suppose P_1 is the garbler and P_3 is the evaluator in the scheme. Let $\ell := \lceil \log_2 n \rceil$. First, P_1 picks a random $x^{(1)} \in \mathbb{Z}_n$ and generates a garbled circuit for $x^{(3)} := x - x^{(1)}$. After that, P_1 sends the garbled circuit, the real-labels $\{k^{(1)}_{t,x^{(1)}[t]}\}_{t\in\mathbb{Z}_\ell}$ of $x^{(1)}$, and the permute bits $\{k^{(3)}_{t,0}[0]\}_{t\in\mathbb{Z}_l}$ of $x^{(3)}$ to P_3. Subsequently, P_3 evaluates the garbled circuit to obtain $\{k^{(3)}_{t,x^{(3)}[t]}\}_{t\in\mathbb{Z}_\ell}$, calculates $x^{(3)}[t] := k^{(3)}_{t,0}[0] \oplus k^{(3)}_{t,x^{(3)}[t]}[0]$ for $t \in \mathbb{Z}_\ell$, and then concatenates $\{x^{(3)}[t]\}_{t\in\mathbb{Z}_\ell}$ to get the value $x^{(3)} \in \mathbb{Z}_n$. Finally, P_1 sends $x^{(1)}$ to P_2 and P_3 sends $x^{(3)}$ to P_4.

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 $\mathcal{F} = \{f(x) : \mathbb{G}^{\mathsf{in}} \to \mathbb{G}^{\mathsf{out}}\}$, 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 $\forall x \in \mathbb{G}^{in}$, $f_1(x) + f_2(x) = f(x) \pmod{|\mathbb{G}^{\mathsf{out}}|}$.

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

- $\mathsf{Gen}(1^\lambda, f_{\alpha,\beta})$ 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 $\lceil \log_2 |\mathbb{G}^{\mathsf{in}}| \rceil + 1$ correction words. Each key is able to efficiently describe the share of $f_{\alpha,\beta}$ without revealing α, β.
- $\mathsf{Eval}(b, \mathcal{K}^{(b)}, x)$ is an evaluation algorithm. For $\forall x \in \mathbb{G}^{\mathsf{in}}$, $\forall b \in [2]$, it outputs $\beta^{(b)}_x \in \mathbb{G}^{\mathsf{out}}$, such that $\beta^{(1)}_x + \beta^{(2)}_x = f_{\alpha,\beta}(x) \pmod{|\mathbb{G}^{\mathsf{out}}|}$.

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

Distributed Comparison Function (DCF) is an FSS scheme for the comparison function $f^{<}_{\alpha,\beta}(x) : \mathbb{G}^{\mathsf{in}} \to \mathbb{G}^{\mathsf{out}}$, which outputs β if $0 \le x < \alpha$ and outputs 0 if $x \ge \alpha$. 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 $f^{\mathsf{IC}}_{p,q}(x) : \mathbb{G}^{\mathsf{in}} \to \mathbb{G}^{\mathsf{out}}$, which outputs 1 if $x \in [p, q]$ and outputs 0 otherwise. DICF is an FSS scheme for the *offset* interval containment function $f^{\mathsf{IC}}_{p,q,r^{\mathsf{in}},r^{\mathsf{out}}}(x) : \mathbb{G}^{\mathsf{in}} \to \mathbb{G}^{\mathsf{out}}$ with given random offset $r^{\mathsf{in}}, r^{\mathsf{out}}$, such that $f^{\mathsf{IC}}_{p,q,r^{\mathsf{in}},r^{\mathsf{out}}}(x + r^{\mathsf{in}}) - r^{\mathsf{out}} = f^{\mathsf{IC}}_{p,q}(x)$. Similar to DPF, DICF also consists of algorithms

Functionality $\mathcal{F}_{\mathsf{ram}}^{n,\ell}$

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

- Upon receiving $(\textsc{Init}, \mathsf{sid}, \mathbf{x}^{(j)})$ from $P_j \in \mathcal{P}$:
  - Send notification $(\textsc{Init}, \mathsf{sid}, P_j)$ to $\mathcal{S}$;
  - Set $\mathcal{J} := \mathcal{J} \cup \{P_j\}$;
  - If $|\mathcal{J}| = 4$:
    - Assert $\mathbf{x}^{(1)} = \mathbf{x}^{(2)}$ and $\mathbf{x}^{(3)} = \mathbf{x}^{(4)}$;
    - Record $\mathbf{x} := \mathbf{x}^{(1)} + \mathbf{x}^{(3)}$;
- Upon receiving $(\textsc{Oread}, \mathsf{sid}, \mathsf{ssid}, i^{(j)})$ from $P_j \in \mathcal{P}$, ignore the request if $|\mathcal{J}| \neq 4$, otherwise:
  - Send notification $(\textsc{Oread}, \mathsf{sid}, \mathsf{ssid}, P_j)$ to $\mathcal{S}$;
  - Once all players have submitted their input, does:
    - Compute $i := \sum_{j=1}^{4} i^{(j)} \pmod{n}$;
    - Upon receiving $(\textsc{Rand}, \mathsf{sid}, \mathsf{ssid}, o^*)$ from $\mathcal{S}$ for the corrupted party P_k:
      - Pick random $o^{(1)}, \ldots, o^{(4)} \leftarrow \mathbb{Z}_{2^\ell}$ under the constraint $\sum_{j=1}^{4} o^{(j)} = x_i \pmod{2^\ell}$ and $o^{(k)} = o^*$;
      - For $j \in [4]$, send $(\textsc{Oread}, \mathsf{sid}, o^{(j)})$ to $P_j \in \mathcal{P}$ via private delayed channel.
- Upon receiving $(\textsc{Owrite}, \mathsf{sid}, \mathsf{ssid}, i^{(j)}, x_i^{(j)}, y^{(j)})$ from $P_j \in \mathcal{P}$, ignore the request if $|\mathcal{J}| \neq 4$, otherwise:
  - Send notification $(\textsc{Owrite}, \mathsf{sid}, \mathsf{ssid}, P_j)$ to $\mathcal{S}$;
  - Once all players have submitted their input, does:
    - Compute $i := \sum_{j=1}^{4} i^{(j)} \pmod{n}$;
    - Set $x_i := \sum_{j=1}^{4} y^{(j)} \pmod{2^\ell}$;

Figure 1: The distributed ORAM functionality $\mathcal{F}_{\mathsf{ram}}^{n,\ell}$.

$(\mathsf{Gen}^{\mathsf{IC}}, \mathsf{Eval}^{\mathsf{IC}})$ as follows:

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

In the rest of paper, we focus on the case of $r^{\mathsf{out}} = 0$ and thus omit r^{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 $(\mathsf{Gen}, \mathsf{Eval})$ is T-secure for function family $\mathcal{F} = \{f : \mathbb{G}^{\mathsf{in}} \to \mathbb{G}^{\mathsf{out}}\}$, if for all non-uniform PPT adversaries $\mathcal{A}$, it holds that

$$\mathsf{Adv}(1^\lambda, \mathcal{A}) = \left| \Pr\left[\begin{array}{l} (f_1, f_2, \phi) \leftarrow \mathcal{A}(1^\lambda); b \leftarrow \{1, 2\}; \\ (\mathcal{K}^{(1)}, \mathcal{K}^{(2)}) \leftarrow \mathsf{Gen}(1^\lambda, f_b); \\ b^* \leftarrow \mathcal{A}((\mathcal{K}^{(i)})_{i \in T}, \phi) : \\ f_1, f_2 \in \mathcal{F} \ \wedge \ b = b^* \end{array} \right] - \frac{1}{2} \right|$$

is negligible in λ.

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 $\mathcal{F}_{\mathsf{ram}}^{n,\ell}$ of our distributed ORAM is described in Fig. 1. As shown in Fig. 2, our initialization protocol $\Pi_{\mathsf{init}}^{n,\ell}$ 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 $\mathbf{x} \in (\mathbb{Z}_{2^\ell})^n$. 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

Protocol $\Pi_{\mathsf{init}}^{n,\ell}$

- With the private data $\mathbf{x} = (x_0, \ldots, x_{n-1}) \in (\mathbb{Z}_{2^\ell})^n$, the client dose:
  - For $i \in \mathbb{Z}_n$, generate $x_i^{(1)} \leftarrow \mathbb{Z}_{2^\ell}$, set $x_i^{(2)} := x_i^{(1)}$ and $x_i^{(3)} := x_i^{(4)} := x_i - x_i^{(1)} \pmod{2^\ell}$;
  - For $j \in [4]$, denote $\mathbf{x}^{(j)} := (x_0^{(j)}, \ldots, x_{n-1}^{(j)})$, and send $(\textsc{Init}, \mathsf{sid}, \mathbf{x}^{(j)})$ to the player P_j;
- Upon receiving $(\textsc{Init}, \mathsf{sid}, \mathbf{x}^{(j)})$, $P_j, j \in [4]$ stores $\mathbf{x}^{(j)}$.

Figure 2: Initialization protocol $\Pi_{\mathsf{init}}^{n,\ell}$.

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 $(\mathcal{K}^{(1)}, \mathcal{K}^{(2)})$ 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 δ to the evaluators, i.e., P_1 and P_2. The evaluators then reconstruct δ, perform full domain evaluation of DPF f_φ and cyclic-shifts the output array to the right δ 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 $\mathbf{x}$ 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 $\langle \mathbf{x} \rangle^{\mathsf{rep}}$, 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 $[\![x_i^{(1)}]\!]$ while P_3 and P_4 obtain the secret sharing of $[\![x_i^{(3)}]\!]$, 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 $(\mathcal{K}_1^{(1)}, \mathcal{K}_1^{(2)})$ 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 $(\mathcal{K}_v^{(1)}, \mathcal{K}_v^{(2)})$ for the point function $f_{r_2,v}(x)$ via MPC, where $r_2 \leftarrow \mathbb{Z}_n$ and $v \leftarrow \mathbb{Z}_{2^\ell}$ are secret. After that, P_1 and P_2 exchange $\mathcal{K}_1^{(1)}$ and $\mathcal{K}_v^{(1)}$; P_3 and P_4 exchange $\mathcal{K}_1^{(2)}$ and $\mathcal{K}_v^{(2)}$. In the online phase, $\delta_{w,1} := i - r_1 \pmod{n}$, $\delta_{w,2} := i - r_2 \pmod{n}$ and $\Delta v := y - x_i - v \pmod{2^\ell}$ 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 $\mathbf{x}$ 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 $\mathsf{DPF.Gen}$ to generate the DPF keys $(\mathcal{K}_{\varphi_1}^{(1)}, \mathcal{K}_{\varphi_1}^{(2)})$ for the point function $f_{\varphi_1,1}(x) : \mathbb{Z}_n \to \mathbb{Z}_{2^\ell}$, where $\varphi_1 \leftarrow \mathbb{Z}_n$ is randomly picked; P_1 invokes $\mathsf{DPF.Gen}$ to generate DPF keys $(\mathcal{K}_{\varphi_2}^{(1)}, \mathcal{K}_{\varphi_2}^{(2)})$ for the point function $f_{\varphi_2,1}(x) : \mathbb{Z}_n \to \mathbb{Z}_{2^\ell}$, where $\varphi_2 \leftarrow \mathbb{Z}_n$ is randomly picked. Subsequently, P_3 sends $\mathcal{K}_{\varphi_1}^{(1)}$ to P_1 and $\mathcal{K}_{\varphi_1}^{(2)}$ to P_2; P_1 sends $\mathcal{K}_{\varphi_2}^{(1)}$ to P_3 and $\mathcal{K}_{\varphi_2}^{(2)}$ to P_2.

In the online phase, the MPC players compute $\langle \delta_{r,1} \rangle := \langle i \rangle - [\![\varphi_1]\!]$ and $\langle \delta_{r,2} \rangle := \langle i \rangle - [\![\varphi_2]\!]$, 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 $\mathsf{DPF.EvalAll}$ to full domain evaluate the DPF key received in the offline phase. Next, P_j cyclic-shifts the evaluation result $\{\beta_{\varphi_1,k}^{(j)}\}_{k \in \mathbb{Z}_n}$ to the right $\delta_{r,1}$ positions, denoted as $\{\tilde{\beta}_{\varphi_1,k}^{(j)}\}_{k \in \mathbb{Z}_n}$. It is easy to see,

```
Protocol $\Pi^{n,\ell}_{\mathsf{read}}$

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 $\varphi_2 \leftarrow \mathbb{Z}_n$, set $\varphi_2^{(1)} := \varphi_2 - \mathsf{PRF}^{\mathbb{Z}_n}_{\eta_2}(\mathsf{sid}, 0)$;
    – Set $(\mathcal{K}^{(1)}_{\varphi_2}, \mathcal{K}^{(2)}_{\varphi_2}) \leftarrow \mathsf{DPF.Gen}(1^\lambda, f_{\varphi_2,1})$;
    – Send $(\mathsf{sid}, \mathcal{K}^{(1)}_{\varphi_2})$ to $P_3$, $(\mathsf{sid}, \mathcal{K}^{(2)}_{\varphi_2})$ to $P_4$;
  • Upon initialization, $P_3$ does:
    – Generate $\varphi_1 \leftarrow \mathbb{Z}_n$, set $\varphi_1^{(1)} := \varphi_1 - \mathsf{PRF}^{\mathbb{Z}_n}_{\eta_4}(\mathsf{sid}, 0)$;
    – Set $(\mathcal{K}^{(1)}_{\varphi_1}, \mathcal{K}^{(2)}_{\varphi_1}) \leftarrow \mathsf{DPF.Gen}(1^\lambda, f_{\varphi_1,1})$;
    – Send $(\mathsf{sid}, \mathcal{K}^{(1)}_{\varphi_1})$ to $P_1$, $(\mathsf{sid}, \mathcal{K}^{(2)}_{\varphi_1})$ to $P_2$;
  • Upon initialization, $P_2$ sets $\varphi_2^{(2)} := \mathsf{PRF}^{\mathbb{Z}_n}_{\eta_2}(\mathsf{sid}, 0)$;
  • Upon initialization, $P_4$ sets $\varphi_1^{(2)} := \mathsf{PRF}^{\mathbb{Z}_n}_{\eta_4}(\mathsf{sid}, 0)$;
Online phase:
  • With the private input $(\textsc{Oread}, \mathsf{sid}, \mathsf{ssid}, i^{(j)})$, the player $P_j, j \in \{1,2\}$ does:
    – Set $w_{1,j} \leftarrow \mathsf{PRF}^{\mathbb{Z}_n}_{\eta_{2j-1}}(\mathsf{sid}, 1)$, $w_{2,j} \leftarrow \mathsf{PRF}^{\mathbb{Z}_n}_{\eta_{2j-1}}(\mathsf{sid}, 2)$;
    – Set $\delta^{(j)}_{r,1} := i^{(j)} + w_{1,j} \pmod n$;
    – Set $\delta^{(j)}_{r,2} := i^{(j)} - \varphi_2^{(j)} + w_{2,j} \pmod n$;
    – Send $(\mathsf{sid}, \mathsf{ssid}, \delta^{(j)}_{r,1})$ to $P_{3-j}$, $(\mathsf{sid}, \mathsf{ssid}, \delta^{(j)}_{r,2})$ to $P_3$ and $P_4$;
  • With the private input $(\textsc{Oread}, \mathsf{sid}, \mathsf{ssid}, i^{(j)})$, the player $P_j, j \in \{3,4\}$ does:
    – Set $w_{1,j-2} \leftarrow \mathsf{PRF}^{\mathbb{Z}_n}_{\eta_{2j-5}}(\mathsf{sid}, 1)$;
    – Set $w_{2,j-2} \leftarrow \mathsf{PRF}^{\mathbb{Z}_n}_{\eta_{2j-5}}(\mathsf{sid}, 2)$;
    – Set $\delta^{(j)}_{r,1} := i^{(j)} - \varphi_1^{(j-2)} - w_{1,j-2} \pmod n$;
    – Set $\delta^{(j)}_{r,2} := i^{(j)} - w_{2,j-2} \pmod n$;
    – Send $(\mathsf{sid}, \mathsf{ssid}, \delta^{(j)}_{r,1})$ to $P_1$ and $P_2$, $(\mathsf{sid}, \mathsf{ssid}, \delta^{(j)}_{r,2})$ to $P_{7-j}$;
  • Upon receiving $(\mathsf{sid}, \mathsf{ssid}, \delta^{(3-j)}_{r,1})$ from $P_{3-j}$, $(\mathsf{sid}, \mathsf{ssid}, \delta^{(3)}_{r,1})$ from $P_3$ , and $(\mathsf{sid}, \mathsf{ssid}, \delta^{(4)}_{r,1})$ from $P_4$,
    $P_j, j \in \{1,2\}$ does:
    – Set $\delta_{r,1} := \delta^{(1)}_{r,1} + \delta^{(2)}_{r,1} + \delta^{(3)}_{r,1} + \delta^{(4)}_{r,1} \pmod n$;
    – Set $(\beta^{(j)}_{\varphi_1,k})_{k\in\mathbb{Z}_n} \leftarrow \mathsf{DPF.EvalAll}(j, \mathcal{K}^{(j)}_{\varphi_1})$;
    – Set $\widetilde{\beta}^{(j)}_{\varphi_1,k} := \beta^{(j)}_{\varphi_1,k+\delta_{r,2} \pmod n}$, for $k \in \mathbb{Z}_n$;
    – Set $\zeta_j \leftarrow \mathsf{PRF}^{\mathbb{Z}_{2^\ell}}_{\eta_{2j-1}}(\mathsf{sid})$, $\zeta_3 \leftarrow \mathsf{PRF}^{\mathbb{Z}_{2^\ell}}_{\eta_2}(\mathsf{sid})$;
    – Return $o^{(j)} := \sum_{k=0}^{n-1}(x_k^{(j)} \cdot \widetilde{\beta}^{(j)}_{\varphi_1,k}) + \zeta_j + (-1)^j \cdot \zeta_3$.
  • Upon receiving $(\mathsf{sid}, \mathsf{ssid}, \delta^{(1)}_{r,2})$ from $P_1$, $(\mathsf{sid}, \mathsf{ssid}, \delta^{(2)}_{r,2})$ from $P_2$, and $(\mathsf{sid}, \mathsf{ssid}, \delta^{(7-j)}_{r,2})$ from $P_{7-j}$,
    $P_j, j \in \{3,4\}$ does:
    – Set $\delta_{r,2} := \delta^{(1)}_{r,2} + \delta^{(2)}_{r,2} + \delta^{(3)}_{r,2} + \delta^{(4)}_{r,2} \pmod n$;
    – Set $(\beta^{(j)}_{\varphi_2,k})_{k\in\mathbb{Z}_n} \leftarrow \mathsf{DPF.EvalAll}(j-2, \mathcal{K}^{(j-2)}_{\varphi_2})$;
    – Set $\widetilde{\beta}^{(j)}_{\varphi_2,k} := \beta^{(j)}_{\varphi_2,k+\delta_{r,2} \pmod n}$, for $k \in \mathbb{Z}_n$;
    – Set $\zeta_{j-2} \leftarrow \mathsf{PRF}^{\mathbb{Z}_{2^\ell}}_{\eta_{2j-5}}(\mathsf{sid})$, $\zeta_4 \leftarrow \mathsf{PRF}^{\mathbb{Z}_{2^\ell}}_{\eta_3}(\mathsf{sid})$;
    – Return $o^{(j)} := \sum_{k=0}^{n-1}(x_k^{(j)} \cdot \widetilde{\beta}^{(j-2)}_{\varphi_2,k}) - \zeta_{j-2} + (-1)^j \cdot \zeta_4$
```

Figure 3: O-read protocol $\Pi^{n,\ell}_{\mathsf{read}}$.

the only non-zero value is $[\![\tilde{\beta}_{\varphi_1,i}]\!] = 1$ in the shifted array. Similarly, P_3 and P_4 use their DPF keys $(\mathcal{K}^{(1)}_{\varphi_2}, \mathcal{K}^{(2)}_{\varphi_2})$ and the corresponding $\delta_{r,2}$ for cyclic-shifting to obtain $\{[\![\tilde{\beta}_{\varphi_2,k}]\!]\}_{k\in\mathbb{Z}_n}$. Consequently, we have $x_i = \sum_{j=1}^{4} \hat{o}^{(j)} \pmod{2^\ell}$ where for $j \in [4]$:

$$\hat{o}^{(j)} := \sum_{k=0}^{n-1}(x_k^{(j)} \cdot \tilde{\beta}^{(q)}_{\varphi_{\lceil j/2\rceil},k}) \pmod{2^\ell}$$

Functionality $\mathcal{F}^{n,\ell}_{\mathsf{dGen}}[\mathsf{DPF.Gen}]$

It interacts with $\mathcal{P} := \{P_1, P_2\}$ and the adversary $\mathcal{S}$. It is parameterized with the DPF key generation algorithm $\mathsf{DPF.Gen}$.

- Upon receiving $(\textsc{KeyGen}, \mathsf{sid}, r^{(j)}, v^{(j)})$ from $P_j, j \in [2]$:
  - Send notification $(\textsc{KeyGen}, \mathsf{sid}, P_j)$ to $\mathcal{S}$;
  - 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 $(\mathcal{K}^{(1)}, \mathcal{K}^{(2)}) \leftarrow \mathsf{DPF.Gen}(1^\lambda, f_{r,v})$, for the point function $f_{r,v}(x) : \mathbb{Z}_n \to \mathbb{Z}_{2^\ell}$;
    - For $j \in [2]$, send $(\textsc{KeyGen}, \mathsf{sid}, \mathcal{K}^{(j)})$ to $P_j \in \mathcal{P}$ via private delayed channel.

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

when $q := 2 + (-1)^j$. Finally, we re-randomize $\hat{o}^{(j)}$ 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 $\mathbf{x}$ 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 $\mathcal{F}_{\mathsf{dGen}}[\mathsf{DPF.Gen}]$ in o-write protocol description for distributed DPF key generation, and its actual 2PC protocol $\Pi_{\mathsf{DPF.dGen}}$ 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 $\eta_3 \in \{0,1\}^\lambda$; P_3 and P_4 agree on a random seed $\eta_4 \in \{0,1\}^\lambda$.

In the offline phase, P_1 and P_3 invoke $\mathcal{F}^{n,\ell}_{\mathsf{dGen}}[\mathsf{DPF.Gen}]$ 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 $(\mathcal{K}^{(1)}_v, \mathcal{K}^{(2)}_v)$ 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 $\mathcal{K}^{(1)}_v$ of $f_{r_2,v}(x)$ and sends it to P_1; P_4 obtains the share $\mathcal{K}^{(2)}_v$ and sends it to P_2.

In the online phase, four MPC players jointly compute and open $\langle \delta_{w,1} \rangle := \langle i \rangle - [\![r_1]\!] \pmod n$, $\langle \delta_{w,2} \rangle := \langle i \rangle - [\![r_2]\!] \pmod n$, and $\langle \Delta v \rangle := \langle y \rangle - \langle x_i \rangle - [\![v]\!] \pmod{2^\ell}$. 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 $\mathbf{x}$. More specifically, P_j full domain evaluates $\mathcal{K}^{(\lceil j/2 \rceil)}_1, \mathcal{K}^{(\lceil j/2 \rceil)}_v$ to get $\{\beta^{(\lceil j/2 \rceil)}_{1,k}\}_{k \in \mathbb{Z}_n}, \{\beta^{(\lceil j/2 \rceil)}_{v,k}\}_{k \in \mathbb{Z}_n}$. After that, P_j cyclic-shifts $\{\beta^{(\lceil j/2 \rceil)}_{1,k}\}_{k \in \mathbb{Z}_n}$ to the right $\delta_{w,1}$ positions, $\{\beta^{(\lceil j/2 \rceil)}_{v,k}\}_{k \in \mathbb{Z}_n}$ to the right $\delta_{w,2}$ positions, and denotes the shifted arrays as $\{\tilde{\beta}^{(\lceil j/2 \rceil)}_{1,k}\}_{k \in \mathbb{Z}_n}$ and $\{\tilde{\beta}^{(\lceil j/2 \rceil)}_{v,k}\}_{k \in \mathbb{Z}_n}$ respectively. P_j then updates the shares of database as

$$x^{(j)}_k := x^{(j)}_k + \Delta v \cdot \tilde{\beta}^{(\lceil j/2 \rceil)}_{1,k} + \tilde{\beta}^{(\lceil j/2 \rceil)}_{v,k} \pmod{2^\ell}, \; k \in \mathbb{Z}_n \; .$$

Note that P_1 and P_2 still have identical shares of $\mathbf{x}$ 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 $\Pi^{n,\ell}_{\mathsf{init}}$, $\Pi^{n,\ell}_{\mathsf{read}}$ and $\Pi^{n,\ell}_{\mathsf{write}}$ with the following theorem, and its proof can be found in Appendix B.

Theorem 1. *Let $\mathsf{DPF}^{\mathbb{Z}_n, \mathbb{Z}_{2^\ell}}$ be a secure function secret sharing scheme for point function $f_{\alpha,\beta}(x) : \mathbb{Z}_n \mapsto \mathbb{Z}_{2^\ell}$ with adversarial advantage $\mathsf{Adv}_{\mathsf{DPF}^{\mathbb{Z}_n, \mathbb{Z}_{2^\ell}}}(1^\lambda, \mathcal{A})$. Let $\mathsf{PRF}^{\mathbb{Z}_n} : \{0,1\}^\lambda \times \{0,1\}^{\mathsf{in}} \mapsto \mathbb{Z}_n$ be*

Protocol $\Pi^{n,\ell}_{\mathsf{write}}$

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_1 sets $r_1^{(1)} \leftarrow \mathbb{Z}_n$, P_3 sets $r_1^{(2)} \leftarrow \mathbb{Z}_n$;
- P_1 and P_3 request $\mathcal{F}^{n,\ell}_{\mathsf{dGen}}[\mathsf{DPF.Gen}]$:
  - P_1 sends $(\textsc{KeyGen}, \mathsf{sid}, r_1^{(1)}, 0)$ and gets $\mathcal{K}_1^{(1)}$;
  - P_3 sends $(\textsc{KeyGen}, \mathsf{sid}, r_1^{(2)}, 1)$ and gets $\mathcal{K}_1^{(2)}$;
- P_1 sends $(\mathsf{sid}, \mathcal{K}_1^{(1)})$ to P_2, P_3 sends $(\mathsf{sid}, \mathcal{K}_1^{(2)})$ to P_4;
- P_2 sets $r_2^{(1)} \leftarrow \mathbb{Z}_n$, $v^{(1)} \leftarrow \mathbb{Z}_{2^\ell}$;
- P_4 sets $r_2^{(2)} \leftarrow \mathbb{Z}_n$, $v^{(2)} \leftarrow \mathbb{Z}_{2^\ell}$;
- P_2 and P_4 request $\mathcal{F}^{n,\ell}_{\mathsf{dGen}}[\mathsf{DPF.Gen}]$:
  - P_2 sends $(\textsc{KeyGen}, \mathsf{sid}, r_2^{(1)}, v^{(1)})$ and gets $\mathcal{K}_v^{(1)}$;
  - P_4 sends $(\textsc{KeyGen}, \mathsf{sid}, r_2^{(2)}, v^{(2)})$ and gets $\mathcal{K}_v^{(2)}$;
- P_2 sends $(\mathsf{sid}, \mathcal{K}_v^{(1)})$ to P_1, P_4 sends $(\mathsf{sid}, \mathcal{K}_v^{(2)})$ to P_3;

Online phase:

- With the private input $(\textsc{OWrite}, \mathsf{sid}, \mathsf{ssid}, i^{(j)}, x_i^{(j)}, y^{(j)})$, the player $P_j, j \in \{1,3\}$ sets:
  - $w'_{k,q} \leftarrow \mathsf{PRF}^{\mathbb{Z}_n}_{\eta_q}(\mathsf{sid}, k)$, $k \in [2]$, $q \in \{1, j+1\}$;
  - $w'_{3,q} \leftarrow \mathsf{PRF}^{\mathbb{Z}_{2^\ell}}_{\eta_q}(\mathsf{sid}, 3)$, $q \in \{1, j+1\}$;
  - $\delta_{w,1}^{(j)} := i^{(j)} - r_1^{(\lceil j/2 \rceil)} + (j-2) \cdot w'_{1,1} + w'_{1,j+1}$;
  - $\delta_{w,2}^{(j)} := i^{(j)} + (j-2) \cdot w'_{2,1} + w'_{2,j+1}$;
  - $\Delta v^{(j)} := y_i^{(j)} - x^{(j)} + (j-2) \cdot w'_{3,1} + w'_{3,j+1}$;
- With the private input $(\textsc{OWrite}, \mathsf{sid}, \mathsf{ssid}, i^{(j)}, x_i^{(j)}, y^{(j)})$, the player $P_j, j \in \{2,4\}$ sets:
  - $w'_{k,q} \leftarrow \mathsf{PRF}^{\mathbb{Z}_n}_{\eta_q}(\mathsf{sid}, k)$, $q \in \{3, j\}$, $k \in [2]$;
  - $w'_{3,q} \leftarrow \mathsf{PRF}^{\mathbb{Z}_{2^\ell}}_{\eta_q}(\mathsf{sid}, 3)$, $q \in \{3, j\}$;
  - $\delta_{w,1}^{(j)} := i^{(j)} + (j-3) \cdot w'_{1,3} - w'_{1,j}$;
  - $\delta_{w,2}^{(j)} := i^{(j)} - r_2^{(j/2)} + (j-3) \cdot w'_{2,3} - w'_{2,j}$;
  - $\Delta v^{(j)} := y_i^{(j)} - x^{(j)} - v^{(j/2)} + (j-3) \cdot w'_{3,3} - w'_{3,j}$;
- $P_j, j \in [4]$ sends $(\mathsf{sid}, \mathsf{ssid}, \delta_{w,1}^{(j)}, \delta_{w,2}^{(j)}, \Delta v^{(j)})$ to others.
- Upon receiving all $(\mathsf{sid}, \mathsf{ssid}, \delta_{w,1}^{(k)}, \delta_{w,2}^{(k)}, \Delta v^{(k)})$ from P_k for $k \in [4]/j$, party $P_j, j \in [4]$ sets:
  - $\delta_{w,k} := \sum_{q=1}^4 \delta_{w,k}^{(q)} \pmod{n}$, for $k \in [2]$;
  - $\Delta v := \sum_{q=1}^4 \Delta v^{(q)} \pmod{2^\ell}$;
  - $(\beta_{1,k}^{(\lceil j/2 \rceil)})_{k \in \mathbb{Z}_n} \leftarrow \mathsf{DPF.EvalAll}(\lceil j/2 \rceil, \mathcal{K}_1^{(\lceil j/2 \rceil)})$;
  - $(\beta_{v,k}^{(\lceil j/2 \rceil)})_{k \in \mathbb{Z}_n} \leftarrow \mathsf{DPF.EvalAll}(\lceil j/2 \rceil, \mathcal{K}_v^{(\lceil j/2 \rceil)})$;
  - **for** $k := 0$ to $n-1$:
    - $\tilde{\beta}_{1,k}^{(\lceil j/2 \rceil)} := \beta_{1,k+\delta_{w,1} \pmod{n}}^{(\lceil j/2 \rceil)}$;
    - $\tilde{\beta}_{v,k}^{(\lceil j/2 \rceil)} := \beta_{v,k+\delta_{w,2} \pmod{n}}^{(\lceil j/2 \rceil)}$;
    - $x_k^{(j)} := x_k^{(j)} + \Delta v \cdot \tilde{\beta}_{1,k}^{(\lceil j/2 \rceil)} + \tilde{\beta}_{v,k}^{(\lceil j/2 \rceil)}$.

Figure 5: O-write protocol $\Pi^{n,\ell}_{\mathsf{write}}$.

Table 3: Our instruction set.

Opcode	Operator	Operands	Effects for registers	Flag
00000	and	$i\ j\ A$	compute bitwise AND of $\mathsf{r}j$ and a and store result in i-th register	result is 0^ℓ
00001	or	$i\ j\ A$	compute bitwise OR of $\mathsf{r}j$ and a and store result in i-th register	result is 0^ℓ
00010	xor	$i\ j\ A$	compute bitwise XOR of $\mathsf{r}j$ and a and store result in i-th register	result is 0^ℓ
00011	not	$i\ A$	compute bitwise NOT of a and store result in i-th register	result is 0^ℓ
00100	shl	$i\ j\ A$	shift $\mathsf{r}j$ by a bits to the left and store result in i-th register	MSB of $\mathsf{r}j$
00101	shr	$i\ j\ A$	shift $\mathsf{r}j$ by a bits to the right and store result in i-th register	LSB of $\mathsf{r}j$
00110	add	$i\ j\ A$	compute $\mathsf{r}j_u + a_u$ and store result in i-th register	overflow
00111	sub	$i\ j\ A$	compute $\mathsf{r}j_u - a_u$ and store result in i-th register	borrow
01000	mull	$i\ j\ A$	compute $\mathsf{r}j_u \cdot a_u$ and store the least significant bits of result in i-th register	overflow
01001	umulh	$i\ j\ A$	compute $\mathsf{r}j_u \cdot a_u$ and store the most significant bits of result in i-th register	overflow
01010	smulh	$i\ j\ A$	compute signed $\mathsf{r}j_s \cdot a_s$ and store the most significant bits of result in i-th register	over/underflow
01011	cmpe	$i\ A$	-	$\mathsf{r}i = a$
01100	cmpa	$i\ A$	-	$\mathsf{r}i_u > a_u$
01101	cmpae	$i\ A$	-	$\mathsf{r}i_u \geq a_u$
01110	cmpg	$i\ A$	-	$\mathsf{r}i_s > a_s$
01111	cmpge	$i\ A$	-	$\mathsf{r}i_s \geq a_s$
10000	mov	$i\ A$	store a in i-th register	-
10001	cmov	$i\ A$	if $\mathsf{flag} = 1$, store a in i-th register	-
10010	store	$A\ i$	store $\mathsf{r}i$ in memory that is aligned to the a-th word	-
10011	load	$i\ A$	store into i-th register the word in memory that is aligned to the a-th word	-
10100	read	$i\ A$	store the next word of a-th tape in i-th register	no remaining words in a-th tape
10101	jmp	A	store a in pc	-
10110	cjmp	A	if $\mathsf{flag} = 1$, store a in pc	-
10111	cnjmp	A	if $\mathsf{flag} = 0$, store a in pc	-
11000	ans	A	stall or halt (and return a)	-

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

$$11 \cdot \mathsf{Adv}_{\mathsf{PRF}^{\mathbb{Z}_n}}(1^\lambda, \mathcal{A}) + 4 \cdot \mathsf{Adv}_{\mathsf{PRF}^{\mathbb{Z}_{2^\ell}}}(1^\lambda, \mathcal{A}) + 3 \cdot \mathsf{Adv}_{\mathsf{DPF}^{\mathbb{Z}_n, \mathbb{Z}_{2^\ell}}}(1^\lambda, \mathcal{A})$$

4 MPC for TinyRAM

Our construction follows the RISC framework with random-access memory, whose word size is denoted by ℓ 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 $\mathsf{pc} \in \{0,1\}^\ell$.
- The condition flag, denoted as $\mathsf{flag} \in \{0,1\}^\ell$.
- m general purpose registers, each storing one word.
- Memory, which is a linear array $(M_i)_{i \in \mathbb{Z}_{2^\ell}}$ of 2^ℓ words.
- Two input tapes: tape_0 is used for the primary input, and tape_1 is used for an auxiliary input. Each tape has a read counter, denoted as $\mathsf{rc}_0, \mathsf{rc}_1 \in \{0,1\}^\ell$ and input length, denoted as $\mathsf{num}_0, \mathsf{num}_1 \in \{0,1\}^\ell$.

During a program execution, we assume that the input tapes are read-only and the num_0-th (or num_1-th) word in tape_0 (or 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 $\mathsf{r}i$ denotes the ℓ-bit string currently stored in the i-th register; A is either an immediate value or a register index, and a denotes its value (i.e., the immediate value itself or the ℓ-bit string currently stored in A-th register). To distinguish between unsigned and signed integers represented by the same ℓ-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 2ℓ-bit string, and it consists of following 6 fields:

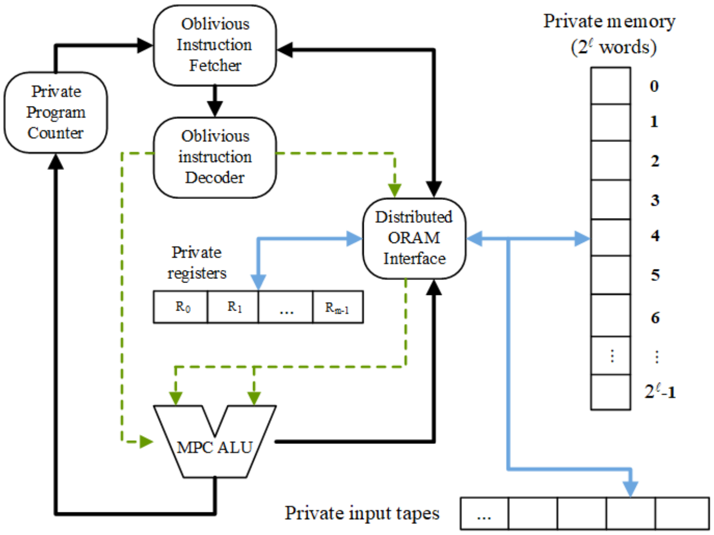


Figure 6: Oblivious RAM-MPC Architecture.

- Field 1 stores the instruction operator, which is identified by an opcode 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 ℓ 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:

$$\underbrace{01000}_{\mathsf{mull}} || \underbrace{001011}_{A \text{ is imm.}} || \underbrace{01000}_{i:=8} || \underbrace{01111}_{j:=15} || \underbrace{\overbrace{0\ldots0}^{11\text{ bits}}}_{\text{padding}} || \underbrace{\overbrace{0\ldots0}^{22\text{ bits}}1000001011}_{A:=523}$$

Field 1 Field 2 Field 3 Field 4 Field 5 Field 6

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, pc, $\{\mathsf{rc}_0, \mathsf{rc}_1\}$ and $\{\mathsf{num}_0, \mathsf{num}_1\}$ 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 $i', j' \leftarrow \mathbb{Z}_m$ 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 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 pc-th instruction and spend only 1 online round. More specifically, four MPC players begin by invoking $\Pi^{2^\ell,*}_{\mathsf{read}}$ with the secret shared program counter $\langle \mathsf{pc} \rangle$ over the private memory, [1] and then locally increase the program counter as $\langle \mathsf{pc} \rangle := \langle \mathsf{pc} \rangle + 2 \pmod{n}$.[2] Note that, if needed, pc will be corrected during the evaluation of jump operations.

After this phase, parties shares the opcode $\langle \mathsf{op} \rangle$, 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 $\langle \mathsf{r}i \rangle$, $\langle \mathsf{r}j \rangle$ and $\langle a \rangle$ by our o-read protocol in parallel as follows. For $\langle \mathsf{r}i \rangle$ and $\langle \mathsf{r}j \rangle$, four MPC players simply invoke $\Pi^{m,\ell}_{\mathsf{read}}$ 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 $\Pi^{(2^{\lceil \log_2 m \rceil}+m),\ell}_{\mathsf{read}}$ with the indicator $\langle d \rangle$ over the replicated shared array in the form of

$$\mathbf{v} := \underbrace{A, \ldots, A}_{2^{\lceil \log_2 m \rceil}} || (\mathsf{r}k)_{k \in \mathbb{Z}_m} \quad .$$

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 $\Pi^{(2^{\lceil \log_2 m \rceil}+m),\ell}_{\mathsf{read}}$.

4.2.2 Operator Determination

We determinate the operator type of current instruction using the DPF scheme. After this step, $\langle \mathsf{op} \rangle$ is converted to a unit vector of 25 elements in the shared form, where the op-th element is 1 and all the other elements are 0's. Namely, the MPC players share the point function $f_{\mathsf{op},1}(x) : \mathbb{Z}_{2^5} \to \mathbb{Z}_{2^\ell}$. More specifically, to construct the shares of $f_{\mathsf{op},1}(x)$ without revealing op, P_3 generates a pair of DPF keys for the point function $f_{\psi_1,1}(x) : \mathbb{Z}_{2^5} \to \mathbb{Z}_{2^\ell}$, where $\psi_1 \in \mathbb{Z}_{2^5}$ 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 $f_{\psi_2,1}(x) : \mathbb{Z}_{2^5} \to \mathbb{Z}_{2^\ell}$, where $\psi_2 \in \mathbb{Z}_{2^5}$ is randomly picked, and then distributes them to P_3, P_4. Meanwhile, four MPC players open $\langle \delta_1 \rangle := \langle \mathsf{op} \rangle + \psi_1 \pmod{2^5}$ to P_1, P_2, $\langle \delta_2 \rangle := \langle \mathsf{op} \rangle + \psi_2 \pmod{2^5}$ 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 $[\![f_{\psi_1,1}(x)]\!]$ 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 $\delta_1 - \psi_1 = \delta_2 - \psi_2 = \mathsf{op}$.

4.3 Operation Evaluation

Let $\tilde{\mathsf{op}} \in \mathbb{Z}_{25}$ denote our 25 supported operations while 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 $\tilde{\mathsf{op}}$ will produce results after execution, usually $\{\mathsf{res}_{\tilde{\mathsf{op}}}, \mathsf{flag}_{\tilde{\mathsf{op}}}\}$.[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 op. Therefore, in the evaluation of some operations, we can use the (intermediate) result of

[1] The $*$ symbol in $\Pi^{n,*}_{\mathsf{read}}$ 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 ans is used to signify the program has finished, thus $\mathsf{res}_{\mathsf{ans}}$ and $\mathsf{flag}_{\mathsf{ans}}$ are not required.

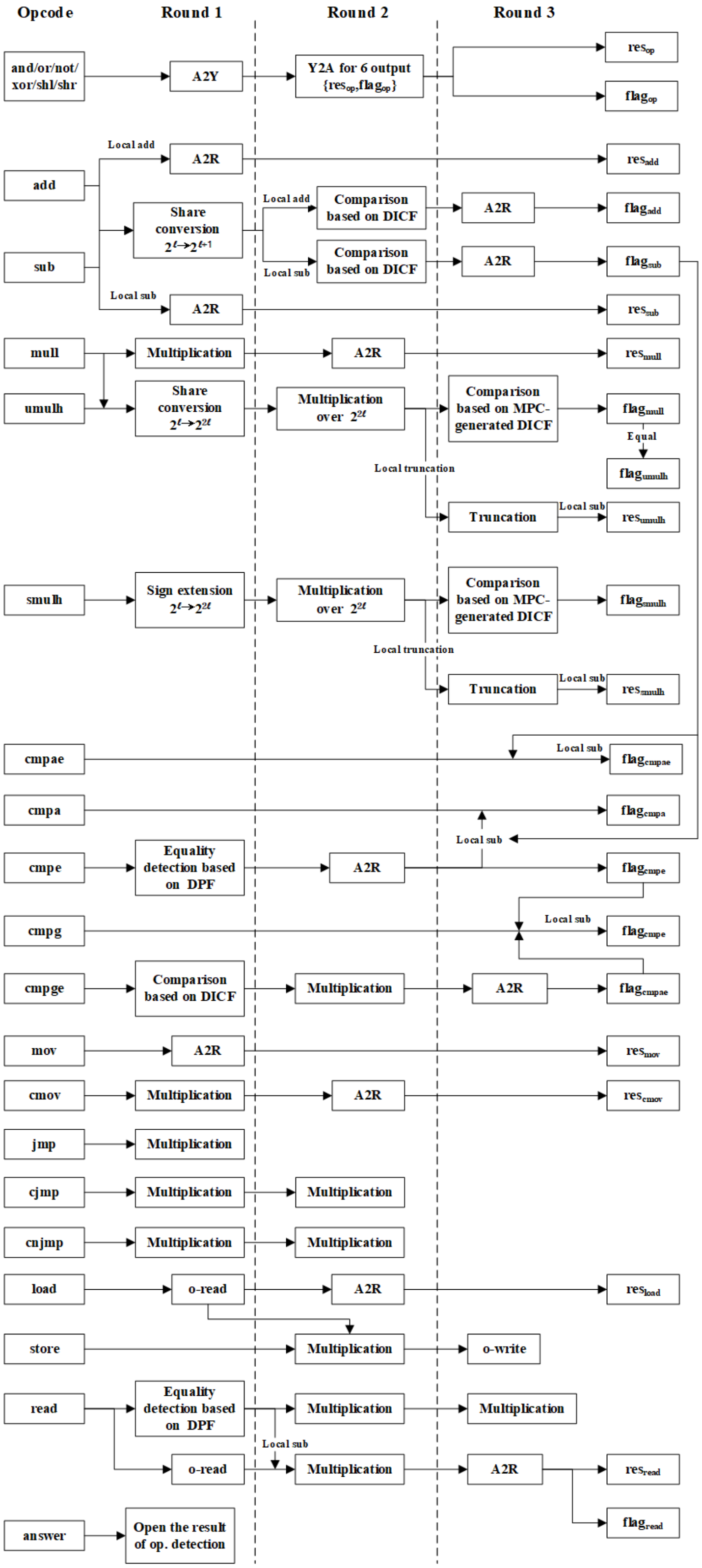


Figure 7: Operations computation: Except for the answer, each $\tilde{op}$ has corresponding $\mathsf{res}_{\tilde{op}}$ and $\mathsf{flag}_{\tilde{op}}$. For ease of representation, the $\mathsf{res}_{\tilde{op}}$ and $\mathsf{flag}_{\tilde{op}}$ whose values must be ri and flag are omitted in the figure.

others to reduce the communication and computation. In addition, since some operations' results without $\mathsf{res}_{\widetilde{\mathsf{op}}}$ or $\mathsf{flag}_{\widetilde{\mathsf{op}}}$ (i.e., these operations do not modify registers or condition flag), we convert $\langle \mathsf{r}i\rangle, \langle \mathsf{flag}\rangle$ into $\langle \mathsf{r}i\rangle^{\mathsf{rep}}, \langle \mathsf{flag}\rangle^{\mathsf{rep}}$ at the same round as instruction decoding, and set $\mathsf{res}_{\widetilde{\mathsf{op}}} := \langle \mathsf{r}i\rangle^{\mathsf{rep}}$ (or $\mathsf{flag}_{\widetilde{\mathsf{op}}} := \langle \mathsf{flag}\rangle^{\mathsf{rep}}$) if $\mathsf{res}_{\widetilde{\mathsf{op}}}$ (or $\mathsf{flag}_{\widetilde{\mathsf{op}}}$) 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 $\mathsf{and}, \mathsf{or}, \mathsf{xor}, \mathsf{not}, \mathsf{shl}, \mathsf{shr}$. 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 $\mathsf{flag}_{\mathsf{mull}}$ (and $\mathsf{flag}_{\mathsf{smulh}}$) computation causes the additional communication round. To avoid that, we let P_1 and P_3 jointly generate DICF keys $(\mathcal{K}_t^{(1)}, \mathcal{K}_t^{(2)})$ for the offset interval containment function $f^{\mathsf{IC}}_{0,2^\ell-1,t}(x) : \mathbb{Z}_{2^{2\ell}} \to \mathbb{Z}_{2^\ell}$ via MPC, and send $\mathcal{K}_t^{(1)}, \mathcal{K}_t^{(2)}$ 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 $\langle \mathsf{flag}_{\mathsf{mull}}\rangle^{\mathsf{rep}}$ 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 $[\![f_{(\delta_1-\psi_1),1}(x)]\!]$ and $[\![f_{(\delta_2-\psi_2),1}(x)]\!]$, and then obliviously write them into flag and i-th register by o-write protocol.

Note that after previous phases, the MPC players already obtain $\langle \mathsf{r}i\rangle^{\mathsf{rep}}$, $\langle \mathsf{flag}\rangle^{\mathsf{rep}}$, $\{\langle \mathsf{res}_k\rangle^{\mathsf{rep}}\}_{k\in\mathbb{Z}_{24}}$ and $\{\langle \mathsf{flag}_k\rangle^{\mathsf{rep}}\}_{k\in\mathbb{Z}_{24}}$ 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

$$\langle \mathsf{res}\rangle := \sum_{k=0}^{23} \mathsf{res}_k^{(1)} \cdot [\![f_{(\delta_1-\psi_1),1}(k)]\!] + \mathsf{res}_k^{(3)} \cdot [\![f_{(\delta_2-\psi_2),1}(k)]\!] \ ,$$

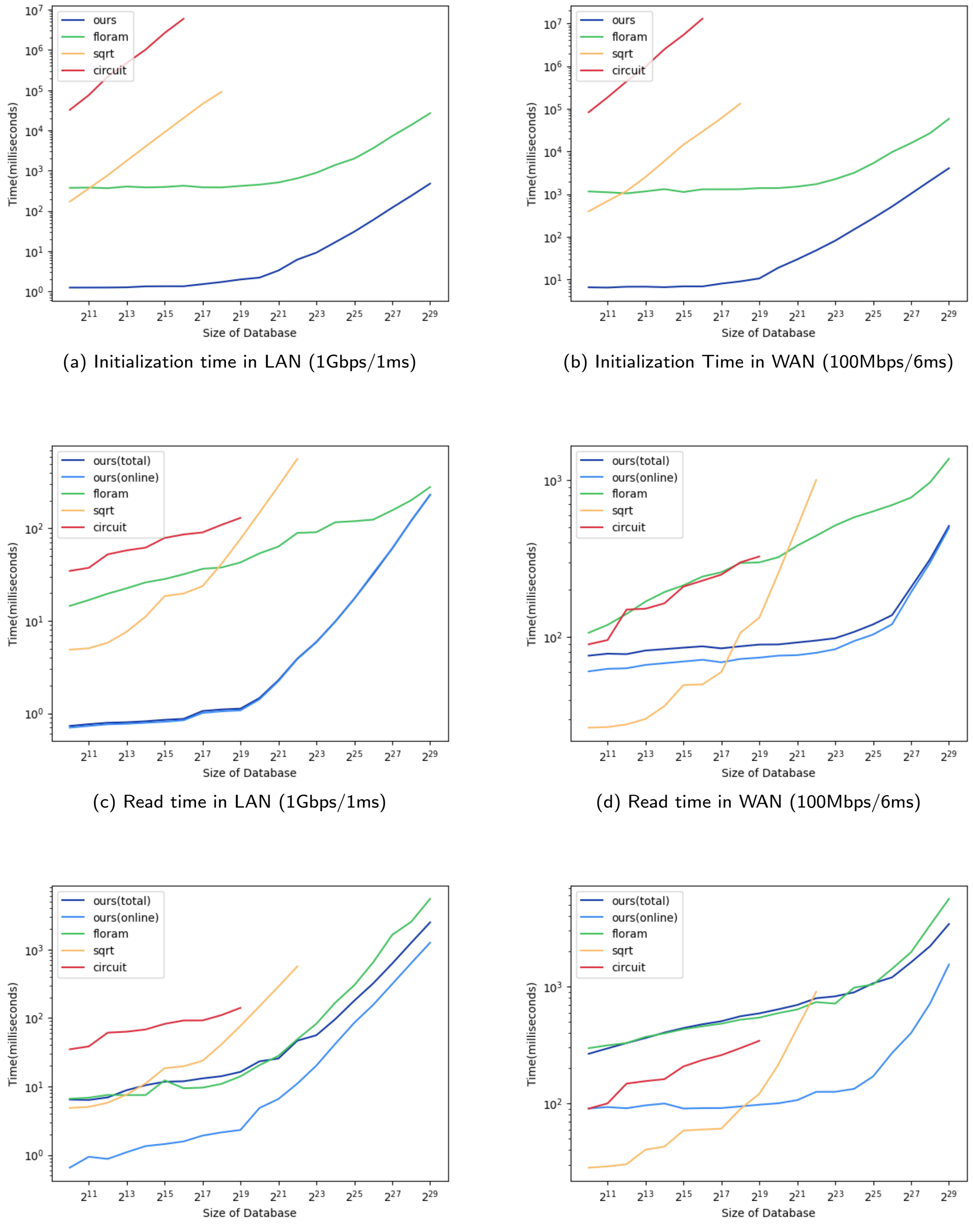
$$\langle \mathsf{flag}\rangle := \sum_{k=0}^{23} \mathsf{flag}_k^{(1)} \cdot [\![f_{(\delta_1-\psi_1),1}(k)]\!] + \mathsf{flag}_k^{(3)} \cdot [\![f_{(\delta_2-\psi_2),1}(k)]\!]$$

over the ring $\mathbb{Z}_{2^\ell}$ without communication. MPC players then re-randomize the shares of $\langle \mathsf{res}\rangle$ and $\langle \mathsf{flag}\rangle$ to ensure their uniform distribution. It is obvious that the condition flag has been updated. To obliviously write $\langle \mathsf{res}\rangle$ into i-th register, the MPC players jointly invoke the o-write protocol with the index $\langle i\rangle$, the old value $\langle \mathsf{r}i\rangle$ and the new value $\langle \mathsf{res}\rangle$. 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.



(a) Initialization time in LAN (1Gbps/1ms)

(b) Initialization Time in WAN (100Mbps/6ms)

(c) Read time in LAN (1Gbps/1ms)

(d) Read time in WAN (100Mbps/6ms)

(e) Write time in LAN (1Gbps/1ms)

(f) Write time in WAN (100Mbps/6ms)

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

Input Size	32	64	128	256	1024
Set Intersection					
# of cycles	427	907	1603	3643	13747
Offline time (s)	12.97	27.27	50.01	110.70	420.72
Fetch time (s)	0.16	0.39	0.72	1.36	5.58
Decode time (s)	0.32	0.67	1.26	2.86	10.65
Eval. time (s)	8.37	17.46	30.47	70.68	269.10
Write time (s)	0.75	1.53	2.94	7.19	23.61
Total time (s)	22.54	47.32	85.39	192.79	729.66
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^{21} 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 ℓ 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 ℓ 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:

$\mathrm{I}_0 : \mathsf{read}\ \%0, 0$
$\mathrm{I}_1 : \mathsf{read}\ \%1, 0$
$\mathrm{I}_2 : \mathsf{read}\ \%2, 0$
$\mathrm{I}_3 : \mathsf{read}\ \%3, 0$
$\mathrm{I}_4 : \mathsf{cmpa}\ \%1, \%0$
$\mathrm{I}_5 : \mathsf{cnjmp}\ \mathrm{I}_{22}$
$\mathrm{I}_6 : \mathsf{cmpa}\ \%3, \%2$
$\mathrm{I}_7 : \mathsf{cnjmp}\ \mathrm{I}_{22}$
$\mathrm{I}_8 : \mathsf{load}\ \%30, \%0$
$\mathrm{I}_9 : \mathsf{load}\ \%31, \%2$
$\mathrm{I}_{10} : \mathsf{cmpe}\ \%31, \%30$
$\mathrm{I}_{11} : \mathsf{cnjmp}\ \mathrm{I}_{16}$
$\mathrm{I}_{12} : \mathsf{add}\ \%0, \%0, 1$
$\mathrm{I}_{13} : \mathsf{add}\ \%2, \%2, 1$
$\mathrm{I}_{14} : \mathsf{add}\ \%29, \%29, 1$
$\mathrm{I}_{15} : \mathsf{jmp}\ \mathrm{I}_4$
$\mathrm{I}_{16} : \mathsf{cmpa}\ \%30, \%31$
$\mathrm{I}_{17} : \mathsf{cnjmp}\ \mathrm{I}_{20}$
$\mathrm{I}_{18} : \mathsf{add}\ \%2, \%2, 1$
$\mathrm{I}_{19} : \mathsf{jmp}\ \mathrm{I}_4$
$\mathrm{I}_{20} : \mathsf{add}\ \%0, \%0, 1$
$\mathrm{I}_{21} : \mathsf{jmp}\ \mathrm{I}_4$
$\mathrm{I}_{22} : \mathsf{ans}\ \%29$

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

Input Size			64	256	1024	4096
WGMK16 [39]			58.35	324.09	3068.19	-
Marcel17 [41]			6.43	44.12	1346.82	-
Ours	best	online	18.10	68.34	275.65	1161.49
		total	42.53	161.03	649.46	2798.08
	worst	online	32.56	135.83	547.16	2553.93
		total	77.99	331.69	1303.22	5956.56

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

	P_1	P_2	P_3	P_4	P_5	P_6	P_7
$\mathbf{x}^{(1)}$	0	0	0	1	1	1	1
$\mathbf{x}^{(2)}$	0	1	1	0	0	1	1
$\mathbf{x}^{(3)}$	0	1	1	1	1	0	0
$\mathbf{x}^{(4)}$	1	0	1	0	1	0	1
$\mathbf{x}^{(5)}$	1	1	0	1	0	1	0

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 collusion 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 $\mathbf{x}^{(i)}$ by abusing notations, of the secret $\mathbf{x}$ such that $\sum_{i=1}^{t+1} \mathbf{x}^{(i)} = \mathbf{x}$. In o-read operation, parties in $\mathcal{P}_i$ jointly evaluate the p_1-party DPF for $f_{\varphi_i,1}$ 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 $f_{r_1,1}, f_{r_1,v}$ 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 $\mathbf{x}^{(i)}$ 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:

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

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 $O(\lambda \ell \sqrt{n \log^3 n})$ 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 piratical 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 $\mathcal{D} := \{\mathsf{and}, \mathsf{or}, \mathsf{xor}, \mathsf{not}, \mathsf{shl}, \mathsf{shr}\}$, 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.

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

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

$$\mathsf{flag}^{(3)}_{\tilde{\mathsf{op}}} := F_{\mathsf{flag}}(F_{\tilde{\mathsf{op}}}(\sum_{q=1}^{4} \mathsf{r}j^{(q)}, \sum_{q=1}^{4} a^{(q)})) - \mathsf{flag}^{(1)}_{\tilde{\mathsf{op}}}$$

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

A.2 Arithmetic Operations

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

add&sub: For $\tilde{\mathsf{op}} \in \{\mathsf{add}, \mathsf{sub}\}$, we compute $\langle \mathsf{r}j + a \rangle$ or $\langle \mathsf{r}j - a \rangle$ over the ring $\mathbb{Z}_{2^\ell}$ and store the result to $\langle \mathsf{res}_{\tilde{\mathsf{op}}} \rangle^{\mathsf{rep}}$ after the A2R share conversion. To determine $\mathsf{flag}_{\tilde{\mathsf{op}}}$, we first convert the shared $\mathsf{r}j$ 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 $2^\ell - 1$. $\mathsf{flag}_{\tilde{\mathsf{op}}}$ is set to 1 if the result is greater than $2^\ell - 1$ and to 0 otherwise. More specifically, it works as follows.

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

$\underline{\mathsf{flag}_{\mathsf{add}} \text{ (or } \mathsf{flag}_{\mathsf{sub}})}$. When shares are individually converted from $\mathbb{Z}_{2^\ell}$ to $\mathbb{Z}_{2^{\ell+1}}$, the extra carry-in value 2^ℓ may appear. Thus the MPC players need to check whether $\sum_{q=1}^{4} \mathsf{r}i^{(q)}, \sum_{q=1}^{4} a^{(q)} \in [0, 2^\ell - 1]$ 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 $f^{\mathsf{IC}}_{0,2^\ell-1,v}(x) : \mathbb{Z}_{2^{\ell+1}} \to \mathbb{Z}_{2^{\ell+1}}$, where $v \leftarrow \mathbb{Z}_{2^{\ell+1}}$ is random offset, and then distributes DICF keys to P_2, P_4. Similarly, P_2 generates DICF keys of $f^{\mathsf{IC}}_{0,2^\ell-1,r}(x)$ and distributes keys to P_1, P_3. After that, P_1, P_3 hold $[\![f^{\mathsf{IC}}_{0,2^\ell-1,r}(x)]\!]$, and P_2, P_4 hold $[\![f^{\mathsf{IC}}_{0,2^\ell-1,v}(x)]\!]$. Parties then open $\langle \mathsf{r}j' \rangle := \langle \mathsf{r}j \rangle + r \pmod{2^{\ell+1}}$ to P_1, P_3, $\langle a' \rangle := \langle a \rangle + v \pmod{2^{\ell+1}}$ to P_2, P_4, and jointly compute

$$\langle \hat{\mathsf{r}j} \rangle := \langle \mathsf{r}j \rangle - (1 - [\![f^{\mathsf{IC}}_{0,2^\ell-1,r}(\mathsf{r}j')]\!]) \cdot 2^\ell \pmod{2^{\ell+1}},$$
$$\langle \hat{a} \rangle := \langle a \rangle - (1 - [\![f^{\mathsf{IC}}_{0,2^\ell-1,v}(a')]\!]) \cdot 2^\ell \pmod{2^{\ell+1}}$$

by evaluating DICF keys in local.

Later, the MPC players jointly compare the $2^\ell - 1$ with the addition (or subtraction) result of $\hat{\mathsf{r}j}$ and $\hat{a}$ over $\mathbb{Z}_{2^{\ell+1}}$. More specifically, we let P_1 generate DICF keys of $f^{\mathsf{IC}}_{0,2^\ell-1,t}(x) : \mathbb{Z}_{2^{\ell+1}} \to \mathbb{Z}_{2^\ell}$, 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

Protocol $\Pi^{\ell}_{\mathsf{mul}}$

- Upon receiving $(\textsc{Mul}, \mathsf{sid}, \mathsf{r}j^{(q)}, a^{(q)})$:
    - $P_{q\in\{1,2\}}$ sends $(\mathsf{sid}, \mathsf{r}j^{(q)})$ to P_{3-q}, $(\mathsf{sid}, a^{(q)})$ to P_{q+2};
    - $P_{q\in\{3,4\}}$ sends $(\mathsf{sid}, \mathsf{r}j^{(q)})$ to P_{7-q}, $(\mathsf{sid}, a^{(q)})$ to P_{q-2};
- Upon receiving $(\mathsf{sid}, \mathsf{r}j^{(3-q)})$, $(\mathsf{sid}, a^{(q+2)})$, $P_{q\in\{1,2\}}$ sets:
    - $\mathsf{res}^{(q)} := (\mathsf{r}j^{(3-q)} + \mathsf{r}j^{(q)}) \cdot (a^{(q)} + a^{(q+2)}) \pmod{2^{\ell}}$;
- Upon receiving $(\mathsf{sid}, \mathsf{r}j^{(7-q)})$, $(\mathsf{sid}, a^{(q-2)})$, $P_{q\in\{3,4\}}$ sets:
    - $\mathsf{res}^{(q)} := (\mathsf{r}j^{(7-q)} + \mathsf{r}j^{(q)}) \cdot (a^{(q)} + a^{(q-2)}) \pmod{2^{\ell}}$;

Figure 9: 4-party multiplication protocol $\Pi^{\ell}_{\mathsf{mul}}$.

$\langle y_t \rangle := \langle \widehat{\mathsf{r}j} \rangle + \langle \hat{a} \rangle + t \pmod{2^{\ell+1}}$ to P_2, P_4. Upon receiving the DICF keys and y_t, P_2 and P_4 jointly compute

$$[\![\mathsf{flag}_{\mathsf{add}}]\!] := 1 - [\![f^{\mathsf{IC}}_{0,2^{\ell}-1,t}(y_t)]\!] \pmod{2^{\ell}}$$

by evaluating DICF keys in local. Finally, to construct $\langle \mathsf{flag}_{\mathsf{add}} \rangle^{\mathsf{rep}}$ among four parties, P_2 sends its share of $\mathsf{flag}_{\mathsf{add}}$ to P_1 and P_4 sends its share to P_3.

mull: This operation computes $\langle \mathsf{r}j \cdot a \rangle$ over the ring $\mathbb{Z}_{2^{\ell}}$ and stores the result to $\mathsf{res}_{\mathsf{mull}}$ in the replicated-shared form. Analogous to the $\mathsf{flag}_{\mathsf{add}}$ determination, to check whether $\mathsf{r}j \cdot a \notin \mathbb{Z}_{2^{\ell}}$, we first convert the shared $\mathsf{r}j$ 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 $2^{\ell} - 1$. $\mathsf{flag}_{\mathsf{mull}}$ is set to 1 if the result is greater than $2^{\ell} - 1$ and to 0 otherwise. More specifically, it works as follows.

$\underline{\mathsf{res}_{\mathsf{mull}}}$. The MPC players jointly invoke our 4-party multiplication protocol $\Pi^{2\ell}_{\mathsf{mul}}$ (cf. Fig. 9) to compute $\langle \mathsf{res}_{\mathsf{mull}} \rangle := \langle \mathsf{r}j \rangle \cdot \langle a \rangle \pmod{2^{\ell}}$, and then convert $\langle \mathsf{res}_{\mathsf{mull}} \rangle$ to $\langle \mathsf{res}_{\mathsf{mull}} \rangle^{\mathsf{rep}}$.

$\underline{\mathsf{flag}_{\mathsf{mull}}}$. First, parties convert $\langle \mathsf{r}j \rangle, \langle a \rangle$ from $\mathbb{Z}_{2^{\ell}}$ to $\mathbb{Z}_{2^{2\ell}}$. This step is similar to $\mathsf{flag}_{\mathsf{add}}$ 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 $f^{\mathsf{IC}}_{0,2^{\ell}-1}(x) : \mathbb{Z}_{2^{\ell+1}} \to \mathbb{Z}_{2^{2\ell}}$ and $f^{\mathsf{IC}}_{0,2^{\ell+1}-1}(x) : \mathbb{Z}_{2^{\ell+2}} \to \mathbb{Z}_{2^{\ell}}$ to detect the values of ℓ-th bit and $\ell+1$-th bit, respectively. Denote the converted shares as $\langle \widehat{\mathsf{r}j} \rangle$ and $\langle \hat{a} \rangle$. Parties then jointly compute $\langle y \rangle := \langle \widehat{\mathsf{r}j} \rangle \cdot \langle \hat{a} \rangle \pmod{2^{2\ell}}$ and compare $\langle y \rangle$ with 2^{ℓ} to get $\langle \mathsf{flag}_{\mathsf{mull}} \rangle^{\mathsf{rep}}$ as shown in Fig. 10.

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

$\underline{\mathsf{res}_{\mathsf{umulh}}}$. The MPC players first convert $\langle \mathsf{r}j \rangle, \langle a \rangle$ from $\mathbb{Z}_{2^{\ell}}$ to $\mathbb{Z}_{2^{2\ell}}$, and multiply them over the ring $\mathbb{Z}_{2^{2\ell}}$. This step to obtain $\langle \mathsf{r}j \cdot a \rangle \in \mathbb{Z}_{2^{2\ell}}$ is the same as in $\mathsf{flag}_{\mathsf{mull}}$. In other words, parties can straightly use the multiplication result $\langle y \rangle$ from $\mathsf{flag}_{\mathsf{mull}}$ computation. After that, parties jointly truncate $\langle y \rangle$ to get the shared value of y's most significant ℓ 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^{ℓ}), two carry-in bits may be lost. To correct this error, parties begin by generating a truncation pair $\langle t \rangle^{\mathsf{rep}}, \langle t' \rangle := \langle t \cdot 2^{\ell} \rangle$ over the ring $\mathbb{Z}_{2^{2\ell}}$, and then open $\langle y_t \rangle := \langle y \rangle - \langle t' \rangle \pmod{2^{2\ell}}$. After that, parties can jointly compute

$$\langle y_h \rangle^{\mathsf{rep}} := \langle t \rangle^{\mathsf{rep}} + y_t >> \ell \pmod{2^{2\ell}},$$

hence, $y_h = y >> \ell$. Finally, $\langle \mathsf{res}_{\mathsf{umulh}} \rangle^{\mathsf{rep}}$ can be obtained by individually removing the most significant ℓ bits of $\langle y_h \rangle^{\mathsf{rep}}$.

$\underline{\mathsf{flag}_{\mathsf{umulh}}}$. For $q \in [4]$, the player P_q sets $\mathsf{res}^{(q)}_{\mathsf{umulh}} := \mathsf{res}^{(q)}_{\mathsf{mull}}$.

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

$\underline{\mathsf{res}_{\mathsf{smulh}}}$. The MPC players first check whether $\mathsf{r}j, a \in [0, 2^{\ell-1}]$ to extract their sign bits $[\![s_r]\!]$ and $[\![s_a]\!]$ respectively. In detail, P_1 and P_2 generate DICF keys for functions $f^{\mathsf{IC}}_{0,2^{\ell-1},v}(x), f^{\mathsf{IC}}_{0,2^{\ell-1},r}(x) : \mathbb{Z}_{2^{\ell}} \to \mathbb{Z}_{2^{2\ell}}$

Protocol $\Pi^{\ell}_{\mathsf{flag}_{\mathsf{mull}}}$

Offline phase:

- Upon initializetion, P_1 does:
  - Set $v_k, \leftarrow \mathbb{Z}_{2^{\ell+k}}$, $v_k^{(1)} := v_k - \mathsf{PRF}_{\eta_2}^{\mathbb{Z}_{2^{\ell+k}}}(\mathsf{sid}, 0)$, $k \in [2]$;
  - Set $(\mathcal{K}_{v,k}^{(1)}, \mathcal{K}_{v,k}^{(2)}) \leftarrow \mathsf{DICF.Gen}^{\mathsf{IC}}_{0,k\cdot 2^{\ell}-1}(1^{\lambda}, f^{\mathsf{IC}}_{0,k\cdot 2^{\ell}-1,v_k})$, $k \in [2]$;
  - Generate $t \leftarrow \mathbb{Z}_{2^{2\ell}}$,
  - Set $(\mathcal{K}_t^{(1)}, \mathcal{K}_t^{(2)}) \leftarrow \mathsf{DICF.Gen}^{\mathsf{IC}}_{0,k\cdot 2^{\ell}-1}(1^{\lambda}, f^{\mathsf{IC}}_{0,k\cdot 2^{\ell}-1,t})$;
  - Send $(\mathsf{sid}, (\mathcal{K}_{v,k}^{(1)})_{k\in[2]}, \mathcal{K}_t^{(1)})$ to P_2;
  - Send $(\mathsf{sid}, (\mathcal{K}_{v,k}^{(2)})_{k\in[2]}, \mathcal{K}_t^{(2)})$ to P_4;
- Upon initializetion, P_2 does:
  - Set $r_k \leftarrow \mathbb{Z}_{2^{\ell+k}}$, $r_k^{(1)} := r_k - \mathsf{PRF}_{\eta_4}^{\mathbb{Z}_{2^{\ell+k}}}(\mathsf{sid}, 0)$, $k \in [2]$;
  - Set $(\mathcal{K}_{r,k}^{(1)}, \mathcal{K}_{r,k}^{(2)}) \leftarrow \mathsf{DICF.Gen}^{\mathsf{IC}}_{0,k\cdot 2^{\ell}-1}(1^{\lambda}, f^{\mathsf{IC}}_{0,k\cdot 2^{\ell}-1,r_k})$, $k \in [2]$;
  - Send $(\mathsf{sid}, (\mathcal{K}_{r,k}^{(1)})_{k\in[2]})$ to P_1, $(\mathsf{sid}, (\mathcal{K}_{r,k}^{(2)})_{k\in[2]})$ to P_3;
- Upon initialization, P_2 sets $v_k^{(2)} := \mathsf{PRF}_{\eta_2}^{\mathbb{Z}_{2^{\ell+k}}}(\mathsf{sid}, 0)$, $k \in [2]$;
- Upon initialization, P_4 sets $r_k^{(2)} := \mathsf{PRF}_{\eta_4}^{\mathbb{Z}_{2^{\ell+k}}}(\mathsf{sid}, 0)$, $k \in [2]$;

Online phase:

- Upon receiving $(\textsc{FlagMull}, \mathsf{sid}, \mathsf{r}j^{(q)}, a^{(q)})$ from the environment $\mathcal{Z}$, player P_q, $q \in \{1, 2\}$ does:
  - For $k \in [2]$:
    - * Set $w_{1,q,k} \leftarrow \mathsf{PRF}_{\eta_{2q-1}}^{\mathbb{Z}_{2^{\ell+k}}}(\mathsf{sid}, 1)$;
    - * Set $w_{2,q,k} \leftarrow \mathsf{PRF}_{\eta_{2q-1}}^{\mathbb{Z}_{2^{\ell+k}}}(\mathsf{sid}, 2)$;
    - * Set $\mathsf{r}j_k^{(q)} := \mathsf{r}j^{(q)} + w_{1,q,k} \pmod{2^{\ell+k}}$;
    - * Set $a_k^{(q)} := a^{(q)} + v_k^{(q)} + w_{2,q,k} \pmod{2^{\ell+k}}$;
  - Send $(\mathsf{sid}, \mathsf{r}j_1^{(q)}, \mathsf{r}j_2^{(q)})$ to P_1, P_3, $(\mathsf{sid}, a_1^{(q)}, a_2^{(q)})$ to P_2, P_4;
- Upon receiving $(\textsc{FlagMull}, \mathsf{sid}, \mathsf{r}j^{(q)}, a^{(q)})$ from the environment $\mathcal{Z}$, player P_q, $q \in \{3, 4\}$ does:
  - For $k \in [2]$:
    - * Set $w_{1,q-2,k} \leftarrow \mathsf{PRF}_{\eta_{2q-5}}^{\mathbb{Z}_{2^{\ell+k}}}(\mathsf{sid}, 1)$;
    - * Set $w_{2,q-2,k} \leftarrow \mathsf{PRF}_{\eta_{2q-5}}^{\mathbb{Z}_{2^{\ell+k}}}(\mathsf{sid}, 2)$
    - * Set $\mathsf{r}j_k^{(q)} := \mathsf{r}j^{(q)} + r^{(q)} - w_{1,q-2} \pmod{2^{2\ell}}$;
    - * Set $a_k^{(q)} := a^{(q)} - w_{2,q-2} \pmod{2^{2\ell}}$;
  - Send $(\mathsf{sid}, \mathsf{r}j_1^{(q)}, \mathsf{r}j_2^{(q)})$ to P_1, P_3, $(\mathsf{sid}, a_1^{(q)}, a_2^{(q)})$ to P_2, P_4;
- Upon receiving $(\mathsf{sid}, \mathsf{r}j_1^{(3-q)}, \mathsf{r}j_2^{(3-q)})$ from P_{3-q}, $(\mathsf{sid}, \mathsf{r}j_1^{(3)}, \mathsf{r}j_2^{(3)})$ from P_3 , and $(\mathsf{sid}, \mathsf{r}j_1^{(4)}, \mathsf{r}j_2^{(4)})$ from P_4, $P_q, q \in \{1, 3\}$ does:
  - Set $\mathsf{r}j_k := \sum_{q=1}^{4} \mathsf{r}j_k^{(q)} \pmod{2^{\ell+k}}$ for $k \in [2]$;
  - Set $\beta_{r,k}^{(q)} \leftarrow \mathsf{DICF.Eval}^{\mathsf{IC}}_{0,k\cdot 2^{\ell}-1}(q, \mathcal{K}_{r,k}^{(q)}, \mathsf{r}j_k), k \in [2]$;
  - Set $\hat{\mathsf{r}j}^{(q)} := \mathsf{r}j^{(q)} + \sum_{k=1}^{2}(\beta_{r,k}^{(q)} \cdot k) \cdot 2^{\ell} \pmod{2^{2\ell}}$;
  - Set $\hat{a}^{(q)} := a^{(q)} - \lceil q/2 \rceil \cdot 2^{\ell} \pmod{2^{2\ell}}$;
  - Invoke $\Pi^{2\ell}_{\mathsf{mul}}$ with $(\textsc{Mul}, \mathsf{sid}, \hat{\mathsf{r}j}^{(q)}, \hat{a}^{(q)})$ to get $y_t^{(q)}$;
  - If $q = 1$, $y_t^{(q)} := y^{(q)} + t \pmod{2^{2\ell}}$;
  - Send $(\mathsf{sid}, y_t^{(q)})$ to P_2 and P_4;
- Upon receiving $(\mathsf{sid}, a_1^{(1)}, a_2^{(1)})$ from P_1, $(\mathsf{sid}, a_1^{(2)}, a_2^{(2)})$ from P_2, $(\mathsf{sid}, a_1^{(7-q)}, a_2^{(7-q)})$ from P_{7-q}, $P_q, q \in \{2, 4\}$ does:
  - Set $a_k := \sum_{q=1}^{4} a_k^{(q)} \pmod{2^{2\ell}}$ for $k \in [2]$;
  - $\beta_{v,k}^{(q-2)} \leftarrow \mathsf{DICF.Eval}^{\mathsf{IC}}_{0,k\cdot 2^{\ell}-1}(q-2, \mathcal{K}_{v,k}^{(q-2)}, a_k), k \in [2]$;
  - Set $\hat{\mathsf{r}j}^{(q)} := \mathsf{r}j^{(q)} - \lceil q/2 \rceil \cdot 2^{\ell} \pmod{2^{2\ell}}$
  - Set $\hat{a}^{(q)} := a^{(q)} + \sum_{k=1}^{2}(\beta_{v,k}^{(q-2)} \cdot k) \cdot 2^{\ell} \pmod{2^{2\ell}}$;
  - Invoke $\Pi^{2\ell}_{\mathsf{mul}}$ with $(\textsc{Mul}, \mathsf{sid}, \hat{\mathsf{r}j}^{(q)}, \hat{a}^{(q)})$ to get $y^{(q)}$;
  - Send $(\mathsf{sid}, y_t^{(q)})$ to P_{6-q};
- Upon receiving $(\mathsf{sid}, y_t^{(k)})$ from $P_{k\in[4]/q}$, $P_{q\in\{2,4\}}$ does:
  - Set $y_t := y_t^{(1)} + y_t^{(2)} + y_t^{(3)} + y_t^{(4)} \pmod{2^{2\ell}}$;
  - Set $\mathsf{flag}_{\mathsf{mull}}^{(q/2)} \leftarrow \mathsf{DICF.Eval}^{\mathsf{IC}}_{0,2^{\ell}-1}(q/2, \mathcal{K}_t^{(q/2)}, y_t)$;
  - Send $\mathsf{flag}_{\mathsf{mull}}^{(q/2)}$ to P_{q-1}.

Figure 10: mull overflow detection protocol $\Pi^{\ell}_{\mathsf{flag}_{\mathsf{mull}}}$.

respectively, where $v, r \leftarrow \mathbb{Z}_{2^\ell}$ are random offsets. P_1 (and P_2) then distribute their DICF keys to P_2, P_4 (and P_1, P_3). Parties then open $\langle \mathsf{r}j' \rangle := \langle \mathsf{r}j \rangle + r \pmod{2^\ell}$ to P_1, P_3, $\langle a' \rangle := \langle a \rangle + v \pmod{2^\ell}$ to P_2, P_4, and jointly compute

$$\llbracket s_r \rrbracket := 1 - \llbracket f^{\mathsf{IC}}_{0,2^\ell-1,r}(\mathsf{r}j') \rrbracket \pmod{2^{2\ell}},$$
$$\llbracket s_a \rrbracket := 1 - \llbracket f^{\mathsf{IC}}_{0,2^\ell-1,v}(a') \rrbracket \pmod{2^{2\ell}}$$

by locally evaluating DICF keys. Meanwhile, four parties convert $\langle \mathsf{r}j \rangle$ and $\langle a \rangle$ from $\mathbb{Z}_{2^\ell}$ to $\mathbb{Z}_{2^{2\ell}}$ as unsigned integers, i.e., they can copy the converted shares of $\langle \widehat{\mathsf{r}j} \rangle$ and $\langle \hat{a} \rangle$ from $\mathsf{flag}_{\mathsf{mull}}$ computation. After that, the MPC players obtain the sign extension values

$$\langle \widehat{\mathsf{r}j}_s \rangle := \langle \widehat{\mathsf{r}j} \rangle + (2^{2\ell} - 2^\ell) \cdot \llbracket s_r \rrbracket \pmod{2^{2\ell}},$$
$$\langle \hat{a}_s \rangle := \langle \hat{a} \rangle + (2^{2\ell} - 2^\ell) \cdot \llbracket s_a \rrbracket \pmod{2^{2\ell}}$$

without communication. Next, parties compute $\langle y_s \rangle := \langle \widehat{\mathsf{r}j}_s \rangle \cdot \langle \hat{a}_s \rangle \pmod{2^{2\ell}}$, and then truncate the most significant ℓ bits of $\langle y_s \rangle$ to get $\langle \mathsf{res}_{\mathsf{smulh}} \rangle^{\mathsf{rep}}$ by the same method as in $\mathsf{res}_{\mathsf{umulh}}$ computation.

$\underline{\mathsf{flag}_{\mathsf{smulh}}}$. The MPC players detect whether $y_s \in [2^{\ell-1}, 2^{2\ell} - 2^{\ell-1} - 1]$ based on DICF schemes. More specifically, P_1 generates DICF keys of the functions $f^{\mathsf{IC}}_{2^{\ell-1}, 2^{2\ell}-2^{\ell-1}-1, t}(x) : \mathbb{Z}_{2^{2\ell}} \to \mathbb{Z}_{2^\ell}$ where $t \leftarrow \mathbb{Z}_{2^\ell}$ is random offset, and distributes them to P_2, P_4. Parties then open $\langle y_t \rangle := \langle y_s \rangle + t \pmod{2^{2\ell}}$ to P_2, P_4. Upon receiving y_t, P_2 and P_4 jointly compute

$$\llbracket \mathsf{flag}_{\mathsf{smulh}} \rrbracket := 1 - \llbracket f^{\mathsf{IC}}_{2^{\ell-1}, 2^{2\ell}-2^{\ell-1}-1, t}(y_t) \rrbracket \pmod{2^\ell}$$

by locally evaluating DICF keys. Finally, to construct $\langle \mathsf{flag}_{\mathsf{add}} \rangle^{\mathsf{rep}}$ among four parties, P_2 sends its share of $\mathsf{flag}_{\mathsf{smulh}}$ 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 cmpe, cmpa, cmpae, cmpg and cmpge operations. Each comparison result is assigned to the corresponding $\mathsf{flag}_{\widetilde{\mathsf{op}}}$ and does not modify any register. As mentioned before, we set $\langle \mathsf{res}_{\widetilde{\mathsf{op}}} \rangle^{\mathsf{rep}} := \langle \mathsf{r}i \rangle^{\mathsf{rep}}$ to hide the operation.

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

cmpae: This operation checks if $\mathsf{r}j$ is above or equal to a. It follows that $\langle \mathsf{flag}_{\mathsf{cmpae}} \rangle^{\mathsf{rep}} := 1 - \langle \mathsf{flag}_{\mathsf{sub}} \rangle^{\mathsf{rep}} \pmod{2^\ell}$.

cmpa: This operation checks if $\mathsf{r}j$ is above to a, whose comparison result is directly derived from the results of cmpe and cmpae by $\langle \mathsf{flag}_{\mathsf{cmpa}} \rangle^{\mathsf{rep}} := \langle \mathsf{flag}_{\mathsf{cmpae}} \rangle^{\mathsf{rep}} - \langle \mathsf{flag}_{\mathsf{cmpe}} \rangle^{\mathsf{rep}} \pmod{2^\ell}$.

cmpge: This operation checks if signed $\mathsf{r}j_s$ is greater than or equal to a_s. We combine the sign bits of $\mathsf{r}j$, a and $y := \mathsf{r}j - a \pmod{2^\ell}$ to compute $\mathsf{flag}_{\mathsf{cmpge}}$. More specifically, in the same way as $\mathsf{res}_{\mathsf{smulh}}$ computation, the MPC players obliviously detect whether $\mathsf{r}j, a, y \in [0, 2^{\ell-1}]$ by DICF scheme to obtain their sign bits $\llbracket s_r \rrbracket$, $\llbracket s_a \rrbracket$ and $\llbracket s_y \rrbracket$ over the ring $\mathbb{Z}_{2^\ell}$. After that, four parties jointly compute

$$\begin{aligned}\langle \mathsf{flag}_{\mathsf{cmpge}} \rangle :=& (1 - \llbracket s_y \rrbracket) \cdot (1 - \llbracket s_r \rrbracket - \llbracket s_a \rrbracket + 2 \cdot \llbracket s_r \rrbracket \cdot \llbracket s_a \rrbracket) \\ &+ \llbracket s_a \rrbracket \cdot (\llbracket s_r \rrbracket + \llbracket s_a \rrbracket - 2 \cdot \llbracket s_r \rrbracket \cdot \llbracket s_a \rrbracket) \pmod{2^\ell}\end{aligned}$$

and then convert $\langle \mathsf{flag}_{\mathsf{cmpge}} \rangle$ to $\langle \mathsf{flag}_{\mathsf{cmpge}} \rangle^{\mathsf{rep}}$.

cmpg: This operation checks if signed $\mathsf{r}j_s$ is greater than a_s. It follows that $\langle \mathsf{flag}_{\mathsf{cmpg}} \rangle^{\mathsf{rep}} := \langle \mathsf{flag}_{\mathsf{cmpge}} \rangle^{\mathsf{rep}} - \langle \mathsf{flag}_{\mathsf{cmpe}} \rangle^{\mathsf{rep}} \pmod{2^\ell}$.

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 $\langle \mathsf{flag}_{\mathsf{mov}} \rangle^{\mathsf{rep}} := \langle \mathsf{flag}_{\mathsf{cmov}} \rangle^{\mathsf{rep}} := \langle \mathsf{flag} \rangle^{\mathsf{rep}}$.

mov: This operation stores $\langle a \rangle$ to $\langle \mathsf{res}_{\mathsf{mov}} \rangle^{\mathsf{rep}}$. We only need to convert $\langle a \rangle$ into $\langle a \rangle^{\mathsf{rep}}$ and then copy it to $\langle \mathsf{res}_{\mathsf{mov}} \rangle^{\mathsf{rep}}$.

cmov: If $\mathsf{flag} = 1$, this operation stores $\langle a \rangle$ to $\langle \mathsf{res}_{\mathsf{cmov}} \rangle^{\mathsf{rep}}$; otherwise, set $\langle \mathsf{res}_{\mathsf{cmov}} \rangle^{\mathsf{rep}} := \langle \mathsf{r}i \rangle^{\mathsf{rep}}$. The MPC players first jointly compute $\langle \mathsf{res}_{\mathsf{cmov}} \rangle := \langle \mathsf{flag} \rangle \cdot \langle a \rangle + (1 - \langle \mathsf{flag} \rangle) \cdot \langle rj \rangle \pmod{2^\ell}$ and then convert $\langle \mathsf{res}_{\mathsf{cmov}} \rangle$ into $\langle \mathsf{res}_{\mathsf{cmov}} \rangle^{\mathsf{rep}}$.

A.5 Memory Operations

We support $\mathsf{load}, \mathsf{store}$ to randomly access the memory, and 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 $\langle \mathsf{res}_{\mathsf{load}} \rangle^{\mathsf{rep}}$, 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.

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

$\underline{\mathsf{flag}_{\mathsf{load}}}$. Since this operation do not modify the condition flag, four MPC players directly set $\langle \mathsf{flag}_{\mathsf{load}} \rangle^{\mathsf{rep}} := \langle \mathsf{flag} \rangle^{\mathsf{rep}}$.

store: This operation stores $\langle \mathsf{r}i \rangle$ 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 $\mathsf{res}_{\mathsf{load}}$ computation, and utilize $[\![f_{(\delta_1 - \psi_1),1}(x)]\!]$ generated in the instruction fetching phase to compute

$$\begin{aligned} \langle y \rangle := & [\![f_{(\delta_1 - \psi_1),1}(\mathsf{store})]\!] \cdot \langle ri \rangle \\ & + (1 - [\![f_{(\delta_1 - \psi_1),1}(\mathsf{store})]\!]) \cdot \langle M_a \rangle \pmod{2^\ell}. \end{aligned}$$

Namely, y is set to M_a if $\mathsf{op} = \mathsf{store}$ and to $\mathsf{r}i$ 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).

$\underline{\mathsf{res}_{\mathsf{store}} \& \mathsf{flag}_{\mathsf{store}}}$. This operation does not modify any register or the condition flag so that we set $\langle \mathsf{res}_{\mathsf{store}} \rangle^{\mathsf{rep}} := \langle ri \rangle^{\mathsf{rep}}$ and $\langle \mathsf{flag}_{\mathsf{store}} \rangle^{\mathsf{rep}} := \langle \mathsf{flag} \rangle^{\mathsf{rep}}$.

read: This operation reads the next word (i.e. $\langle \mathsf{rc}_a \rangle$-th word) on the a-th tape, then stores it into $\mathsf{res}_{\mathsf{read}}$ and sets $\mathsf{flag} := 0$; if the a-th tape does not have remaining words, it stores 0 into $\mathsf{res}_{\mathsf{read}}$ and sets $\mathsf{flag} := 1$. We use our o-read protocol to obliviously fetch the next word in tape_0 and tape_1. After that, we select the right word according to a. Since the num_a-th word of tape_a is 0, we keep $\mathsf{rc}_a = \mathsf{num}_a$ 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 $\mathsf{tape}_0, \mathsf{tape}_1$ have input words and to 0 otherwise. Once the input words in tape_0 (or tape_1) are consumed, t_0 (or t_1) will set to 0. And then t_a is the value of $\mathsf{flag}_{\mathsf{read}}$. More specifically, it works as follows.

$\underline{\mathsf{res}_{\mathsf{read}}}$. First of all, the MPC players invoke o-read protocol twice to obliviously fetch $\langle \mathsf{rc}_0 \rangle$-th word $\langle x_0 \rangle$ in tape_0 and $\langle \mathsf{rc}_1 \rangle$-th word $\langle x_1 \rangle$ in tape_1. Meanwhile, P_1 generates two pairs of DPF keys for the point functions $f_{r_0,1}(x), f_{r_1,1}(x) : \mathbb{Z}_{2^\ell} \to \mathbb{Z}_{2^\ell}$, where $r_0, r_1 \leftarrow \mathbb{Z}_{2^\ell}$ are randomly picked, and distributes these keys to P_3 and P_4. The MPC players open $\langle \delta_0 \rangle := \langle \mathsf{rc}_0 \rangle - \langle \mathsf{num}_0 \rangle - 1 + r_0 \pmod{2^\ell}$ and $\langle \delta_1 \rangle := \langle \mathsf{rc}_1 \rangle - \langle \mathsf{num}_1 \rangle - 1 + r_1 \pmod{2^\ell}$ to P_3 and P_4. After that, parties can jointly compute

$$\begin{aligned} \langle k \rangle &:= [\![f_{(\delta_1 - \psi_1),1}(\mathsf{read})]\!] \cdot \langle a \rangle \pmod{2^\ell}, \\ \langle t_0' \rangle &:= \langle t_0 \rangle - [\![f_{r_0,1}(\delta_0)]\!] \pmod{2^\ell}, \\ \langle t_1' \rangle &:= \langle t_1 \rangle - [\![f_{r_1,1}(\delta_1)]\!] \pmod{2^\ell} \end{aligned}$$

by local DPF key evaluation. If $\tilde{\mathsf{op}} = \mathsf{read}$, a is set to 0 or 1 and k takes the value of a; if $\tilde{\mathsf{op}} \neq \mathsf{read}$, k takes the value of 0. And t_0' is set to 1 if tape_0 has remaining words (i.e. $\mathsf{rc}_0 < \mathsf{num}_0$) and set to 0 otherwise; similarly for t_1'. Parties then calculate the operation result and increases in read counters as

$$\begin{aligned} \langle \mathsf{res}_{\mathsf{read}} \rangle &:= (1 - \langle k \rangle) \cdot \langle x_0 \rangle + \langle k \rangle \cdot \langle x_1 \rangle \pmod{2^\ell}, \\ \langle \delta_0 \rangle &:= (1 - \langle k \rangle) \cdot [\![f_{(\delta_1 - \psi_1),1}(\mathsf{read})]\!] \pmod{2^\ell}, \\ \langle \delta_1 \rangle &:= \langle k \rangle \cdot [\![f_{(\delta_1 - \psi_1),1}(\mathsf{read})]\!] \pmod{2^\ell}, \end{aligned}$$

Finally, parties convert $\langle \mathsf{res}_{\mathsf{read}} \rangle$ into $\langle \mathsf{res}_{\mathsf{read}} \rangle^{\mathsf{rep}}$.

$\underline{\mathsf{flag}_{\mathsf{read}}}$. With the intermediate result in $\mathsf{res}_{\mathsf{read}}$ computation, parties jointly compute $\langle \mathsf{flag}_{\mathsf{read}} \rangle := (1 - \langle k \rangle) \cdot \langle t_0' \rangle + \langle k \rangle \cdot \langle t_1' \rangle \pmod{2^\ell}$, and then convert $\langle \mathsf{flag}_{\mathsf{read}} \rangle$ into $\langle \mathsf{flag}_{\mathsf{read}} \rangle^{\mathsf{rep}}$. In addition, to obliviously update t_0, t_1 and the read counters accordingly, parties need to compute

$$\langle t_0 \rangle := (1 - [\![f_{(\delta_1-\psi_1),1}(\mathsf{read})]\!])\langle t_0 \rangle + [\![f_{(\delta_1-\psi_1),1}(\mathsf{read})]\!]\langle t_0' \rangle,$$
$$\langle t_1 \rangle := (1 - [\![f_{(\delta_1-\psi_1),1}(\mathsf{read})]\!])\langle t_1 \rangle + [\![f_{(\delta_1-\psi_1),1}(\mathsf{read})]\!]\langle t_1' \rangle,$$
$$\langle \mathsf{rc}_0 \rangle := \langle \mathsf{rc}_0 \rangle + \langle \delta_0 \rangle \langle \mathsf{flag}_{\mathsf{read}} \rangle, \ \ \langle \mathsf{rc}_1 \rangle := \langle \mathsf{rc}_1 \rangle + \langle \delta_1 \rangle \langle \mathsf{flag}_{\mathsf{read}} \rangle$$

over the ring $\mathbb{Z}_{2^\ell}$. Namely, in the case of $\tilde{\mathsf{op}} = \mathsf{read}$, rc_a will be increased if $\mathsf{rc}_a < \mathsf{num}_a$ and keep $\mathsf{rc}_a = \mathsf{num}_a$ otherwise.

A.6 Jump Operations

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

jmp: This instruction is a standard jump operation that stores $\langle a \rangle$ in $\langle \mathsf{pc} \rangle$. The MPC players only need to compute

$$\langle \mathsf{pc} \rangle := [\![f_{(\delta_1-\psi_1),1}(\mathsf{jmp})]\!] \cdot [\![a]\!] + (1 - [\![f_{(\delta_1-\psi_1),1}(\mathsf{jmp})]\!]) \cdot \langle \mathsf{pc} \rangle$$

over the ring $\mathbb{Z}_{2^\ell}$.

cjmp: This instruction stores $\langle a \rangle$ in $\langle \mathsf{pc} \rangle$ if $\mathsf{flag} = 1$. The MPC players should jointly compute

$$\begin{aligned}\langle \mathsf{pc} \rangle :=& [\![f_{(\delta_1-\psi_1),1}(\mathsf{cjmp})]\!] \cdot ([\![a]\!] \cdot \langle \mathsf{flag} \rangle + \langle \mathsf{pc} \rangle \cdot (1 - \langle \mathsf{flag} \rangle)) \\ &+ (1 - [\![f_{(\delta_1-\psi_1),1}(\mathsf{cjmp})]\!]) \cdot \langle \mathsf{pc} \rangle \pmod{2^\ell}.\end{aligned}$$

cnjmp: This operation stores $\langle a \rangle$ in $\langle \mathsf{pc} \rangle$ if $\mathsf{flag} = 0$. Similar to cjmp, it requires four parties to jointly compute

$$\begin{aligned}\langle \mathsf{pc} \rangle :=& [\![f_{(\delta_1-\psi_1),1}(\mathsf{cnjmp})]\!] \cdot ([\![a]\!] \cdot (1 - \langle \mathsf{flag} \rangle) + \langle \mathsf{pc} \rangle \cdot \langle \mathsf{flag} \rangle) \\ &+ (1 - [\![f_{(\delta_1-\psi_1),1}(\mathsf{cnjmp})]\!]) \cdot \langle \mathsf{pc} \rangle \pmod{2^\ell}.\end{aligned}$$

A.7 Answer Operation

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 $f_{(\delta_1-\psi_1),1}(\mathsf{ans})$ between each other. If $f_{(\delta_1-\psi_1),1}(\mathsf{ans}) = 1$, 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

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

Case 1: P_1 (or P_2) 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 $\mathcal{F}^{n,\ell}_{\mathsf{dGen}}[\mathsf{DPF.Gen}]$ as well as honest parties P_2, P_3, P_4 In addition, $\mathcal{S}$ simulates the following interactions with $\mathcal{A}$.

- Upon initialization, $\mathcal{S}$ acts as the honest party P_3 to do:
  - Set $\varphi_1, \varphi_1^{(2)} \leftarrow \mathbb{Z}_n$, $\varphi_1^{(1)} := \varphi_1 - \varphi_1^{(2)}$, $r_1^{(2)} \leftarrow \mathbb{Z}_n$;
  - Set $\mathcal{K}^{(1)}_{\varphi_1}, \mathcal{K}^{(2)}_{\varphi_1} \leftarrow \mathsf{DPF.Gen}(1^\lambda, f_{\varphi_1,1})$;
  - Send $(\mathsf{sid}, \mathcal{K}^{(1)}_{\varphi_1})$ to P_1, $(\mathsf{sid}, \mathcal{K}^{(2)}_{\varphi_1})$ to P_2;
  - Send $(\textsc{KeyGen}, \mathsf{sid}, r_1^{(2)}, 1)$ to $\mathcal{F}^{n,\ell}_{\mathsf{dGen}}[\mathsf{DPF.Gen}]$;

  - Get $(\mathsf{sid}, \mathcal{K}_1^{(2)})$ from $\mathcal{F}_{\mathsf{dGen}}^{n,\ell}[\mathsf{DPF.Gen}]$;
  - Send $(\mathsf{sid}, \mathcal{K}_1^{(2)})$ to P_4;
- Upon initialization, the simulator $\mathcal{S}$ acts as the honest party $P_j, j \in \{2,4\}$ to do:
  - Set $r_2^{(j/2)} \leftarrow \mathbb{Z}_n$, $v^{(j/2)} \leftarrow \mathbb{Z}_{2^\ell}$;
  - Send $(\textsc{KeyGen}, \mathsf{sid}, r_2^{(j/2)}, v^{(j/2)})$ to $\mathcal{F}_{\mathsf{dGen}}^{n,\ell}[\mathsf{DPF.Gen}]$, then gets $(\mathsf{sid}, \mathcal{K}_v^{(j/2)})$ and send $(\mathsf{sid}, \mathcal{K}_v^{(j/2)})$ to P_{j-1};
- The simulator $\mathcal{S}$ does:
  - Pick random $w_{1,1}, w_{1,2} \leftarrow \mathbb{Z}_n$;
  - Pick random $w'_{k,q} \leftarrow \mathbb{Z}_n$, for $k \in [2]$ and $q \in [4]$;
  - Pick random $w'_{3,q} \leftarrow \mathbb{Z}_{2^\ell}$, for $q \in [4]$;
- Upon receiving $(\textsc{ORead}, \mathsf{sid}, \mathsf{ssid}, P_j)$ for an honest party P_j, $j \in \{1,2\}$ from the external $\mathcal{F}_{\mathsf{ram}}^{n,\ell}$, the simulator $\mathcal{S}$ does:
  - Set $\delta_{r,1}^{(j)} := w_{1,j}$ and $\delta_{r,2}^{(j)} := 0$;
  - Send $(\mathsf{sid}, \mathsf{ssid}, \delta_{r,1}^{(j)})$ to P_{3-j}, $(\mathsf{sid}, \mathsf{ssid}, \delta_{r,2}^{(j)})$ to P_3 on behave of P_j;
- Upon receiving $(\textsc{ORead}, \mathsf{sid}, \mathsf{ssid}, P_j)$ for an honest party P_j, $j \in \{3,4\}$ from the external $\mathcal{F}_{\mathsf{ram}}^{n,\ell}$, the simulator $\mathcal{S}$ does:
  - Set $\delta_{r,1}^{(j)} := -\varphi_1^{(j-2)} - w_{1,j-2}$ and $\delta_{r,2}^{(j)} := 0$;
  - Send $(\mathsf{sid}, \mathsf{ssid}, \delta_{r,1}^{(j)})$ to P_1 and P_2, $(\mathsf{sid}, \mathsf{ssid}, \delta_2^{(j)})$ to P_{7-j} on behave of P_j;
- Upon receiving $(\mathsf{sid}, \mathsf{ssid}, \delta_{r,1}^{(1)})$ from the corrupted P_1 to P_2 and $(\mathsf{sid}, \mathsf{ssid}, \delta_{r,2}^{(1)})$ from the corrupted P_1 to P_3, P_4, the simulator $\mathcal{S}$ does:
  - Extract $i^{(1)} := \delta_{r,1}^{(1)} - \mathsf{PRF}_{\eta_1}^{\mathbb{Z}_n}(\mathsf{sid}, 1) \pmod{n}$;
  - Send $(\textsc{ORead}, \mathsf{sid}, i^{(1)})$ to the external $\mathcal{F}_{\mathsf{ram}}^{n,\ell}$;
  - Compute $\delta_{r,1} := \delta_{r,1}^{(1)} + \delta_{r,1}^{(2)} + \delta_{r,1}^{(3)} + \delta_{r,1}^{(4)} \pmod{n}$;
  - Set $(\beta_{\varphi_1,k}^{(1)})_{k\in\mathbb{Z}_n} \leftarrow \mathsf{DPF.EvalAll}(1, \mathcal{K}_{\varphi_1}^{(1)})$;
  - Set $\widetilde{\beta}_{\varphi_1,k}^{(1)} := \beta_{\varphi_1,k+\delta_{r,1} \pmod{n}}^{(1)}$, for $k \in \mathbb{Z}_n$;
  - Set $\zeta_1 \leftarrow \mathsf{PRF}_{\eta_1}^{\mathbb{Z}_{2^\ell}}(\mathsf{sid})$, $\zeta_3 \leftarrow \mathsf{PRF}_{\eta_2}^{\mathbb{Z}_{2^\ell}}(\mathsf{sid})$;
  - Compute $o^{(1)} := \sum_{k=0}^{n-1}(x_k^{(1)} \cdot \widetilde{\beta}_{\varphi_1,k}^{(1)}) + \zeta_1 - \zeta_3 \pmod{2^\ell}$.
  - Send $(\textsc{Rand}, \mathsf{sid}, \mathsf{ssid}, o^{(1)})$ to the external $\mathcal{F}_{\mathsf{ram}}^{n,\ell}$;
- Upon receiving $(\textsc{OWrite}, \mathsf{sid}, \mathsf{ssid}, P_j)$ for an honest party P_j, $j \in \{1,3\}$ from the external $\mathcal{F}_{\mathsf{ram}}^{n,\ell}$, the simulator $\mathcal{S}$ does:
  - Set $\delta_{w,1}^{(j)} := -r_1^{(\lceil j/2 \rceil)} + (j-2) \cdot w'_{1,1} + w'_{1,j+1}$;
  - Set $\delta_{w,2}^{(j)} := (j-2) \cdot w'_{2,1} + w'_{2,j+1}$;
  - Set $\Delta v^{(j)} := (j-2) \cdot w'_{3,1} + w'_{3,j+1}$;
  - Send $(\delta_{w,1}^{(j)}, \delta_{w,2}^{(j)}, \Delta v^{(j)})$ to $P_q, q \in [4]/j$ on behave of P_j;
- Upon receiving $(\textsc{OWrite}, \mathsf{sid}, \mathsf{ssid}, P_j)$ for an honest party P_j, $j \in \{2,4\}$ from the external $\mathcal{F}_{\mathsf{ram}}^{n,\ell}$, the simulator $\mathcal{S}$ does:
  - $\delta_{w,1}^{(j)} := (j-3) \cdot w'_{1,3} - w'_{1,j}$;
  - $\delta_{w,2}^{(j)} := -r_2^{(j/2)} + (j-3) \cdot w'_{2,3} - w'_{2,j}$;
  - $\Delta v^{(j)} := -v^{(j/2)} + (j-3) \cdot w'_{3,3} - w'_{3,j}$;
  - Send $(\delta_{w,1}^{(j)}, \delta_{w,2}^{(j)}, \Delta v^{(j)})$ to $P_q, q \in [4]/j$ on behave of P_j;

Indistinguishability. We assume that the parties $P_1, \ldots, P_4$ communicate with each other via the secure channel functionality $\mathcal{F}_{\mathsf{sc}}$ (omitted in the protocol description for simplicity). The indistinguishability is proven through a series of hybrid worlds $\mathcal{H}_0, \ldots, \mathcal{H}_3$.

Hybrid $\mathcal{H}_0$: It is the real protocol execution $\mathsf{Exec}_{\Pi_{\mathsf{ram}}^{n,\ell}, \mathcal{A}, \mathcal{Z}}$.

Hybrid $\mathcal{H}_1$: $\mathcal{H}_1$ is the same as $\mathcal{H}_0$ except that in $\mathcal{H}_1$, $\{w_{1,q}\}_{q\in[2]}$, $\varphi_1^{(2)}$ and $\{w'_{k,q}\}_{k\in[2],q\in[4]}$ are picked uniformly random from $\mathbb{Z}_n$ instead of calculating from $\mathsf{PRF}^{\mathbb{Z}_n}$; $\{w'_{3,q}\}_{q\in[4]}$ is picked uniformly random from $\mathbb{Z}_{2^\ell}$ instead of calculating from $\mathsf{PRF}^{\mathbb{Z}_{2^\ell}}$.

Claim 1. *If* $\mathsf{PRF}^{\mathbb{Z}_n} : \{0,1\}^\lambda \times \{0,1\}^{\mathsf{in}} \mapsto \mathbb{Z}_n$ *is a secure pseudorandom function with adversarial advantage* $\mathsf{Adv}_{\mathsf{PRF}^{\mathbb{Z}_n}}(1^\lambda, \mathcal{A})$ *and* $\mathsf{PRF}^{\mathbb{Z}_{2^\ell}} : \{0,1\}^\lambda \times \{0,1\}^{\mathsf{in}} \mapsto \mathbb{Z}_{2^\ell}$ *is a secure pseudorandom function* *with adversarial advantage* $\mathsf{Adv}_{\mathsf{PRF}^{\mathbb{Z}_{2^\ell}}}(1^\lambda, \mathcal{A})$*, then* $\mathcal{H}_1$ *and* $\mathcal{H}_0$ *are indistinguishable with advantage* $\epsilon_1 := 11 \cdot \mathsf{Adv}_{\mathsf{PRF}^{\mathbb{Z}_n}}(1^\lambda, \mathcal{A}) + 4 \cdot \mathsf{Adv}_{\mathsf{PRF}^{\mathbb{Z}_{2^\ell}}}(1^\lambda, \mathcal{A})$*.*

Proof. We have changed 11 $\mathsf{PRF}^{\mathbb{Z}_n}$ outputs and 4 $\mathsf{PRF}^{\mathbb{Z}_{2^\ell}}$ outputs to uniformly random strings; therefore, the overall advantage is $11 \cdot \mathsf{Adv}_{\mathsf{PRF}^{\mathbb{Z}_n}}(1^\lambda, \mathcal{A}) + 4 \cdot \mathsf{Adv}_{\mathsf{PRF}^{\mathbb{Z}_{2^\ell}}}(1^\lambda, \mathcal{A})$ 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 $(\textsc{ORead}, \mathsf{sid}, \mathsf{ssid}, P_j)$ for $j \in \{1, 2\}$:
  - Set $\delta_{r,1}^{(j)} := w_1^{(j)}$ and $\delta_{r,2}^{(j)} := 0$.
- Upon receiving $(\textsc{ORead}, \mathsf{sid}, \mathsf{ssid}, P_j)$ for $j \in \{3, 4\}$:
  - Set $\delta_{r,1}^{(j)} := -\varphi_1^{(j-2)} - w_1^{(j)}$ and $\delta_{r,2}^{(j)} := 0$.
- Upon receiving $(\textsc{OWrite}, \mathsf{sid}, \mathsf{ssid}, P_j)$ for $j \in \{1, 3\}$:
  - Set $\delta_{w,1}^{(j)} := -r_1^{(\lceil j/2 \rceil)} + (j-2) \cdot w'_{1,1} + w'_{1,j+1}$;
  - Set $\delta_{w,2}^{(j)} := (j-2) \cdot w'_{2,1} + w'_{2,j+1}$;
- Upon receiving $(\textsc{OWrite}, \mathsf{sid}, \mathsf{ssid}, P_j)$ for $j \in \{2, 4\}$:
  - Set $\delta_{w,1}^{(j)} := (j-3) \cdot w'_{1,3} - w'_{1,j}$, $\delta_{w,2}^{(j)} := (j-3) \cdot w'_{2,3} - w'_{2,j}$;

instead of

- Upon receiving $(\textsc{ORead}, \mathsf{sid}, \mathsf{ssid}, P_j)$ for $j \in \{1, 2\}$:
  - Set $\delta_{r,1}^{(j)} := i^{(j)} + w_{1,j}$ and $\delta_{r,2}^{(j)} := i^{(j)} - \varphi_2^{(j)} + w_{2,j}$.
- Upon receiving $(\textsc{ORead}, \mathsf{sid}, \mathsf{ssid}, P_j)$ for $j \in \{3, 4\}$:
  - Set $\delta_{r,1}^{(j)} := i^{(j)} - \varphi_1^{(j-2)} - w_{1,j-2}$ and $\delta_2^{(j)} := i^{(j)} - w_{2,j-2}$.
- Upon receiving $(\textsc{OWrite}, \mathsf{sid}, \mathsf{ssid}, P_j)$ for $j \in \{1, 3\}$:
  - Set $\delta_{w,1}^{(j)} := i^{(j)} - r_1^{(\lceil j/2 \rceil)} + (j-2) \cdot w'_{1,1} + w'_{1,j+1}$;
  - Set $\delta_{w,2}^{(j)} := i^{(j)} + (j-2) \cdot w'_{2,1} + w'_{2,j+1}$;
- Upon receiving $(\textsc{OWrite}, \mathsf{sid}, \mathsf{ssid}, P_j)$ for $j \in \{2, 4\}$:
  - Set $\delta_{w,1}^{(j)} := i^{(j)} + (j-3) \cdot w'_{1,3} - w'_{1,j}$;
  - Set $\delta_{w,2}^{(j)} := i^{(j)} - r_2^{(j/2)} + (j-3) \cdot w'_{2,3} - w'_{2,j}$;

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

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

- $\delta_{r,1}^{(1)} := i^{(1)} + w_{1,1}$, $\delta_{r,1}^{(2)} := i^{(2)} + w_{1,2}$;
- $\delta_{r,1}^{(3)} := i^{(1)} - \varphi_1^{(1)} - w_{1,1}$, $\delta_{r,1}^{(4)} := i^{(1)} - \varphi_1^{(2)} - w_{1,2}$;
- $\delta_{w,1}^{(1)} := i^{(j)} - r_1^{(1)} - w'_{1,1} + w'_{1,2}$, $\delta_{w,1}^{(2)} := i^{(j)} - w'_{1,3} - w'_{1,2}$;
- $\delta_{w,1}^{(3)} := i^{(j)} - r_1^{(2)} + w'_{1,1} + w'_{1,4}$, $\delta_{w,1}^{(4)} := i^{(j)} + w'_{1,3} - w'_{1,4}$;
- $\delta_{w,2}^{(1)} := i^{(j)} - w'_{2,1} + w'_{2,2}$, $\delta_{w,2}^{(2)} := i^{(j)} - r_2^{(1)} - w'_{2,3} - w'_{2,2}$;
- $\delta_{w,2}^{(3)} := i^{(j)} + w'_{2,1} + w'_{2,4}$, $\delta_{w,2}^{(4)} := i^{(j)} - r_2^{(2)} + w'_{2,3} - w'_{2,4}$;

It is straightforward that the distribution of $\{\delta_{r,1}^{(j)}\}_{j \in [4]}$ are uniformly random under the condition $\delta_{r,1} := \sum_{k=1}^{4} \delta_{r,1}^{(k)} = i - \varphi_1$ where φ_1 is used to generate the DPF keys $(\mathcal{K}_{\varphi_1}^{(1)}, \mathcal{K}_{\varphi_1}^{(2)})$; similarly for $\{\delta_{w,1}^{(j)}\}_{j \in [4]}$ and $\{\delta_{w,2}^{(j)}\}_{j \in [4]}$. 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 $\mathcal{A}$ who can distinguish the view of $\mathcal{H}_2$ from the view of $\mathcal{H}_1$ then we can construct an adversary $\mathcal{B}$ who uses $\mathcal{A}$ in a blackbox fashion can break at least one of the three $\mathsf{DPF}^{\mathbb{Z}_n, \mathbb{Z}_{2^\ell}} := (\mathsf{Gen}, \mathsf{Eval})$ with the same advantage. Therefore, $\mathcal{H}_2$ and $\mathcal{H}_1$ are indistinguishable with adversarial advantage $\epsilon_2 := 3 \cdot \mathsf{Adv}_{\mathsf{DPF}^{\mathbb{Z}_n, \mathbb{Z}_{2^\ell}}}(1^\lambda, \mathcal{A})$. □

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

- Upon receiving $(\textsc{OWrite}, \mathsf{sid}, \mathsf{ssid}, P_j)$ for $j \in \{1, 3\}$:
  - Set $\Delta v^{(j)} := (j-2) \cdot w'_{3,1} + w'_{3,j+1}$;

- Upon receiving $(\textsc{OWrite}, \mathsf{sid}, \mathsf{ssid}, P_j)$ 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 $(\textsc{OWrite}, \mathsf{sid}, \mathsf{ssid}, P_j)$ for $j \in \{1, 3\}$:
  - Set $\Delta v^{(j)} := y_i^{(j)} - x^{(j)} + (j-2) \cdot w'_{3,1} + w'_{3,j+1}$;
- Upon receiving $(\textsc{OWrite}, \mathsf{sid}, \mathsf{ssid}, P_j)$ for $j \in \{2, 4\}$:
  - Set $\Delta v^{(j)} := y_i^{(j)} - x^{(j)} - v^{(j/2)} + (j-3) \cdot w'_{3,3} - w'_{3,j}$;

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

Proof. Since $\{w'_{3,j}\}_{j\in[4]}$, $\{y_i^{(j)}\}_{j\in[4]}$ and $\{x^{(j)}\}_{j\in[4]}$ are uniformly random in $\mathbb{Z}_{2^\ell}$, the distribution of $\{\Delta v^{(j)}\}_{j\in[4]}$ 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 $\mathsf{Exec}_{\mathcal{F}^{n,\ell}_{\mathsf{read}}, \mathcal{S}, \mathcal{Z}}$. Therefore, the overall distinguishing advantage is

$$11 \cdot \mathsf{Adv}_{\mathsf{PRF}^{\mathbb{Z}_n}}(1^\lambda, \mathcal{A}) + 4 \cdot \mathsf{Adv}_{\mathsf{PRF}^{\mathbb{Z}_{2^\ell}}}(1^\lambda, \mathcal{A}) + 3 \cdot \mathsf{Adv}_{\mathsf{DPF}^{\mathbb{Z}_n, \mathbb{Z}_{2^\ell}}}(1^\lambda, \mathcal{A}) \ .$$

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 $\mathcal{F}^{n,\ell}_{\mathsf{dGen}}[\mathsf{DPF.Gen}]$ as well as honest parties P_1, P_2, P_4 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 $\mathcal{K}^{(1)}_{\varphi_2}, \mathcal{K}^{(2)}_{\varphi_2} \leftarrow \mathsf{DPF.Gen}(1^\lambda, f_{\varphi_2,1})$;
  - Send $(\mathsf{sid}, \mathcal{K}^{(1)}_{\varphi_2})$ to P_3, $(\mathsf{sid}, \mathcal{K}^{(2)}_{\varphi_2})$ to P_4;
  - Send $(\textsc{KeyGen}, \mathsf{sid}, r_1^{(1)}, 0)$ to $\mathcal{F}^{n,\ell}_{\mathsf{dGen}}[\mathsf{DPF.Gen}]$ then get $(\mathsf{sid}, \mathcal{K}^{(1)}_1)$ and send $(\mathsf{sid}, \mathcal{K}^{(1)}_1)$ to P_2;
- Upon initialization, the simulator $\mathcal{S}$ acts as the honest party $P_j, j \in \{2, 4\}$ to do:
  - Set $r_2^{(j/2)} \leftarrow \mathbb{Z}_n$, $v^{(j/2)} \leftarrow \mathbb{Z}_{2^\ell}$;
  - Send $(\textsc{KeyGen}, \mathsf{sid}, r_2^{(j/2)}, v^{(j/2)})$ to $\mathcal{F}^{n,\ell}_{\mathsf{dGen}}[\mathsf{DPF.Gen}]$, then get $(\mathsf{sid}, \mathcal{K}^{(j/2)}_v)$ and send $(\mathsf{sid}, \mathcal{K}^{(j/2)}_v)$ to P_{j-1};
- The simulator $\mathcal{S}$ does:
  - Pick random $w_{2,1}, w_{2,2} \leftarrow \mathbb{Z}_n$;
  - Pick random $w'_{k,q} \leftarrow \mathbb{Z}_n$, for $k \in [2]$ and $q \in [4]$;
  - Pick random $w'_{3,q} \leftarrow \mathbb{Z}_{2^\ell}$, for $q \in [4]$;
- Upon receiving $(\textsc{ORead}, \mathsf{sid}, \mathsf{ssid}, P_j)$ for an honest party P_j, $j \in \{1, 2\}$ from the external $\mathcal{F}^{n,\ell}_{\mathsf{ram}}$, the simulator $\mathcal{S}$ does:
  - Set $\delta^{(j)}_{r,1} := 0$ and $\delta^{(j)}_{r,2} := w_{2,j} - \varphi_2^{(j)}$;
  - Send $(\mathsf{sid}, \mathsf{ssid}, \delta^{(j)}_{r,1})$ to P_{3-j}, $(\mathsf{sid}, \mathsf{ssid}, \delta^{(j)}_{r,2})$ to P_3 on behave of P_j;
- Upon receiving $(\textsc{ORead}, \mathsf{sid}, \mathsf{ssid}, P_j)$ for an honest party P_j, $j \in \{3, 4\}$ from the external $\mathcal{F}^{n,\ell}_{\mathsf{ram}}$, the simulator $\mathcal{S}$ does:
  - Set $\delta^{(j)}_{r,1} := 0$ and $\delta^{(j)}_{r,2} := -w_{2,j-2}$;
  - Send $(\mathsf{sid}, \mathsf{ssid}, \delta^{(j)}_{r,1})$ to P_1 and P_2, $(\mathsf{sid}, \mathsf{ssid}, \delta^{(j)}_{r,2})$ to P_{7-j} on behave of P_j;
- Upon receiving $(\mathsf{sid}, \mathsf{ssid}, \delta^{(3)}_{r,2})$ from the corrupted P_3 to P_4 and $(\mathsf{sid}, \mathsf{ssid}, \delta^{(3)}_{r,1})$ from the corrupted P_3 to P_1, P_2, the simulator $\mathcal{S}$ does:
  - Extract $i^{(3)} := \delta^{(3)}_{r,2} - \mathsf{PRF}^{\mathbb{Z}_n}_{\eta_1}(\mathsf{sid}, 2) \pmod{n}$;
  - Send $(\textsc{ORead}, \mathsf{sid}, i^{(3)})$ to the external $\mathcal{F}^{n,\ell}_{\mathsf{ram}}$;
  - Compute $\delta_{r,2} := \delta^{(1)}_{r,2} + \delta^{(2)}_{r,2} + \delta^{(3)}_{r,2} + \delta^{(4)}_{r,2} \pmod{n}$;
  - Set $(\beta^{(1)}_{\varphi_2,k})_{k\in\mathbb{Z}_n} \leftarrow \mathsf{DPF.EvalAll}(1, \mathcal{K}^{(1)}_{\varphi_2})$;
  - Set $\widetilde{\beta}^{(1)}_{\varphi_2,k} := \beta^{(1)}_{\varphi_2,k+\delta_{r,2} \pmod{n}}$, for $k \in \mathbb{Z}_n$;
  - Set $\zeta_1 \leftarrow \mathsf{PRF}^{\mathbb{Z}_{2^\ell}}_{\eta_1}(\mathsf{sid})$, $\zeta_4 \leftarrow \mathsf{PRF}^{\mathbb{Z}_{2^\ell}}_{\eta_4}(\mathsf{sid})$;
  - Compute $o^{(3)} := \sum_{k=0}^{n-1}(x_k^{(1)} \cdot \widetilde{\beta}^{(1)}_{\varphi_1,k}) - \zeta_1 - \zeta_4 \pmod{2^\ell}$.
  - Send $(\textsc{Rand}, \mathsf{sid}, \mathsf{ssid}, o^{(3)})$ to the external $\mathcal{F}^{n,\ell}_{\mathsf{ram}}$;

- Upon receiving $(\text{OWRITE}, \mathsf{sid}, \mathsf{ssid}, P_j)$ for an honest party P_j, $j \in \{1,3\}$ from the external $\mathcal{F}^{n,\ell}_{\mathsf{ram}}$, the simulator $\mathcal{S}$ does:
  - Set $\delta^{(j)}_{w,1} := -r_1^{(\lceil j/2 \rceil)} + (j-2) \cdot w'_{1,1} + w'_{1,j+1}$;
  - Set $\delta^{(j)}_{w,2} := (j-2) \cdot w'_{2,1} + w'_{2,j+1}$;
  - Set $\Delta v^{(j)} := (j-2) \cdot w'_{3,1} + w'_{3,j+1}$;
  - Send $(\mathsf{sid}, \mathsf{ssid}, \delta^{(j)}_{w,1}, \delta^{(j)}_{w,2}, \Delta v^{(j)})$ to $P_q, q \in [4]/j$ on behave of P_j;
- Upon receiving $(\text{OWRITE}, \mathsf{sid}, \mathsf{ssid}, P_j)$ for an honest party P_j, $j \in \{2,4\}$ from the external $\mathcal{F}^{n,\ell}_{\mathsf{ram}}$, the simulator $\mathcal{S}$ does:
  - $\delta^{(j)}_{w,1} := (j-3) \cdot w'_{1,3} - w'_{1,j}$;
  - $\delta^{(j)}_{w,2} := -r_2^{(j/2)} + (j-3) \cdot w'_{2,3} - w'_{2,j}$;
  - $\Delta v^{(j)} := -v^{(j/2)} + (j-3) \cdot w'_{3,3} - w'_{3,j}$;
  - Send $(\mathsf{sid}, \mathsf{ssid}, \delta^{(j)}_{w,1}, \delta^{(j)}_{w,2}, \Delta v^{(j)})$ to $P_q, q \in [4]/j$ on behave of P_j;

Indistinguishability. We assume that the parties $P_1, \ldots, P_4$ communicate with each other via the secure channel functionality $\mathcal{F}_{\mathsf{sc}}$ (omitted in the protocol description for simplicity). The indistinguishability is proven through a series of hybrid worlds $\mathcal{H}_0, \ldots, \mathcal{H}_3$.

Hybrid $\mathcal{H}_0$: It is the real protocol execution $\mathsf{Exec}_{\Pi^{n,\ell}_{\mathsf{ram}},\mathcal{A},\mathcal{Z}}$.

Hybrid $\mathcal{H}_1$: $\mathcal{H}_1$ is the same as $\mathcal{H}_0$ except that in $\mathcal{H}_1$, $\{w_{2,q}\}_{q\in[2]}$, $\varphi_1^{(2)}$ and $\{w'_{k,q}\}_{k\in[2],q\in[4]}$ are picked uniformly random from $\mathbb{Z}_n$ instead of calculating from $\mathsf{PRF}^{\mathbb{Z}_n}$; $\{w'_{3,q}\}_{q\in[4]}$ is picked uniformly random from $\mathbb{Z}_{2^\ell}$ instead of calculating from $\mathsf{PRF}^{\mathbb{Z}_{2^\ell}}$.

Claim 4. *If* $\mathsf{PRF}^{\mathbb{Z}_n} : \{0,1\}^\lambda \times \{0,1\}^{\mathsf{in}} \mapsto \mathbb{Z}_n$ *is a secure pseudorandom function with adversarial advantage* $\mathsf{Adv}_{\mathsf{PRF}^{\mathbb{Z}_n}}(1^\lambda, \mathcal{A})$ *and* $\mathsf{PRF}^{\mathbb{Z}_{2^\ell}} : \{0,1\}^\lambda \times \{0,1\}^{\mathsf{in}} \mapsto \mathbb{Z}_{2^\ell}$ *is a secure pseudorandom function with adversarial advantage* $\mathsf{Adv}_{\mathsf{PRF}^{\mathbb{Z}_{2^\ell}}}(1^\lambda, \mathcal{A})$*, then* $\mathcal{H}_1$ *and* $\mathcal{H}_0$ *are indistinguishable with advantage* $\epsilon_1 := 11 \cdot \mathsf{Adv}_{\mathsf{PRF}^{\mathbb{Z}_n}}(1^\lambda, \mathcal{A}) + 4 \cdot \mathsf{Adv}_{\mathsf{PRF}^{\mathbb{Z}_{2^\ell}}}(1^\lambda, \mathcal{A})$.

Proof. We have changed 11 $\mathsf{PRF}^{\mathbb{Z}_n}$ outputs and 4 $\mathsf{PRF}^{\mathbb{Z}_{2^\ell}}$ outputs to uniformly random strings; therefore, the overall advantage is $11 \cdot \mathsf{Adv}_{\mathsf{PRF}^{\mathbb{Z}_n}}(1^\lambda, \mathcal{A}) + 4 \cdot \mathsf{Adv}_{\mathsf{PRF}^{\mathbb{Z}_{2^\ell}}}(1^\lambda, \mathcal{A})$ 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 $(\text{OREAD}, \mathsf{sid}, \mathsf{ssid}, P_j)$ for $j \in \{1,2\}$:
  - Set $\delta^{(j)}_{r,1} := 0$ and $\delta^{(j)}_{r,2} := w_{2,j} - \varphi_2^{(j)}$;
- Upon receiving $(\text{OREAD}, \mathsf{sid}, \mathsf{ssid}, P_j)$ for $j \in \{3,4\}$:
  - Set $\delta^{(j)}_{r,1} := 0$ and $\delta^{(j)}_{r,2} := -w_{2,j-2}$;
- Upon receiving $(\text{OWRITE}, \mathsf{sid}, \mathsf{ssid}, P_j)$ for $j \in \{1,3\}$:
  - Set $\delta^{(j)}_{w,1} := -r_1^{(\lceil j/2 \rceil)} + (j-2) \cdot w'_{1,1} + w'_{1,j+1}$;
  - Set $\delta^{(j)}_{w,2} := (j-2) \cdot w'_{2,1} + w'_{2,j+1}$;
- Upon receiving $(\text{OWRITE}, \mathsf{sid}, \mathsf{ssid}, P_j)$ for $j \in \{2,4\}$:
  - Set $\delta^{(j)}_{w,1} := (j-3) \cdot w'_{1,3} - w'_{1,j}$;
  - Set $\delta^{(j)}_{w,2} := (j-3) \cdot w'_{2,3} - w'_{2,j}$;

instead of

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

Claim 5. *If* $\mathsf{DPF}^{\mathbb{Z}_n,\mathbb{Z}_{2^\ell}} := (\mathsf{Gen},\mathsf{Eval})$ *is a secure function secret sharing scheme for point function* $f_{\alpha,\beta}(x) : \mathbb{Z}_n \mapsto \mathbb{Z}_{2^\ell}$ *with adversarial advantage* $\mathsf{Adv}_{\mathsf{DPF}}^{\mathbb{Z}_n,\mathbb{Z}_{2^\ell}}(1^\lambda,\mathcal{A})$*, then* $\mathcal{H}_2$ *and* $\mathcal{H}_1$ *are indistinguishable with advantage* $\epsilon_2 := 3\mathsf{Adv}_{\mathsf{DPF}}^{\mathbb{Z}_n,\mathbb{Z}_{2^\ell}}(1^\lambda,\mathcal{A})$*.*

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

- $\delta_{r,2}^{(1)} := i^{(1)} - \varphi_2^{(1)} + w_{2,1}$, $\delta_{r,2}^{(2)} := i^{(2)} - \varphi_2^{(2)} + w_{2,2}$;
- $\delta_{r,2}^{(3)} := i^{(1)} - w_{2,1}$, $\delta_{r,2}^{(4)} := i^{(1)} - w_{2,2}$;
- $\delta_{w,1}^{(1)} := i^{(j)} - r_1^{(1)} - w'_{1,1} + w'_{1,2}$, $\delta_{w,1}^{(2)} := i^{(j)} - w'_{1,3} - w'_{1,2}$;
- $\delta_{w,1}^{(3)} := i^{(j)} - r_1^{(2)} + w'_{1,1} + w'_{1,4}$, $\delta_{w,1}^{(4)} := i^{(j)} + w'_{1,3} - w'_{1,4}$;
- $\delta_{w,2}^{(1)} := i^{(j)} - w'_{2,1} + w'_{2,2}$, $\delta_{w,2}^{(2)} := i^{(j)} - r_2^{(1)} - w'_{2,3} - w'_{2,2}$;
- $\delta_{w,2}^{(3)} := i^{(j)} + w'_{2,1} + w'_{2,4}$, $\delta_{w,2}^{(4)} := i^{(j)} - r_2^{(2)} + w'_{2,3} - w'_{2,4}$;

It is straightforward that the distribution of $\{\delta_{r,2}^{(j)}\}_{j\in[4]}$ are uniformly random under the condition $\delta_{r,2} := \sum_{k=1}^{4}\delta_{r,2}^{(k)} = i - \varphi_2$ where φ_2 is used to generate the DPF keys $(\mathcal{K}_{\varphi_2}^{(1)},\mathcal{K}_{\varphi_2}^{(2)})$; similarly for $\{\delta_{w,1}^{(j)}\}_{j\in[4]}$ and $\{\delta_{w,2}^{(j)}\}_{j\in[4]}$. Whereas $\delta_{r,2} := -\varphi_2$, $\delta_{w,1} := -r_1$ and $\delta_{w,2} := -r_2$ in the hybrid $\mathcal{H}_2$, we can show that if there exists an adversary $\mathcal{A}$ who can distinguish the view of $\mathcal{H}_2$ from the view of $\mathcal{H}_1$ then we can construct an adversary $\mathcal{B}$ who uses $\mathcal{A}$ in a blackbox fashion can break at least one of the three $\mathsf{DPF}^{\mathbb{Z}_n,\mathbb{Z}_{2^\ell}} := (\mathsf{Gen},\mathsf{Eval})$ with the same advantage. Therefore, $\mathcal{H}_2$ and $\mathcal{H}_1$ are indistinguishable with adversarial advantage $\epsilon_2 := 3 \cdot \mathsf{Adv}_{\mathsf{DPF}^{\mathbb{Z}_n,\mathbb{Z}_{2^\ell}}}(1^\lambda,\mathcal{A})$. □

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

- Upon receiving $(\textsc{Owrite},\mathsf{sid},\mathsf{ssid},P_j)$ for $j \in \{1,3\}$:
  - Set $\Delta v^{(j)} := (j-2)\cdot w'_{3,1} + w'_{3,j+1}$;
- Upon receiving $(\textsc{Owrite},\mathsf{sid},\mathsf{ssid},P_j)$ 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 $(\textsc{Owrite},\mathsf{sid},\mathsf{ssid},P_j)$ for $j \in \{1,3\}$:
  - Set $\Delta v^{(j)} := y_i^{(j)} - x^{(j)} + (j-2)\cdot w'_{3,1} + w'_{3,j+1}$;
- Upon receiving $(\textsc{Owrite},\mathsf{sid},\mathsf{ssid},P_j)$ for $j \in \{2,4\}$:
  - Set $\Delta v^{(j)} := y_i^{(j)} - x^{(j)} - v^{(j/2)} + (j-3)\cdot w'_{3,3} - w'_{3,j}$;

Claim 6. *$\mathcal{H}_3$ and $\mathcal{H}_2$ are perfectly indistinguishable.*

Proof. Since $\{w'_{3,j}\}_{j\in[4]}$, $\{y_i^{(j)}\}_{j\in[4]}$ and $\{x^{(j)}\}_{j\in[4]}$ are uniformly random in $\mathbb{Z}_{2^\ell}$, the distribution of $\{\Delta v^{(j)}\}_{j\in[4]}$ 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 $\mathsf{Exec}_{\mathcal{F}_{\mathsf{read}}^{n,\ell},\mathcal{S},\mathcal{Z}}$. Therefore, the overall distinguishing advantage is

$$11\cdot\mathsf{Adv}_{\mathsf{PRF}^{\mathbb{Z}_n}}(1^\lambda,\mathcal{A}) + 4\cdot\mathsf{Adv}_{\mathsf{PRF}^{\mathbb{Z}_{2^\ell}}}(1^\lambda,\mathcal{A}) + 3\cdot\mathsf{Adv}_{\mathsf{DPF}^{\mathbb{Z}_n,\mathbb{Z}_{2^\ell}}}(1^\lambda,\mathcal{A})\ .$$

This concludes the proof. □

References

[1] J. Brickell, D. E. Porter, V. Shmatikov, and E. Witchel, "Privacy-preserving remote diagnostics," in *ACM CCS 2007*, P. Ning, S. De Capitani di Vimercati, and P. F. Syverson, Eds. Alexandria, Virginia, USA: ACM Press, Oct. 28–31, 2007, pp. 498–507.

[2] M. Barni, P. Failla, V. Kolesnikov, R. Lazzeretti, A.-R. Sadeghi, and T. Schneider, "Secure evaluation of private linear branching programs with medical applications," in *ESORICS 2009*, ser. LNCS, vol. 5789. Saint-Malo, France: Springer, Sep. 21–23, 2009, pp. 424–439.

[3] S. Niksefat, B. Sadeghiyan, P. Mohassel, and S. Sadeghian, "Zids: A privacy-preserving intrusion detection system using secure two-party computation protocols," *The Computer Journal*, vol. 57, no. 4, pp. 494–509, 2014.

[4] M. Abadi and J. Feigenbaum, "Secure circuit evaluation," *Journal of Cryptology*, vol. 2, no. 1, pp. 1–12, Feb. 1990.

[5] V. Kolesnikov and T. Schneider, "A practical universal circuit construction and secure evaluation of private functions," in *FC 2008*, ser. LNCS, vol. 5143. Cozumel, Mexico: Springer, Jan. 28–31, 2008, pp. 83–97.

[6] A.-R. Sadeghi and T. Schneider, "Generalized universal circuits for secure evaluation of private functions with application to data classification," in *ICISC 08*, ser. LNCS, vol. 5461. Seoul, Korea: Springer, Dec. 3–5, 2009, pp. 336–353.

[7] R. Raz, "Elusive functions and lower bounds for arithmetic circuits," in *40th ACM STOC*, R. E. Ladner and C. Dwork, Eds. Victoria, BC, Canada: ACM Press, May 17–20, 2008, pp. 711–720.

[8] A. Paus, A.-R. Sadeghi, and T. Schneider, "Practical secure evaluation of semi-private functions," in *ACNS 09*, ser. LNCS, vol. 5536. Paris-Rocquencourt, France: Springer, Jun. 2–5, 2009, pp. 89–106.

[9] J. Katz and L. Malka, "Constant-round private function evaluation with linear complexity," in *ASIACRYPT 2011*, ser. LNCS, vol. 7073. Seoul, South Korea: Springer, Dec. 4–8, 2011, pp. 556–571.

[10] P. Mohassel and S. S. Sadeghian, "How to hide circuits in MPC an efficient framework for private function evaluation," in *EUROCRYPT 2013*, ser. LNCS, vol. 7881. Athens, Greece: Springer, May 26–30, 2013, pp. 557–574.

[11] P. Mohassel, S. S. Sadeghian, and N. P. Smart, "Actively secure private function evaluation," in *ASIACRYPT 2014, Part II*, ser. LNCS, vol. 8874. Kaoshiung, Taiwan, R.O.C.: Springer, Dec. 7–11, 2014, pp. 486–505.

[12] Á. Kiss and T. Schneider, "Valiant's universal circuit is practical," in *EUROCRYPT 2016, Part I*, ser. LNCS, vol. 9665. Vienna, Austria: Springer, May 8–12, 2016, pp. 699–728.

[13] D. Günther, Á. Kiss, and T. Schneider, "More efficient universal circuit constructions," in *ASIACRYPT 2017, Part II*, ser. LNCS, vol. 10625. Hong Kong, China: Springer, Dec. 3–7, 2017, pp. 443–470.

[14] O. Bicer, M. A. Bingol, M. S. Kiraz, and A. Levi, "Highly efficient and re-executable private function evaluation with linear complexity," *IEEE Transactions on Dependable and Secure Computing*, pp. 1–1, 2020.

[15] O. Goldreich and R. Ostrovsky, "Software protection and simulation on oblivious rams," *Journal of the ACM (JACM)*, vol. 43, no. 3, pp. 431–473, 1996.

[16] P. Williams and R. Sion, "Usable PIR," in *NDSS 2008*. San Diego, CA, USA: The Internet Society, Feb. 10–13, 2008.

[17] P. Williams, R. Sion, and B. Carbunar, "Building castles out of mud: practical access pattern privacy and correctness on untrusted storage," in *ACM CCS 2008*. Alexandria, Virginia, USA: ACM Press, Oct. 27–31, 2008, pp. 139–148.

[18] B. Pinkas and T. Reinman, "Oblivious RAM revisited," in *CRYPTO 2010*, ser. LNCS, vol. 6223. Santa Barbara, CA, USA: Springer, Aug. 15–19, 2010, pp. 502–519.

[19] M. T. Goodrich and M. Mitzenmacher, "Privacy-preserving access of outsourced data via oblivious RAM simulation," in *ICALP 2011, Part II*, ser. LNCS, vol. 6756. Zurich, Switzerland: Springer, Jul. 4–8, 2011, pp. 576–587.

[20] E. Shi, T.-H. H. Chan, E. Stefanov, and M. Li, "Oblivious RAM with $O((\log N)^3)$ worst-case cost," in *ASIACRYPT 2011*, ser. LNCS, vol. 7073. Seoul, South Korea: Springer, Dec. 4–8, 2011, pp. 197–214.

[21] E. Kushilevitz, S. Lu, and R. Ostrovsky, "On the (in)security of hash-based oblivious RAM and a new balancing scheme," in *23rd SODA*. Kyoto, Japan: ACM-SIAM, Jan. 17–19, 2012, pp. 143–156.

[22] P. Williams and R. Sion, "Single round access privacy on outsourced storage," in *ACM CCS 2012*, T. Yu, G. Danezis, and V. D. Gligor, Eds. Raleigh, NC, USA: ACM Press, Oct. 16–18, 2012, pp. 293–304.

[23] E. Stefanov and E. Shi, "ObliviStore: High performance oblivious cloud storage," in *2013 IEEE Symposium on Security and Privacy*. Berkeley, CA, USA: IEEE Computer Society Press, May 19–22, 2013, pp. 253–267.

[24] E. Stefanov, M. van Dijk, E. Shi, C. W. Fletcher, L. Ren, X. Yu, and S. Devadas, "Path ORAM: an extremely simple oblivious RAM protocol," in *ACM CCS 2013*. Berlin, Germany: ACM Press, Nov. 4–8, 2013, pp. 299–310.

[25] S. Lu and R. Ostrovsky, "Distributed oblivious RAM for secure two-party computation," in *TCC 2013*, ser. LNCS, A. Sahai, Ed., vol. 7785. Tokyo, Japan: Springer, Mar. 3–6, 2013, pp. 377–396.

[26] S. D. Gordon, J. Katz, and X. Wang, "Simple and efficient two-server ORAM," in *ASIACRYPT 2018, Part III*, ser. LNCS, T. Peyrin and S. Galbraith, Eds., vol. 11274. Brisbane, Queensland, Australia: Springer, Dec. 2–6, 2018, pp. 141–157.

[27] E. Kushilevitz and T. Mour, "Sub-logarithmic distributed oblivious RAM with small block size," in *PKC 2019, Part I*, ser. LNCS, D. Lin and K. Sako, Eds., vol. 11442. Beijing, China: Springer, Apr. 14–17, 2019, pp. 3–33.

[28] E. Dauterman, M. Rathee, R. A. Popa, and I. Stoica, "Waldo: A private time-series database from function secret sharing," p. 1661, 2021. [Online]. Available: https://eprint.iacr.org/2021/1661

[29] X. S. Wang, Y. Huang, T.-H. H. Chan, a. shelat, and E. Shi, "SCORAM: Oblivious RAM for secure computation," in *ACM CCS 2014*. Scottsdale, AZ, USA: ACM Press, Nov. 3–7, 2014, pp. 191–202.

[30] S. Faber, S. Jarecki, S. Kentros, and B. Wei, "Three-party ORAM for secure computation," in *ASIACRYPT 2015, Part I*, ser. LNCS, T. Iwata and J. H. Cheon, Eds., vol. 9452. Auckland, New Zealand: Springer, Nov. 30 – Dec. 3, 2015, pp. 360–385.

[31] X. Wang, T.-H. H. Chan, and E. Shi, "Circuit ORAM: On tightness of the Goldreich-Ostrovsky lower bound," in *ACM CCS 2015*. Denver, CO, USA: ACM Press, Oct. 12–16, 2015, pp. 850–861.

[32] S. Zahur, X. S. Wang, M. Raykova, A. Gascón, J. Doerner, D. Evans, and J. Katz, "Revisiting square-root ORAM: Efficient random access in multi-party computation," in *2016 IEEE Symposium on Security and Privacy*. San Jose, CA, USA: IEEE Computer Society Press, May 22–26, 2016, pp. 218–234.

[33] S. Jarecki and B. Wei, "3PC ORAM with low latency, low bandwidth, and fast batch retrieval," in *ACNS 18*, ser. LNCS, B. Preneel and F. Vercauteren, Eds., vol. 10892. Leuven, Belgium: Springer, Jul. 2–4, 2018, pp. 360–378.

[34] J. Doerner and a. shelat, "Scaling ORAM for secure computation," in *ACM CCS 2017*. Dallas, TX, USA: ACM Press, Oct. 31 – Nov. 2, 2017, pp. 523–535.

[35] P. Bunn, J. Katz, E. Kushilevitz, and R. Ostrovsky, "Efficient 3-party distributed ORAM," in *SCN 20*, ser. LNCS, vol. 12238. Amalfi, Italy: Springer, Sep. 14–16, 2020, pp. 215–232.

[36] B. H. Falk, D. Noble, and R. Ostrovsky, "3-party distributed ORAM from oblivious set membership," p. 1463, 2021. [Online]. Available: https://eprint.iacr.org/2021/1463

[37] E. Boyle, N. Gilboa, and Y. Ishai, "Function secret sharing," in *EUROCRYPT 2015, Part II*, ser. LNCS, vol. 9057. Sofia, Bulgaria: Springer, Apr. 26–30, 2015, pp. 337–367.

[38] E. Ben-Sasson, A. Chiesa, D. Genkin, E. Tromer, and M. Virza, “SNARKs for C: Verifying program executions succinctly and in zero knowledge,” in *CRYPTO 2013, Part II*, ser. LNCS, vol. 8043. Santa Barbara, CA, USA: Springer, Aug. 18–22, 2013, pp. 90–108.

[39] X. S. Wang, S. D. Gordon, A. McIntosh, and J. Katz, “Secure computation of MIPS machine code,” in *ESORICS 2016, Part II*, ser. LNCS, I. G. Askoxylakis, S. Ioannidis, S. K. Katsikas, and C. A. Meadows, Eds., vol. 9879. Heraklion, Greece: Springer, Sep. 26–30, 2016, pp. 99–117.

[40] M. Keller and A. Yanai, “Efficient maliciously secure multiparty computation for RAM,” in *EUROCRYPT 2018, Part III*, ser. LNCS, J. B. Nielsen and V. Rijmen, Eds., vol. 10822. Tel Aviv, Israel: Springer, Apr. 29 – May 3, 2018, pp. 91–124.

[41] M. Keller, “The oblivious machine,” in *Progress in Cryptology – LATINCRYPT 2017*, T. Lange and O. Dunkelman, Eds. Cham: Springer International Publishing, 2019, pp. 271–288.

[42] R. Canetti, “Universally composable security: A new paradigm for cryptographic protocols,” in *FOCS’01*. IEEE, 2001, pp. 136–145.

[43] V. Kolesnikov, N. Matania, B. Pinkas, M. Rosulek, and N. Trieu, “Practical multi-party private set intersection from symmetric-key techniques,” in *ACM CCS 2017*, B. M. Thuraisingham, D. Evans, T. Malkin, and D. Xu, Eds. Dallas, TX, USA: ACM Press, Oct. 31 – Nov. 2, 2017, pp. 1257–1272.

[44] S. Zahur, M. Rosulek, and D. Evans, “Two halves make a whole - reducing data transfer in garbled circuits using half gates,” in *EUROCRYPT 2015, Part II*, ser. LNCS, vol. 9057. Sofia, Bulgaria: Springer, Apr. 26–30, 2015, pp. 220–250.

[45] D. Beaver, S. Micali, and P. Rogaway, “The round complexity of secure protocols (extended abstract),” in *22nd ACM STOC*. Baltimore, MD, USA: ACM Press, May 14–16, 1990, pp. 503–513.

[46] E. Boyle, N. Gilboa, and Y. Ishai, “Function secret sharing: Improvements and extensions,” in *ACM CCS 2016*. Vienna, Austria: ACM Press, Oct. 24–28, 2016, pp. 1292–1303.

[47] E. Boyle, N. Chandran, N. Gilboa, D. Gupta, Y. Ishai, N. Kumar, and M. Rathee, “Function secret sharing for mixed-mode and fixed-point secure computation,” in *EUROCRYPT 2021, Part II*, ser. LNCS, vol. 12697. Zagreb, Croatia: Springer, Oct. 17–21, 2021, pp. 871–900.

[48] V. Kolesnikov and T. Schneider, “Improved garbled circuit: Free XOR gates and applications,” in *ICALP 2008, Part II*, ser. LNCS, vol. 5126. Reykjavik, Iceland: Springer, Jul. 7–11, 2008, pp. 486–498.

[49] X. Wang, A. J. Malozemoff, and J. Katz, “EMP-toolkit: Efficient MultiParty computation toolkit,” https://github.com/emp-toolkit, 2016.