Ramen: Souper Fast Three-Party Computation for RAM Programs

Full Version*

Lennart Braun[1], Mahak Pancholi[1], Rahul Rachuri[2]**, and Mark Simkin[3]
{braun,mahakp}@cs.au.dk,srachuri@visa.com,mark.simkin@ethereum.org

[1] Aarhus University
[2] Visa Research
[3] Ethereum Foundation

Abstract. Secure RAM computation allows a number of parties to evaluate a function represented as a random-access machine (RAM) program in a way that reveals nothing about the private inputs of the parties except from what is already revealed by the function output itself. In this work we present *Ramen*, which is a new protocol for computing RAM programs securely among three parties, tolerating up to one passive corruption. Ramen provides reasonable asymptotic guarantees and is concretely efficient at the same time. We have implemented our protocol and provide extensive benchmarks for various settings.

Asymptotically, our protocol requires a constant number of rounds and an amortized sublinear amount of communication and computation per memory access. In terms of concrete efficiency, our protocol outperforms previous solutions. For a memory of size 2^{26} our memory accesses are 25× faster in the LAN and 8× faster in the WAN setting, when compared to the previously fastest, and concurrent, solution by Vadapalli, Henry, and Goldberg (USENIX Security 2023). Due to our superior asymptotic guarantees, the efficiency gap is only widening as the memory gets larger and for this reason Ramen provides the currently most scalable concretely efficient solution for securely computing RAM programs.

1 Introduction

In the secure computation setting, multiple mutually distrustful parties wish to compute a joint function of their private inputs, without revealing any information not already revealed by the function's output. To perform this task, the parties need to agree on a model of computation, which defines how the function is represented and how individual computational steps look like. Different models of computation, like circuits or random-access machines (RAMs), are suitable for different functions. The canonical example for a computation that can be performed very efficiently in one model, but not in another is binary search. A RAM program that searches through a sorted list of length n needs to only perform $\log n$ memory accesses, but the functionally equivalent circuit would need to parse the full list as an input, which means that the circuit size depends at least linearly on the length of the list. More generally speaking, RAMs are usually preferable over circuits for computations that involve sparse data-dependent accesses across large datasets. Prominent examples of such computations are breadth-first search, Dijkstra's path finding, and Gale-Shapley's algorithm for computing stable matchings.

In the early years of secure computation research [Yao82, Yao86, GMW87, BGW88, CCD88] and throughout the 90s and 00s the overwhelming majority of research works exclusively focused on efficiently computing functions represented as circuits, but over the past decade or so there has been an increased interest in securely computing RAM programs [GKK+12, GHL+14, GLOS15, GLO15, FJKW15, GGMP16, ZWR+16, Ds17, JW18, KY18, BKKO20, HV21, HKO22]. Existing works on securely computing RAM programs can be roughly categorized as follows:

Theoretical Foundations. The first construction for securely computing RAM programs with a poly-logarithmic computational and bandwidth overhead per party per access was presented by Gordon et

---

* This is the full version of [BPRS23] which appeared at CCS'23.
** The majority of the work was done while at Aarhus University.

al. [GKK+12].[4] Subsequently, a series of theoretical works have shown how to construct protocols that run in a constant number of rounds [LO13] and that only require blackbox use of minimal computational building blocks [GHL+14, GLOS15, GLO15, GGMP16]. These results provide important theoretical insights, but do not lead to practically efficient protocols.

Concretely Efficient Protocols. A different line of works [FJKW15, ZWR+16, Ds17, JW18, KY18, BKKO20, HV21, HKO22] has focused on constructing concretely efficient protocols in the two- and multiparty setting. Protocols that tolerate a dishonest majority of participants [ZWR+16, Ds17, KY18, HV21] naturally require more expensive public-key operations and thus are generally slower than protocols in the honest majority setting [FJKW15, JW18, BKKO20, HV21, VHG23]. One crucial observation underlying most of these works is that a larger amount of cheap operations can be executed faster than a small amount of expensive operations. With this in mind, most of these works sacrifice asymptotic efficiency for concrete performance gains. The protocols of Doerner and Shelat [Ds17], Bunn et al. [BKKO20], and Vadapalli, Henry, and Goldberg [VHG23], for example, all require each party to perform a linear amount of cheap operations for each memory access. Unfortunately, asymptotics are bound to kick in as the memory gets larger and thus a scalable protocol needs to balance asymptotic as well as concrete efficiency guarantees.

1.1 Our Contribution

In this work, we present a new protocol, called *Ramen*, for securely computing RAM programs among three parties in the presence of one passive corruption. Our protocol outperforms previous works in terms of concrete efficiency, while at the same time providing reasonable asymptotic efficiency guarantees. In terms of asymptotic guarantees, we provide a comparison to the most relevant related works in Table 1. We have implemented our protocol and benchmarked its concrete efficiency in various settings. As an exemplary data point, for a memory of size $n = 2^{26}$, our average memory access time is 25× faster in the LAN and 8× faster in the WAN setting, when compared to the currently concretely fastest protocol of Vadapalli, Henry, and Goldberg [VHG23]. Since our protocol has better asymptotic guarantees, this efficiency gap is only going to widen, when the memory size increases. For this reason Ramen represents the currently *most concretely efficient and scalable solution* for secure RAM computation in the three-party setting with one passive corruption.

Table 1: Comparison with most relevant prior works in the three-party setting with one corruption. The table displays amortized costs per access. In case of [VHG23], the polylog(n) factor is the preprocessing cost, while $\mathcal{O}(1)$ is the online cost.

	Rounds	Communication	Computation
Jarecki et al. [JW18]	$\mathcal{O}(\text{polylog}(n))$	$\mathcal{O}(\text{polylog}(n))$	$\mathcal{O}(\text{polylog}(n))$
Bunn et al. [BKKO20]	$\mathcal{O}(1)$	$\mathcal{O}(\sqrt{n})$	$\mathcal{O}(n)$
Vadapalli et al. [VHG23]	$\mathcal{O}(\text{polylog}(n)) + \mathcal{O}(1)$	$\mathcal{O}(\text{polylog}(n)) + \mathcal{O}(1)$	$\mathcal{O}(n)$
Ramen (This Work)	$\mathcal{O}(1)$	$\mathcal{O}(\sqrt{n} \cdot \text{polylog}(n))$	$\mathcal{O}(\sqrt{n} \cdot \text{polylog}(n))$

1.2 Technical Overview

Similarly to other protocols [ZWR+16, Ds17, HV21, VHG23] aiming for practical efficiency, we start with the high-level concept introduced by Gordon et al. [GKK+12]. We have a stateful CPU and a memory M of length n. The CPU will repeatedly provide read or write instructions and in case of a read operation, the read memory value will be provided as input to the CPU at its next invocation. We will assume

[4] Similar ideas have already appeared in the earlier works of Ostrovsky and Shoup [OS97] as well as Damgård, Meldgaard and Nielsen [DMN11].

that the CPU can be realized using generic secure computation techniques for circuits and that each CPU invocation will return the operation and memory location in a secret-shared form to the involved parties. The main technical challenge and focus of our work is to design efficient protocols that take the secret-shared operation and efficiently execute it on the secret-shared memory data structure.

Square-Root ORAM Our starting point is the square-root oblivious RAM (ORAM) data structure of Goldreich and Ostrovsky [GO96]. ORAM allows a client to access an encrypted memory held by an untrusted server in a communication-efficient manner that does not reveal which operation was performed at what location. Goldreich and Ostrovsky's construction allows the client to perform read and write operations with an amortized communication overhead of $\mathcal{O}\left(\sqrt{n} \cdot \mathrm{polylog}(n)\right)$.

The square-root construction works as follows: To encode a memory M of size n, the client first appends $\sqrt{n}$ dummy elements to the real memory and picks a pseudorandom permutation π over the domain $\{1, \ldots, n + \sqrt{n}\}$. The encoded memory is defined as $\widetilde{\mathsf{M}}[\pi(i)] := \mathsf{M}[i]$ for all $1 \leq i \leq n + \sqrt{n}$. The client stores π and the server stores an encryption of $\widetilde{\mathsf{M}}$ along with an initially empty array stash of size $\sqrt{n}$.

Whenever the client wants to perform an operation on $\mathsf{M}[i]$, it downloads the full stash and checks whether $\mathsf{M}[i]$ is in it. If it is, then the client performs the desired read or write operation on the element in the stash and reads a dummy element from $\widetilde{\mathsf{M}}$ that has not yet been read. If the desired entry $\mathsf{M}[i]$ is not in the stash, then the client directly accesses $\widetilde{\mathsf{M}}[\pi(i)]$, performs the desired operation, and moves it to the stash. To ensure that the stash does not become overfull, the client will, roughly speaking, download everything from the server every $\sqrt{n}$ accesses, move all updates from the stash into main memory and then reinitialize this data structure with a new empty stash and a fresh permutation π. In the following, we will call each such time window of $\sqrt{n}$ accesses an *epoch*. On an intuitive level, the server can never tell which operation is being performed at what location, since the client always accesses a fresh random address in $\widetilde{\mathsf{M}}$ and fully downloads the stash.

In the context of secure computation, the main idea of Gordon et al. [GKK+12], and many of the subsequent works, was to let the parties jointly play the role of the server and at the same time simulate the ORAM client through an efficient secure computation protocol. We will refer to this collection of protocols and secret-shared values as the *distributed ORAM*. To securely simulate the client for the square-root ORAM construction, we need to be able to efficiently access a secret-shared stash, efficiently access a secret-shared version of the encoded memory $\widetilde{\mathsf{M}}$, and to be able to securely perform the reinitialization step. Before providing a high-level overview of how we realize those components, let us introduce two important cryptographic tools that we will make extensive use of.

Distributed Point Functions (DPFs) Let $f \colon X \to Y$ be a point function, parameterized by values s and y, which is defined as

$$f(x) = \begin{cases} y & \text{if } x = s \\ 0 & \text{otherwise,} \end{cases}$$

and let λ be the security parameter. Gilboa and Ishai [GI14] introduced the concept of distributed point functions, which allow one to take f as input and produce functions f_1 and f_2 of size $\mathcal{O}(\lambda \log X)$ bits each, such that both look pseudorandom individually, but $f_1(x) + f_2(x) = f(x)$ for all inputs x. Such functions can be realized with good concrete efficiency from one-way functions and are useful for performing efficient two-server private information retrieval. Here we have servers S_1 and S_2 holding vector $\boldsymbol{v}$ of length n and client C wanting to privately retrieve $\boldsymbol{v}[i]$. The client generates functions f_1 and f_2 for a point function f that evaluates to 1 at i and sends f_b to S_b for $b \in \{1, 2\}$. The servers independently compute $\sum_{i=1}^{n} f_b(i) \cdot \boldsymbol{v}[i]$ and send the result back to C. The client can sum up the two received values to retrieve $\boldsymbol{v}[i]$ without either of the servers having learned i. Distributed point functions have also been generalized to multi-point functions that allow for encoding of more than one pair of input and (non-zero) output [BCGI18].

Distributed Oblivious PRFs (DOPRFs) Let $\mathcal{F}_{\mathsf{DOPRF}}$ be an ideal functionality that can be accessed by parties P_1, P_2, and P_3. Assume either P_1 alone or both P_1 and P_2 hold a secret key k for a pseudorandom function PRF and that all parties jointly hold an additive secret sharing of a value x, i.e., each P_i holds

x_i so that $x = x_1 + x_2 + x_3$. Using $\mathcal{F}_{\mathsf{DOPRF}}$, the parties can jointly compute an additive secret sharing of the value $\mathsf{PRF}(k, x)$. Whenever k, x are given in plain, the value $\mathsf{PRF}(k, x)$ can be computed locally without using $\mathcal{F}_{\mathsf{DOPRF}}$. We will assume that the output domain is sufficiently large to ensure that no polynomial-time adversary, not knowing k, can find two inputs that produce the same output. The function PRF is thus indistinguishable from a pseudorandom permutation, a fact also known as the switching lemma.

Initializing the Distributed ORAM Now that we have introduced the main cryptographic tools relevant to our work, let us discuss the general structure of our distributed ORAM. The main memory M, along with appended dummy elements, will be secret-shared into shares $\mathsf{M}_1, \mathsf{M}_2, \mathsf{M}_3$, such that $\mathsf{M} = \mathsf{M}_1 + \mathsf{M}_2 + \mathsf{M}_3$. For $i \in \{1, 2, 3\}$, parties P_{i-1} and P_{i+1} will hold a PRF key k_i.[5] Additionally, P_{i-1} will hold a random mask vector $\boldsymbol{r}_i$ of length $m := n + \sqrt{n}$. Party P_i will hold a randomly permuted and masked version of the memory share M_{i-1}. More precisely, P_i will have a vector M'_i of tuples $(\mathsf{PRF}(k_i, j), \mathsf{M}_{i-1}[j] + \boldsymbol{r}_i[\mathsf{PRF}(k_i, j)])$, for $j \in \{1, \ldots, m\}$, sorted by $\mathsf{PRF}(k_i, j)$. In this overview, we will largely ignore the role of the mask vectors $\boldsymbol{r}_i$, but we would like to stress that they will help improving the concrete efficiency in our full construction during reinitialization at the end of each epoch.

The stash is realized by an additive secret sharing of an ordered list denoted by stash among all parties. Additionally, P_2 and P_3 hold an initially empty ordered list L_{stash} in plain and P_1 holds a PRF key k_{stash}. The list L_{stash} will hold values $\mathsf{PRF}(k_{\mathsf{stash}}, \mathsf{adr})$ for all addresses adr that have already been read in the current epoch. The secret-shared stash will hold tuples $(\mathsf{val}, \mathsf{val}_{\mathsf{old}})$, where the j-th entry corresponds to the memory location accessed at time j in the current epoch. The value val is the current value at that memory location, and $\mathsf{val}_{\mathsf{old}}$ is the value that was at that address at the beginning of the epoch. Looking ahead, storing both val and $\mathsf{val}_{\mathsf{old}}$ will help the parties to just add the difference between the old and the new value to their additive secret shares of M.

Accessing the Stash Whenever the parties receive an additive secret sharing of adr from the CPU, the parties need to determine, whether the address is already in the stash. For this, the parties first use $\mathcal{F}_{\mathsf{DOPRF}}$ to compute a secret sharing of $\mathsf{PRF}(k_{\mathsf{stash}}, \mathsf{adr})$. Parties P_2 and P_3 agree on a mask R and let P_1 reconstruct the value $\mathsf{PRF}(k_{\mathsf{stash}}, \mathsf{adr}) + R$. Party P_1 then generates DPF keys k_2, k_3 for a point function that evaluates to 1 at $\mathsf{PRF}(k_{\mathsf{stash}}, \mathsf{adr}) + R$ and to 0 everywhere else. P_2 and P_3 receive k_2 and k_3, respectively. Using their list L_{stash} of read addresses, they add the mask R to each entry and evaluate their DPF key shares on each entry. If $\mathsf{PRF}(k_{\mathsf{stash}}, \mathsf{adr})$ was read at access j in the current epoch, then at this point, P_2 and P_3 will hold a secret sharing of a vector $\boldsymbol{v}$ of zero entries with a single 1 at location j. If the address was not yet read, then all entries in $\boldsymbol{v}$ will be 0. P_2 and P_3 non-interactively compute $\mathsf{flag} \in \{0, 1\}$, which is sum of all entries in $\boldsymbol{v}$, and loc, which is the inner product of $\boldsymbol{v}$ and the public vector $(1, 2, \ldots, \sqrt{n})$. At this point, the parties have a secret sharing of a flag indicating, whether the address is already in the stash, as well as a secret sharing of the potential location of the address in the stash. At the same time none of the parties learn whether adr was actually found in the stash.

Using a small amount of generic secure computation, the parties can use loc to either reveal $\mathsf{PRF}(k_{\mathsf{stash}}, \mathsf{adr})$ or $\mathsf{PRF}(k_{\mathsf{stash}}, \widetilde{\mathsf{adr}})$, where $\widetilde{\mathsf{adr}}$ is an unused dummy address, to P_2 and P_3, who can add it to their list L_{stash}. Finally, the parties can use similar tricks in combination with their knowledge of loc to either read from or write to the desired address in the stash.

Accessing the Memory In addition to the stash access, the parties also need to access adr', which is either the desired address adr or an unused dummy address $\widetilde{\mathsf{adr}}$ in the main memory. Recall that each party P_i holds key k_{i+1}, mask $\boldsymbol{r}_{i+1}$, and a secret share of the memory that is permuted using key k_i and masked using mask $\boldsymbol{r}_i$. Using three invocations of $\mathcal{F}_{\mathsf{DOPRF}}$, the parties can all learn which location in their memory share they should retrieve. Each party will locally keep track of which location in their memory was accessed at which time step during an epoch. Note that we never access the same address twice, always either an address not in the stash or a fresh dummy address, in the main memory. Thus we successfully hide, which location is actually being touched.

[5] We assume that parties' indices wrap around, i.e., that $P_{3+1} = P_1$ and $P_{1-1} = P_3$.

Flushing the Stash As already mentioned above, the stash has a fixed capacity for storing $\sqrt{n}$ entries. For this reason, we need to move all modified data entries residing in the stash back into the secret-shared memory M and reinitialize our whole distributed ORAM data structure afresh every $\sqrt{n}$ accesses. At the end of each epoch, each P_i holds a sorted list of addresses it has touched in its local permuted memory M'_i during the epoch. Additionally, she holds a secret sharing of the old and new values corresponding to each of those accesses in the stash. Parties P_{i-1} and P_{i+1} know how the memory of P_i was permuted, i.e., they know key k_i. For the sake of a concrete example, assume an epoch is two accesses long and that P_i holds a list of read values

$$\Big((\mathsf{PRF}(k_i, \mathsf{adr}), \mathsf{PRF}(k_i, \widetilde{\mathsf{adr}})\Big)$$

along with a secret share of stash

$$\Big((\mathsf{val}, \mathsf{val}_{\mathsf{old}}), (\widetilde{\mathsf{val}}, \widetilde{\mathsf{val}}_{\mathsf{old}})\Big).$$

P_i creates keys f_{i-1} and f_{i+1} for a multi-point function that evaluates to her share of $\mathsf{val} - \mathsf{val}_{\mathsf{old}}$ and $\widetilde{\mathsf{val}} - \widetilde{\mathsf{val}}_{\mathsf{old}}$ at locations $\mathsf{PRF}(k_i, \mathsf{adr})$ and $\mathsf{PRF}(k_i, \widetilde{\mathsf{adr}})$, respectively. She sends f_{i-1} and f_{i+1} to P_{i-1} and P_{i+1}, who evaluate their received functions at $\mathsf{PRF}(k_i, 1), \ldots, \mathsf{PRF}(k_i, n + \sqrt{n})$. These evaluations create an additively secret-shared vector between P_{i-1} and P_{i+1}. At positions adr and $\widetilde{\mathsf{adr}}$, the vector contains P_i's share of the updates to the values in memory M. At all other positions, the vector contains a secret sharing of 0. Doing this three times, rotating which party creates the multi-point DPF keys, allows the parties to create secret-shared vectors that can be added to secret shares of M to apply the updates from the stash to M privately. At this point parties can pick fresh permutations, and produce fresh shares of the permuted and masked memory that can be used in the next epoch.

We stress that flushing the memory updates from the secret-shared stash back into the secret-shared main memory requires no complicated or expensive secure computation techniques and only relies on plain DPF key generation and evaluation as well as evaluations of a PRF, all operations which are highly efficient. In our technical overview we have omitted several details that are important both for concrete efficiency and security. These will be provided in the relevant technical sections.

1.3 Related Work

Now that we are familiar with the conceptual ideas of our work, it is worth taking a step back and discussing how our work differs from previous constructions. In our discussion here we will focus on concretely efficient protocols in the three-party setting with one passive corruption and works that are similar to ours on a conceptual level [FJKW15, ZWR+16, Ds17, JW18, BKKO20, HV21, VHG23].

The works of Faber et al. [FJKW15] as well as Jarecki and Wei [JW18] construct concretely efficient three-party protocols secure against one passive corruption. Both works use tree-based ORAM constructions as their starting point. These have $\mathcal{O}(\mathrm{polylog}(n))$ bandwidth overheads per access, but are generally difficult to adapt to the distributed ORAM setting. Their resulting protocols are rather complex, require $\mathcal{O}(\log n)$ rounds of communication for each memory access, and thus suffer in terms of concrete efficiency.

Zahur et al. [ZWR+16] were the first to construct secure multiparty RAM computation protocols based on square-root ORAM that were secure against a majority of corrupt parties. Their bandwidth and computational overhead is $\mathcal{O}(\sqrt{n} \cdot \mathrm{polylog}(n))$ per party per access and each access requires $\mathcal{O}(\log n)$ rounds of communication. Since their work considers a dishonest majority of parties, they necessarily require larger amounts of computationally expensive public key operations, which affects their concrete efficiency.

Doerner and Shelat [Ds17] were the first to use DPFs in conjunction with the square-root ORAM construction to construct efficient two-party secure computation protocols that have a bandwidth overhead of $\mathcal{O}(\sqrt{n})$, a computational overhead of $\mathcal{O}(n)$, and require $\mathcal{O}(1)$ rounds per access. The authors show that their construction outperforms the constructions of Zahur et al. [ZWR+16], despite their asymptotically prohibitively expensive computational overhead for each access. Bunn et al. [BKKO20] extend the idea of using DPFs for ORAM constructions to the three-party setting with one passive corruptions, while achieving similar asymptotic guarantees.

Hamlin and Varia [HV21] construct the first secure two-party protocol based on square-root ORAM that achieves a sublinear amount of computation, a sublinear bandwidth overhead, and a requires a constant number rounds per access. Conceptually their construction has similarities to ours. In their work, the parties jointly permute the secret-shared memory M without revealing the permutation itself to either of the parties. Since the permutation is not known to either party, their reinitialization step at the end of each epoch is more complicated and expensive. Concretely, it requires amortized $\sqrt{n}$ invocations of the $\mathcal{F}_{\mathsf{DOPRF}}$ functionality per access. In contrast to their protocol, our reinitialization is very simple and we only require a small constant number of $\mathcal{F}_{\mathsf{DOPRF}}$ invocations per access.

Concurrently and independently from our work, Vadapalli, Henry, and Goldberg [VHG23] have recently published a new three-party protocol, based on the ideas of Doerner and Shelat [Ds17], with security against one passive corruption that outperforms all previous works in terms of concrete efficiency in the same setting. Their protocol requires $\mathcal{O}(\text{polylog}(n))$ rounds and communication in the preprocessing phase, and only $\mathcal{O}(1)$ rounds, and bandwidth overhead of $\mathcal{O}(1)$ in the online phase. However, it suffers from a computational overhead of $\mathcal{O}(n)$ per access per party. In contrast to their work, we also require $\mathcal{O}(1)$ rounds per access and we have both a bandwidth and computational overhead of $\mathcal{O}(\sqrt{n} \cdot \text{polylog}(n))$ per access per party. As we show in Section 5, the superior asymptotic costs of our construction start to matter, when the memory size gets larger and our protocol becomes concretely faster than the one of Vadapalli, Henry, and Goldberg. Notably, the increase of the average time per access between memories of sizes 2^8 and 2^{26} in the WAN setting is less than a factor of 1.5 for our protocol, but more than a factor of 20 for the protocol of Vadapalli, Henry, and Goldberg.

2 Preliminaries

2.1 Notation

The computational security parameter is denoted by λ, and stat is the statistical security parameter. Elements in memory M are elements from a field $\mathbb{F}_p$, where the field size is at least 2^λ. We use $:=$ to indicate assignment, $\leftarrow$ for sampling uniformly at random, and $=$ for comparisons. We use $[\ell]$ to denote integers $\{1, \ldots, \ell\}$.

RAM Program A RAM program consists of a sequence of CPU instructions which are realized by a next step function NS. We let $\mathsf{st}_{\mathsf{NS}}$ denote the current internal state of the program execution. The function NS takes $\mathsf{st}_{\mathsf{NS}}$ as well as the last read value as input and outputs an instruction I and the updated state $\mathsf{st}_{\mathsf{NS}}$. The instruction I is a tuple $(\mathsf{op}, \mathsf{adr}, \mathsf{val})$, where op is either read or write and val is the value to be written into the memory M at location adr. The state $\mathsf{st}_{\mathsf{NS}}$ contains start, stop, or $\mathsf{continue}$ denoting the current state of program execution. The state might also contain additional information, such as the final output, which we denote by z.

Memory We parse M as an array of tuples $(\mathsf{indx}, \mathsf{val})$, where indx denotes the index ranging from $1, \ldots, n$, and val is the value at that index. We write $\mathsf{M}[j].\mathsf{val}$ and $\mathsf{M}[j].\mathsf{indx}$ to denote the value, respectively the index at location j in M.

Stash We write stash to denote the stash which is an array of size $\sqrt{n}$ where each element is a tuple $(\mathsf{adr}, \mathsf{val}, \mathsf{val}_{\mathsf{old}})$, where val is the most recent value at $\mathsf{M}[\mathsf{adr}]$, and $\mathsf{val}_{\mathsf{old}}$ is the value that was read from $\mathsf{M}[\mathsf{adr}]$ when the element was first moved to the stash, i.e., the value in $\mathsf{M}[\mathsf{adr}]$ at the start of the epoch. We write $\mathsf{stash}[j].\mathsf{adr}$ (similarly for $\mathsf{stash}[j].\mathsf{val}$ and $\mathsf{stash}[j].\mathsf{val}_{\mathsf{old}}$) to denote the address stored at index j in the stash. We write $\mathsf{st}_{\mathsf{stash}}$ to denote the state of stash, which is a tuple $(\mathsf{flag}, \mathsf{loc}, \mathsf{val}_{\mathsf{st}})$. The flag flag indicates whether or not the address in the current instruction is already stored in the stash at some index, loc stores this index, and $\mathsf{val}_{\mathsf{st}}$ stores the (up-to-date) value at this index.

Linear Secret Sharing A value $\mathsf{val} \in \mathbb{F}_p$ is additively secret shared among P_1, P_2, P_3, if P_i for $i \in \{1, 2, 3\}$ holds $\langle \mathsf{val} \rangle_i \in \mathbb{F}_p$ such that $\mathsf{val} = \sum_{i=1}^{3} \langle \mathsf{val} \rangle_i$. We omit the party subscript i wherever it is obvious from the context. We also use replicated secret sharing, which is denoted by $[\![\cdot]\!]$, where each P_i is given two of three additive shares, such that any two parties can reconstruct val.

Pseudorandom Function (PRF) Let $\mathsf{PRF}\colon \mathcal{K}_{\mathsf{PRF}} \times \mathcal{X}_{\mathsf{PRF}} \to \mathcal{Y}_{\mathsf{PRF}}$ be a pseudorandom function, where $\mathcal{K}_{\mathsf{PRF}}$ is the key space, $\mathcal{X}_{\mathsf{PRF}}$ is the input, and $\mathcal{Y}_{\mathsf{PRF}}$ is the output domains.

Pseudorandom Permutation (PRP) Let $\mathsf{PRP}\colon \mathcal{K}_{\mathsf{PRP}} \times \mathcal{X}_{\mathsf{PRP}} \to \mathcal{X}_{\mathsf{PRP}}$ be a pseudorandom permutation, where $\mathcal{K}_{\mathsf{PRP}}$ is the key space and $\mathcal{X}_{\mathsf{PRP}}$ is input and output domain.

2.2 Distributed Point Functions (DPFs)

Let us formally define DPFs, which have already been discussed informally in the technical overview.

Definition 1 (Distributed Point Function (DPF)). *For any $t, m \in \mathbb{N}$, let $\mathcal{F} = \{f_{S,\mathbf{y}}\colon [m] \to \mathbb{F}\}$ be a class of (m,t)-multi-point functions with input domain $[m]$ and output domain the field $\mathbb{F}$, where $S = \{s_1, \ldots, s_t\}$ is a subset of $[m]$ of size t, and $\mathbf{y} = \{y_1, \ldots, y_t\} \in \mathbb{F}^t$, and for all $x \in [m]$,*

$$f_{S,\mathbf{y}}(x) = \begin{cases} y_j, & \text{if } x = s_j \text{ for some } j \in [t], \\ 0, & \text{otherwise.} \end{cases}$$

Let λ be the computational, and stat be the statistical parameter. A DPF scheme Φ consists of the following two algorithms:

$(k_0, k_1) \leftarrow \Phi.\mathsf{Gen}(1^\lambda, \mathsf{stat}, f)$*: Given description $f \in \mathcal{F}$, the algorithm returns two keys k_0 and k_1.*
$f_b(x) \leftarrow \Phi.\mathsf{Eval}(k_b, x)$*: Given key k_b for party $b \in \{0,1\}$ and input $x \in [m]$, return share $f_b(x)$ of $f(x)$.*

A distributed point function has to satisfy the following properties:

- ***Correctness:** For any $f \in \mathcal{F}$ and any $x \in [m]$, it holds that*

$$\Pr\left[\textstyle\sum_{b\in\{0,1\}} \Phi.\mathsf{Eval}(k_b, x) \neq f(x) \;\middle|\; (k_0, k_1) \leftarrow \Phi.\mathsf{Gen}(1^\lambda, \mathsf{stat}, f)\right] \leq 2^{-\mathsf{stat}}.$$

- ***Security:** For any $b \in \{0,1\}$, there exists a PPT simulator Sim such that for all polynomial size function sequences $f_\lambda \in \mathcal{F}$:*

$$\{k_b \mid (k_0, k_1) \leftarrow \Phi.\mathsf{Gen}(1^\lambda, \mathsf{stat}, f_\lambda)\} \stackrel{c}{\approx} \{k_b \leftarrow \mathsf{Sim}_b(1^\lambda, \mathsf{stat}, \mathbb{F})\}.$$

2.3 Secure Multiparty Computation

We assume that all parties are connected via a synchronous communication network and that all parties have access to private point-to-point channels between each other. Our protocols are secure against one passive corruption. We will formally prove security in the universal composability framework of Canetti [Can01] and we recall the corresponding formal definitions in Appendix A.1.

3 Three-Party Random OT

We first present a crucial building block that will be used in our construction for distributed RAM computation, fully presented in Section 4. Looking ahead, this building block will allow parties to obtain mask vectors that will be used to mask their shares of memory in the initialization phase (as explained in the Section 1.2). This will be done so that the communication complexity is independent of the mask vector size. This is an optimization that helps us achieve better overall efficiency during reinitialization at the end of each epoch.

Functionality $\mathcal{F}_{\mathsf{3\text{-}OT}}$ (Fig. 1) defines a three-party variant of random oblivious transfer (OT) [Bea95]. In the Init phase, the *sender* S learns a random secret vector $\boldsymbol{r}$ of length ℓ. Then, the Access phase allows the index party (or *chooser*) C to repeatedly reveal entries in $\boldsymbol{r}$ at positions of its choice to the *receiver* party R. The receiver R learns nothing about the index or the other values in $\boldsymbol{r}$.

In the Init phase of the protocol $\Pi_{\mathsf{3\text{-}OT}}$ (Fig. 2), the parties S, C reveal a masked and permuted version of a vector $\boldsymbol{r}$ to R. Both the permutation and the mask are known to C, but $\boldsymbol{r}$ is only known to S. During Access, C takes an index indx as input and selectively reveals the mask at the permuted location $\pi(\mathsf{indx})$ along with $\pi(\mathsf{indx})$ to R. Since we only reveal one permuted index in $\boldsymbol{r}$, parties can run Access multiple times, without having to repeat Init. A naive implementation of the above functionality would

Functionality $\mathcal{F}_{3\text{-OT}}$

Parameters: Parties $\mathsf{S}, \mathsf{C}, \mathsf{R}$, vector length ℓ, and output range $\mathcal{Y}$.
Init: If S is corrupt, receive (Init) from C and R, and $(\mathsf{Init}, \boldsymbol{r})$ from S, store $\boldsymbol{r}$. Else, receive (Init) from all parties, sample and store a random vector $\boldsymbol{r} \leftarrow \mathcal{Y}^{\ell}$, output $\boldsymbol{r}$ to S.
Access: For up to ℓ times, on input (Access) from S and R, and $(\mathsf{Access}, \mathsf{indx})$ from C, output $x := \boldsymbol{r}[\mathsf{indx}]$ to R.

Fig. 1: Ideal functionality for three-party random OT for three parties $\mathsf{S}, \mathsf{C}, \mathsf{R}$, which allows R to learn values in a secret random vector known to S, at secret locations known to C.

be to send a masked and permuted vector to R and then selectively provide masks to reveal positions in the vector. This solution would have a communication complexity that is linear in the size of the vector, which we want to avoid. Instead, we exploit that the vector $\boldsymbol{r}$ is supposed to be random and can be represented succinctly by a short PRF key, assuming that the PRF key is known only to S. In this work, this assumption indeed holds wherever $\Pi_{3\text{-OT}}$ is used as a sub-protocol and S is honest. In the Init phase, S defines a secret sharing of its pseudorandom vector by sampling two PRF keys k_{C}, k_{R} along with a PRP key k_{PRP} that defines a permutation π. S defines each entry j of the vector $\boldsymbol{r}$ as $\mathsf{PRF}(k_{\mathsf{C}}, \pi(j)) + \mathsf{PRF}(k_{\mathsf{R}}, \pi(j))$. Then S sends the keys $(k_{\mathsf{C}}, k_{\mathsf{PRP}})$ to C and k_{R} to R. This completes the Init phase. During Access, when C receives an index indx as input, it can locally compute the permuted index $\pi(\mathsf{indx})$ and reveal $\mathsf{PRF}(k_{\mathsf{C}}, \pi(\mathsf{indx}))$ to R. To allow R to compute $\mathsf{PRF}(k_{\mathsf{R}}, \pi(\mathsf{indx}))$, party C also sends $\pi(\mathsf{indx})$. Finally, R can locally compute $\boldsymbol{r}[\mathsf{indx}] = \mathsf{PRF}(k_{\mathsf{C}}, \pi(\mathsf{indx})) + \mathsf{PRF}(k_{\mathsf{R}}, \pi(\mathsf{indx}))$. Since, only one out of two PRF keys are revealed to C and R, they never learn the vector $\boldsymbol{r}$. For the same reason, R only learns the entries in the vector that are chosen by C and nothing more. Moreover, since only a permuted index is revealed to R, the party remains oblivious to the actual index that was being accessed. The security of $\Pi_{3\text{-OT}}$ is summarized in Theorem 1 and the security proof is deferred to Appendix A.2.

Protocol $\Pi_{3\text{-OT}}$

Parameters: A $\mathsf{PRF} : \mathcal{K} \times [\ell] \to \mathcal{Y}$, and a $\mathsf{PRP} : \mathcal{K}_{\mathsf{PRP}} \times [\ell] \to [\ell]$.
Init:
1. S samples two PRF keys $k_{\mathsf{C}}, k_{\mathsf{R}} \leftarrow \mathcal{K}$ and a PRP key $k_{\mathsf{PRP}} \leftarrow \mathcal{K}_{\mathsf{PRP}}$.
2. S sends $(k_{\mathsf{C}}, k_{\mathsf{PRP}})$ to C and k_{R} to R.
3. S defines vector $\boldsymbol{r}$ as follows. For $j \in [\ell]$,
   (a) Evaluate permuted index; $\pi(j) := \mathsf{PRP}(k_{\mathsf{PRP}}, j)$.
   (b) Set $\boldsymbol{r}[j] := \mathsf{PRF}(k_{\mathsf{C}}, \pi(j)) + \mathsf{PRF}(k_{\mathsf{R}}, \pi(j))$.
4. S outputs $\boldsymbol{r}$.

Access:
1. C receives as input indx and computes permuted index $\pi(\mathsf{indx}) := \mathsf{PRP}(k_{\mathsf{PRP}}, \mathsf{indx})$ and $a := \mathsf{PRF}(k_{\mathsf{C}}, \pi(\mathsf{indx}))$.
2. C sends $(\pi(\mathsf{indx}), a)$ to R.
3. R computes and outputs $x := \mathsf{PRF}(k_{\mathsf{R}}, \pi(\mathsf{indx})) + a$.

Fig. 2: Protocol for three-party random OT.

4 Three-Party Secure RAM Computation

In this section, we explain our three-party construction for secure RAM computation in detail. We follow a top-down approach, starting with the high level functionalities and fleshing out the details as we proceed.

The functionality that we want to achieve is $\mathcal{F}_{\mathsf{f}}$ (Fig. 3), which allows us to execute a RAM program for a function $f(x, \mathsf{M})$, where x is the input, and M is the memory of size n. Functionality $\mathcal{F}_{\mathsf{f}}$ executes the RAM program by executing NS (the next step function) iteratively until $\mathsf{st}_{\mathsf{NS}}$ outputs stop. At this point, the final output of the function evaluation is stored in $\mathsf{st}_{\mathsf{NS}}$ which is secret shared between the parties and, if needed, can be reconstructed to obtain the output of the computation. As mentioned in the technical overview, the general approach to realize $\mathcal{F}_{\mathsf{f}}$ is to execute NS using any efficient MPC protocol of choice ($\mathcal{F}_{\mathsf{MPC}}$, Fig. 24), to obtain additive shares of instruction I, and use a specialized distributed ORAM functionality ($\mathcal{F}_{\mathsf{DORAM}}$, Fig. 13) for executing I on memory M. The formal protocol description (Π_{f}, Fig. 11) and corresponding security proof (Theorem 2)appear in Appendix A.3, since they are not new to our work. Next, we elaborate how to instantiate $\mathcal{F}_{\mathsf{DORAM}}$.

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

Parameters: A next-step function NS.
Run: On receiving $(\mathsf{Run}, \langle \mathsf{M} \rangle, \langle x \rangle)$ from all parties, do:

1. Reconstruct memory M and input x.
2. Initialize $\mathsf{st}_{\mathsf{NS}} := (\mathsf{start}, x)$, $d := 0$, and $\mathsf{rnd} := 0$.
3. While $\mathsf{st}_{\mathsf{NS}} \neq (\mathsf{stop}, z)$, do:
   (a) Run $(\mathsf{st}_{\mathsf{NS}}, \mathsf{I}) \leftarrow \mathsf{NS}(\mathsf{st}_{\mathsf{NS}}, d)$.
   (b) If $\mathsf{I} = (\mathsf{read}, \mathsf{adr}, \perp)$, set $d := \mathsf{M}[\mathsf{adr}].\mathsf{val}$.
   (c) Else if $\mathsf{I} = (\mathsf{write}, \mathsf{adr}, \mathsf{val})$, set $d := \mathsf{M}[\mathsf{adr}].\mathsf{val}$ and update $\mathsf{M}[\mathsf{adr}].\mathsf{val} := \mathsf{I}.\mathsf{val}$.
   (d) Update $\mathsf{rnd} := \mathsf{rnd} + 1$.
4. Output additive shares $\langle \mathsf{st}_{\mathsf{NS}} \rangle$, $\langle \mathsf{M} \rangle$, rnd to all parties.

Fig. 3: Ideal functionality for secure evaluation of a RAM program f between three parties P_1, P_2, P_3.

4.1 Instantiating $\mathcal{F}_{\mathsf{DORAM}}$

The functionality $\mathcal{F}_{\mathsf{DORAM}}$ (Fig. 13, Appendix A.4) receives additive shares of M in the Init phase, reconstructs and stores M locally. Upon receiving additive shares of instruction I in the Access phase, $\mathcal{F}_{\mathsf{DORAM}}$ reconstructs and executes I locally on M, then outputs additive shares of the result. Finally, $\mathcal{F}_{\mathsf{DORAM}}$ can output the current state of M in secret-shared form to the parties. Our starting point for instantiating this is the basic square-root ORAM construction.

As explained in Section 1.2, intuitively, M is used as a read-only memory while the writes are stashed. For this purpose, at the start of an epoch, in the $\mathsf{Refresh}$ phase, the parties initialize an empty stash of size $\sqrt{n}$, append $\sqrt{n}$ dummy values to their shares of M, and initialize the data-structure needed to perform read operations on M. In each iteration of the Access phase, parties receive additive shares of the instruction $\mathsf{I} = (\mathsf{op}, \mathsf{adr}, \mathsf{val})$ to be executed. To hide the type of operation, for both read and write operations, parties execute the following steps in sequence: 1. Check stash for the most up-to-date value for address adr. 2. Read a value from M. 3. Write updates in stash. 4. Once the stash is full, i.e., after $\sqrt{n}$ iterations, update M and reinitialize the setup for the next epoch.

The formal description of our protocol Π_{DORAM} (Fig. 14)and its security proof (Theorem 3)appear in Appendix A.4. For sake of modularity, it is described in terms of calls to functionalities $\mathcal{F}_{\mathsf{Stash}}$, $\mathcal{F}_{\mathsf{r\text{-}M}}$, and $\mathcal{F}_{\mathsf{w\text{-}M}}$ which handle the main complexity of our construction, and whose instantiation is explained in the next subsections.

We now discuss the four steps of Π_{DORAM} in more detail: Throughout the protocol execution we maintain the invariant that, if adr was previously read, then the most recent value can be found in the stash, otherwise, it is in the memory M. Hence, in Step 1 the stash is first searched for adr by calling $\mathcal{F}_{\mathsf{Stash}}$(Fig. 16, Appendix A.5). Then in Step 2, using $\mathcal{F}_{\mathsf{r\text{-}M}}$(Fig. 18, Appendix A.6), M is searched for *some* address, which is determined as follows: If adr was previously read during the epoch, then parties read a fresh dummy address from M in Step 2, otherwise they read adr. The dummy address for iteration c is deterministically set to be $n + c$. In both steps, depending on whether or not adr was previously accessed, parties either receive secret shares of the most up-to-date value or a dummy value.

In Step 3, the stash must be updated in two ways. In each iteration, parties write to the stash the value they read from M in Step 2, using $\mathcal{F}_{\mathsf{Stash}}$. This update is always made at location c, for iteration c. A second update must be made, which corresponds to the case when $\mathsf{op} = \mathsf{write}$. If adr is a repeat address, the stash is updated at loc, where the most recent value of adr is stored. If adr is new, it is updated at c. In order to hide the type of operation, this is done even for $\mathsf{op} = \mathsf{read}$ but the update value is set to 0. For privacy in Steps 2 and 3, choices dependent on whether or not adr was read previously and whether $\mathsf{op} = \mathsf{read}$ or write need to be made obliviously. This is done using a generic MPC instantiation of $\mathcal{F}_{\mathsf{Select}}$(Fig. 23, Appendix C).

The final Step 4 of each epoch consists of writing the stashed updates from the stash back into M using $\mathcal{F}_{\mathsf{w\text{-}M}}$(Fig. 20, Appendix A.7), and reinitializing the setup by with the $\mathsf{Refresh}$ protocol.

4.2 Instantiating $\mathcal{F}_{\mathsf{Stash}}$

The functionality $\mathcal{F}_{\mathsf{Stash}}$ (Fig. 16, Appendix A.6)for accessing the stash is instantiated by Π_{stash}, which consists of two parts.

The protocol for the Read command is given in Fig. 4. Parties start with an additive sharing of stash as well as the instruction I and want to compute secret shares of $(\mathsf{flag}, \mathsf{loc}, \mathsf{val}_{\mathsf{st}})$. As previously explained, to obtain $\langle\mathsf{flag}\rangle, \langle\mathsf{loc}\rangle$, parties P_2, P_3 maintain a list L_{stash} of PRF evaluations of all the addresses that have been read from the memory in the current epoch. The key for this PRF is held by P_1. Using $\mathcal{F}_{\mathsf{DOPRF}}$, they reveal the masked address adr' to P_1, who generates DPF keys with respect to adr'. Parties P_2 and P_3, using their knowledge of the mask, evaluate the keys on all entries present in L_{stash}, obtaining $\langle\mathsf{flag}\rangle$ and $\langle\mathsf{loc}\rangle$ as a result. In case adr was not present in L_{stash}, we want to set loc to c, the current iteration counter. This is achieved via one call to $\mathcal{F}_{\mathsf{Select}}$, to obliviously select between the computed loc and c. Finally, in order to establish $\langle\mathsf{stash}[\mathsf{loc}].\mathsf{val}\rangle$, parties first form a replicated sharing of stash to maintain the invariant that each share of stash is possessed by two parties. This is necessary for reading using DPF keys. Then using similar tricks as before, parties reveal the masked location loc' to one of the parties who generates DPF keys required to compute $\langle\mathsf{stash}[\mathsf{loc}].\mathsf{val}\rangle$.

The part of Π_{stash} for the Write command is shown in Fig. 5. Before writing a value to the stash, parties first update the list L_{stash} by opening the PRF evaluation of the address read, denoted by $\mathsf{adr}_{\mathsf{M}}$, towards P_2, P_3 who append it to L_{stash}. Then, all of the parties update stash at location c by appending $(\langle\mathsf{adr}_{\mathsf{M}}\rangle, \langle\mathsf{val}_{\mathsf{M}}\rangle, \langle\mathsf{val}_{\mathsf{M}}\rangle)$ to it. Finally, the stash must be updated once more with respect to op. Recall in Step 3 of "Instantiating $\mathcal{F}_{\mathsf{DORAM}}$", we have the following possible cases, depending on flag and op. If $\mathsf{op} = \mathsf{write}$, then $\mathsf{stash}[\mathsf{loc}].\mathsf{val}$ should be updated to $\mathsf{val}_{\mathsf{I}}$, where $\mathsf{val}_{\mathsf{I}}$ is the value in the instruction. Moreover, if $\mathsf{op} = \mathsf{read}$, then $\mathsf{stash}[\mathsf{loc}].\mathsf{val}$ should remain unchanged, i.e., the update $\Delta := 0$. These choices in order to set Δ are executed by two calls to $\mathcal{F}_{\mathsf{Select}}$. Once Δ is obtained, the stash is updated using DPF keys and techniques similar to what has been previously discussed.

4.3 Instantiating $\mathcal{F}_{\mathsf{r\text{-}M}}$

The functionality $\mathcal{F}_{\mathsf{r\text{-}M}}$ (Fig. 18, Appendix A.6)receives additive shares of address to be read and outputs shares of the memory value at that address. Note that this address is either a dummy or the actual address in the instruction I, and hence might differ from the address mentioned in I. The high level idea to instantiate the functionality was already discussed in the technical overview. Here, we fill in the missing details.

During Init, for $i \in [3]$, P_{i-1} defines a vector $\boldsymbol{r}_i$ using an Init call to $\mathcal{F}_{\mathsf{3\text{-}OT}}$. It also picks a PRF key k_i along with P_{i+1}. Then P_{i-1} sends the masked and permuted memory share M'_i consisting of tuples $(\mathsf{PRF}(k_i, j), \mathsf{M}_{i-1}[j] + \boldsymbol{r}_i[\mathsf{PRF}(k_i, j)])$ for $j \in [m]$, sorted by their first component, to P_i.

In Access, the value $\mathsf{adr}' = \mathsf{PRF}(k_i, \mathsf{adr})$ is revealed to P_i via one oblivious PRF evaluation. P_i locally retrieves the value at $\mathsf{M}'[\mathsf{adr}']$ which is equal to $\mathsf{M}_{i-1}[\mathsf{adr}].\mathsf{val} + \boldsymbol{r}_i[\mathsf{adr}']$. To form an additive sharing between P_i and P_{i+1}, we must reveal the mask at the index adr' to P_{i+1}. At this point, notice that, P_{i-1} holds the vector $\boldsymbol{r}_i$, P_i holds the index adr', and P_{i+1} should learn the mask $\boldsymbol{r}_i[\mathsf{adr}']$. A call to $\mathcal{F}_{\mathsf{3\text{-}OT}}(\mathsf{Access})$ allows us to do exactly this. One execution of the above procedure results in additive shares of $\langle\mathsf{M}[\mathsf{adr}].\mathsf{val}\rangle$ held by P_i, P_{i+1}. Repeating this procedure thrice for each party's share of M, gives us additive shares of $\mathsf{M}[\mathsf{adr}].\mathsf{val}$.

Observe that since the shuffling of $\langle\mathsf{M}\rangle$ mentioned above is performed by sorting PRF evaluations of the addresses, M'_i consists of tuples of the form $(\mathsf{indx}', \mathsf{val}')$, where $\mathsf{indx}' \in \mathcal{Y}_{\mathsf{PRF}}$. For correctness, we require $\mathcal{Y}_{\mathsf{PRF}}$ to be big enough in order to avoid any collisions, which means that it can be much bigger than m. Naively implementing $\boldsymbol{r}_i$ would then require making it as big as the size of $\mathcal{Y}_{\mathsf{PRF}}$. However, this can be prevented by leveraging the fact that the sorted list $\{\mathsf{PRF}(k_i, \mathsf{indx})\}_{\{\mathsf{indx} \in [m]\}}$ is public, as k_i is known to P_{i-1}, P_{i+1} and the list is revealed to P_i, and that there are no collisions in PRF evaluations. Let the sorted list be denoted by L_i. Now the parties define an inverse map denoted by L_i^{-1} and defined

Protocol Π_{stash} (Part I: Reading)

Parameters: Stash size $\sqrt{n}$, memory size n. A $\mathsf{PRF}\colon \mathcal{K} \times [m] \to \mathcal{Y}$. Three DPF schemes: Φ_1 for functions $\mathcal{Y} \to \{0,1\}$, Φ_2 for $\mathbb{F}_p \to \{0,1\}$, and Φ_3 for $\mathbb{F}_p \to \mathbb{F}_p$.
Read in Stash: Each party P_i has input $(c, \langle \mathsf{stash} \rangle, \langle \mathsf{I} \rangle)$. Let $\langle \mathsf{adr} \rangle := \langle \mathsf{I.adr} \rangle$.

Compute flag and possible stash index loc:

1. If $c = 1$, parties call $\mathcal{F}_{\mathsf{DOPRF}}$ with (KeyGen, P_1), where P_1 acts as K and receives a PRF key k_{stash}. P_2, P_3 act as R_1 and R_2, respectively, and initialize an empty list L_{stash}.
2. Parties initialize $\langle \mathsf{flag} \rangle := 0$ and $\langle \mathsf{loc} \rangle := 0$.
3. P_2, P_3 call $\mathcal{F}_{\mathsf{DOPRF}}$ with $(\mathsf{Eval}, \mathsf{masked}, \langle \mathsf{adr} \rangle)$[a], and P_1 calls $\mathcal{F}_{\mathsf{DOPRF}}$ with $(\mathsf{Eval}, \mathsf{masked}, \langle \mathsf{adr} \rangle, k_{\mathsf{stash}})$. P_1 receives $\mathsf{adr}' := \mathsf{PRF}(k_{\mathsf{stash}}, \mathsf{adr}) \oplus R$, and P_2, P_3 receive R.
4. P_1 defines $f\colon \mathcal{Y} \to \{0,1\}$ such that $f(j) := 1$ if $j = \mathsf{adr}'$. It generates DPF keys $(k_2, k_3) \leftarrow \Phi_1.\mathsf{Gen}(1^\lambda, \mathsf{stat}, f)$ and sends them to P_2 and P_3.
5. Party P_i for $i \in [2,3]$ compute for $j \in [|L_{\mathsf{stash}}|]$: $\langle \mathsf{flag} \rangle := \langle \mathsf{flag} \rangle + \Phi_1.\mathsf{Eval}(k_i, L_{\mathsf{stash}}[j] \oplus R)$, and $\langle \mathsf{loc} \rangle := \langle \mathsf{loc} \rangle + \Phi_1.\mathsf{Eval}(k_i, L_{\mathsf{stash}}[j] \oplus R) \cdot j$.
6. Each party P_i calls $\langle \mathsf{loc} \rangle \leftarrow \mathcal{F}_{\mathsf{Select}}(\langle \mathsf{flag} \rangle, \langle \mathsf{loc} \rangle, c)$.
7. Randomize additive shares: Each party P_i calls $\mathcal{F}_{\mathsf{Zero}}(1)$ to receive $\langle 0 \rangle$ and sets $\langle \mathsf{flag} \rangle := \langle \mathsf{flag} \rangle + \langle 0 \rangle$.

Read the stashed value $\mathsf{stash}[\mathsf{loc}].\mathsf{val}$:

8. Convert $\langle \mathsf{stash.val} \rangle$ to $[\![\mathsf{stash.val}]\!]$: For $i \in \{1,2,3\}$, and for $j \in [|\mathsf{stash}|]$, each P_i sends $\langle \mathsf{stash}[j].\mathsf{val} \rangle_i$ to P_{i+1}.
9. Each party initializes $\langle \mathsf{val}_{\mathsf{st}} \rangle := 0$.
10. For $i \in \{1,2,3\}$, do:
    (a) P_{i-1} samples $r_{i-1,i}, r_{i+1,i} \leftarrow \mathbb{F}_p^2$ and sends them to P_{i+1}. Both P_{i-1} and P_{i+1} set $r_i := r_{i-1,i} + r_{i+1,i}$.
    (b) P_{i-1} sends $\langle \mathsf{loc} \rangle + r_{i-1,i}$ and P_{i+1} sends $\langle \mathsf{loc} \rangle + r_{i+1,i}$ to P_i. P_i reconstructs $\mathsf{loc}_i := \mathsf{loc} + r_i$.
    (c) P_i defines $f_i\colon \mathbb{F}_p \to \{0,1\}$ such that $f_i(j) := 1$ if $j = \mathsf{loc}_i$. It generates DPF keys $(k_{i,i-1}, k_{i,i+1}) \leftarrow \Phi_2.\mathsf{Gen}(1^\lambda, \mathsf{stat}, f_i)$. P_i sends $k_{i,i-1}, k_{i,i+1}$ to P_{i-1} and P_{i+1}, respectively.
    (d) For $j \in [|\mathsf{stash}|]$, P_{i-1} computes $\langle \mathsf{val}_{\mathsf{st}} \rangle := \langle \mathsf{val}_{\mathsf{st}} \rangle + \Phi_2.\mathsf{Eval}(k_{i,i-1}, j + r_i) \cdot \langle \mathsf{stash}[j].\mathsf{val} \rangle_{i+1}$. Similarly, P_{i+1} computes $\langle \mathsf{val}_{\mathsf{st}} \rangle := \langle \mathsf{val}_{\mathsf{st}} \rangle + \Phi_2.\mathsf{Eval}(k_{i,i+1}, j + r_i) \cdot \langle \mathsf{stash}[j].\mathsf{val} \rangle_{i+1}$.
11. Randomize additive shares: Each party P_i calls $\mathcal{F}_{\mathsf{Zero}}(1)$ to receive $\langle 0 \rangle$ and sets $\langle \mathsf{val}_{\mathsf{st}} \rangle := \langle \mathsf{val}_{\mathsf{st}} \rangle + \langle 0 \rangle$.
12. Each P_i outputs $\langle \mathsf{st}_{\mathsf{stash}} \rangle := (\langle \mathsf{flag} \rangle, \langle \mathsf{loc} \rangle, \langle \mathsf{val}_{\mathsf{st}} \rangle)$.

[a] We use two types of $\mathcal{F}_{\mathsf{DOPRF}}$ evaluation: (i) masked allows the parties to get secret shares of the output; (ii) unmasked allows one of the parties to get the output in clear.

Fig. 4: Protocol for reading from stash (see Fig. 5 for writing).

as the position at which $\mathsf{PRF}(\mathsf{indx})$ lies in L_i, i.e. for $j \in [m]$, the element $L_i[j]$ maps to j. Given L_i^{-1} they can uniquely map the PRF output to a value in $[m]$.

The formal protocol description of $\Pi_{\mathsf{r\text{-}M}}$ appears in Fig. 6, while the functionality ($\mathcal{F}_{\mathsf{r\text{-}M}}$, Fig. 18) and the security proof (Theorem 5) are deferred to Appendix A.

4.4 Instantiating $\mathcal{F}_{\mathsf{w\text{-}M}}$

The functionality $\mathcal{F}_{\mathsf{w\text{-}M}}$ (Fig. 20, Appendix A)flushes the stash and updates the memory. The main idea of this has already been discussed in the technical overview 1.2. As in the case of $\Pi_{\mathsf{r\text{-}M}}$, here too, we reduce communication and computation required for generating DPF keys by leveraging the map L_i^{-1}, using which, P_i can generate DPF keys for a much smaller input domain $[m]$ instead of $\mathcal{Y}_{\mathsf{PRF}}$. The formal protocol description appears in $\Pi_{\mathsf{w\text{-}M}}$ (Fig. 7). The proof of security is deferred to Theorem 6 in Appendix A.

4.5 Instantiating $\mathcal{F}_{\mathsf{DOPRF}}$

We use the Legendre PRF to instantiate the $\mathcal{F}_{\mathsf{DOPRF}}$ functionality [Dam90]. A single bit Legendre PRF, $f_{\mathsf{PRF}}\colon \mathbb{F}_p \times \mathbb{F}_p \to \{0,1\}$, where p is a public prime is defined as follows:

$$f_{\mathsf{PRF}}(k, x) = L_p(k + x) \quad \text{with} \quad L_p(a) = \frac{1}{2}\left(\left(\frac{a}{p}\right) + 1\right) \bmod p.$$

Protocol Π_{stash} (Part II: Writing)

Write in Stash: Each party has input $\langle \mathsf{val_M} \rangle, \langle \mathsf{adr_M} \rangle, c, \langle \mathsf{stash} \rangle, \langle \mathsf{st_{stash}} \rangle, \langle \mathsf{I} \rangle$.

1. P_2, P_3 call $\mathcal{F}_{\mathsf{DOPRF}}$ with $(\mathsf{Eval}, \mathsf{unmasked}, \langle \mathsf{adr_M} \rangle)$, and P_1 calls it with $(\mathsf{Eval}, \mathsf{unmasked}, \langle \mathsf{adr} \rangle, k_{\mathsf{stash}})$ acting as K. P_2, acting as R_2, receives $\mathsf{PRF}(k_{\mathsf{stash}}, \mathsf{adr})$.
2. P_2 sends $\mathsf{PRF}(k_{\mathsf{stash}}, \mathsf{adr})$ to P_3 and they both append list $L_{\mathsf{stash}} := L_{\mathsf{stash}} \,||\, \mathsf{PRF}(k_{\mathsf{stash}}, \mathsf{adr_M})$.
3. Each party P_i locally updates its share $\langle \mathsf{stash} \rangle$ at location c as: $\langle \mathsf{stash}[c] \rangle := (\langle \mathsf{adr_M} \rangle, \langle \mathsf{val_M} \rangle, \langle \mathsf{val_M} \rangle)$.
4. Parties parse $\langle \mathsf{st_{stash}} \rangle$ as $(\langle \mathsf{flag} \rangle, \langle \mathsf{loc} \rangle, \langle \mathsf{val_{st}} \rangle)$, and $\langle \mathsf{I} \rangle$ as $(\langle \mathsf{op_I} \rangle, \langle \mathsf{adr_I} \rangle, \langle \mathsf{val_I} \rangle)$.
5. Parties call $\langle \mathsf{val_{old}} \rangle \leftarrow \mathcal{F}_{\mathsf{Select}}(\langle \mathsf{flag} \rangle, \langle \mathsf{val_{st}} \rangle, \langle \mathsf{val_M} \rangle)$ and $\langle \mathsf{val_{new}} \rangle \leftarrow \mathcal{F}_{\mathsf{Select}}(\langle \mathsf{op_I} \rangle, \langle \mathsf{val_I} \rangle, \langle \mathsf{val_{old}} \rangle)$.
6. Each P_i computes $\langle \Delta \rangle := \langle \mathsf{val_{new}} \rangle - \langle \mathsf{val_{old}} \rangle$ and initializes a vector $\boldsymbol{\delta}_i$ of length $|\mathsf{stash}|$ as $(0, \ldots, 0)$.
7. For $i \in \{1, 2, 3\}$, do:
   (a) P_{i-1} samples $\rho_{i-1,i}, \rho_{i+1,i} \leftarrow \mathbb{F}_p^2$, and sends them to P_{i+1}. Both P_{i-1} and P_{i+1} compute $\rho_i := \rho_{i-1,i} + \rho_{i+1,i}$.
   (b) P_{i-1} sends $\langle \mathsf{loc} \rangle + \rho_{i-1,i}$, and P_{i+1} sends $\langle \mathsf{loc} \rangle + \rho_{i+1,i}$ to P_i.
   (c) P_i reconstructs $\mathsf{loc}_i := \mathsf{loc} + \rho_i$.
   (d) P_i defines $f_i : \mathbb{F}_p \to \mathbb{F}_p$, such that $f_i(j) := \langle \Delta \rangle_i$ at $j = \mathsf{loc}_i$. It generates DPF keys $(k_{i,i-1}, k_{i,i+1}) \leftarrow \Phi_3.\mathsf{Gen}(1^\lambda, \mathsf{stat}, f_i)$. P_i sends $k_{i,i-1}, k_{i,i+1}$ to P_{i-1} and P_{i+1}, respectively.
   (e) For $j \in [|\mathsf{stash}|]$, P_{i-1} updates $\boldsymbol{\delta}_{i-1}[j] := \boldsymbol{\delta}_{i-1}[j] + \Phi_3.\mathsf{Eval}(k_{i,i-1}, j + \rho_i)$, and P_{i+1} updates $\boldsymbol{\delta}_{i+1}[j] := \boldsymbol{\delta}_{i+1}[j] + \Phi_3.\mathsf{Eval}(k_{i,i+1}, j + \rho_i)$.
8. All parties call $\langle \mathbf{0}_1 \rangle \leftarrow \mathcal{F}_{\mathsf{Zero}}(|\mathsf{stash}|)$.
9. For $j \in [|\mathsf{stash}|]$, each party P_i updates $\langle \mathsf{stash}[j].\mathsf{val} \rangle := \langle \mathsf{stash}[j].\mathsf{val} \rangle + \boldsymbol{\delta}_i[j] + \langle \mathbf{0}_1[j] \rangle$.
10. All parties call $\langle \mathbf{0}_2 \rangle \leftarrow \mathcal{F}_{\mathsf{Zero}}(|\mathsf{stash}|)$ and $\langle \mathbf{0}_3 \rangle \leftarrow \mathcal{F}_{\mathsf{Zero}}(|\mathsf{stash}|)$[a].
11. For $j \in [|\mathsf{stash}|]$, each party P_i updates $\langle \mathsf{stash}[j].\mathsf{adr} \rangle := \langle \mathsf{stash}[j].\mathsf{adr} \rangle + \langle \mathbf{0}_2[j] \rangle$ and $\langle \mathsf{stash}[j].\mathsf{val_{old}} \rangle := \langle \mathsf{stash}[j].\mathsf{val_{old}} \rangle + \langle \mathbf{0}_3[j] \rangle$.
12. Parties output $\langle \mathsf{stash} \rangle$. In addition, P_2, P_3 output L_{stash}.

[a] steps 10 and 11 can be avoided with a minor change in the $\mathcal{F}_{\mathsf{Stash}}$ functionality.

Fig. 5: Protocol for writing to the stash.

Here, $\left(\frac{a}{p}\right)$ denotes the Legendre symbol, i.e., it evaluates to 1 if a is non-zero and a quadratic residue modulo p, 0 if $a \equiv 0 \pmod p$, and -1 otherwise. $L_p(\cdot)$ simply maps this value to $\{0, 1, (p+1)/2\}$. To extend the range to an ℓ bit output, we select ℓ independent keys and replicate the computation for a single bit output ℓ times. More specifically, for an ℓ bit output, we pick a vector of keys of size ℓ, denoted as $\boldsymbol{k}$, and define $\mathsf{PRF}(\boldsymbol{k}, x)$ as:

$$\mathsf{PRF}(\boldsymbol{k}, x) = \left(\sum_{i=0}^{\ell-1} 2^i \cdot L_p(k_i + x)\right) \bmod p.$$

We optimize the implementation of our protocol by using two different types of $\mathcal{F}_{\mathsf{DOPRF}}$ evaluations, which we call *the unmasked* and *the masked* type, respectively. In both types, one party K holds the PRF key k. In the unmasked type, some other receiver party R_2 learns, in clear, the PRF evaluation of an additively shared value x. On the other hand, in the masked type, we require all parties to learn secret shares of the evaluation. Specifically, two of the parties $(\mathsf{R}_1, \mathsf{R}_2)$ learn a random bit r, while K learns $f(k, x) \oplus r$. The former type performs slightly better as it requires two less rounds of communication compared to the latter. We use it to obtain PRF evaluations in the write phase of Π_{stash}, and for all PRF evaluations in $\Pi_{\mathsf{r\text{-}M}}$. The masked type is used to obtain PRF evaluations in the read phase of Π_{stash}.

The functionality $\mathcal{F}_{\mathsf{DOPRF}}$ can be called in two modes corresponding to the two types, and it appears in Fig. 8. Next, we explain both our constructions. Both the protocols are inspired by the analogous protocol in [GRR+16], which given additive shares of x, allows all parties to learn additive shares of $f(k, x)$. Here however, the protocols are modified and optimized for our specific and different use case of $\mathcal{F}_{\mathsf{DOPRF}}$. For completeness, we present the masked version Π^m_{DOPRF} here in Fig. 9, while the unmasked version Π_{DOPRF} is deferred to Appendix B in Fig. 22. Given Π^m_{DOPRF}, parties can always obtain outputs consistent with the unmasked mode by simply reconstructing the shares towards one party.

Both the protocols exploit multiplicativity of Legendre symbols. In the unmasked protocol, the main idea is to mask $k + x$ with a random square s^2 and open $s^2(k + x)$ to R_1. Since, s^2 is a quadratic residue modulo p, it holds that $L_p(s^2(k + x)) = L_p(k + x)$ due to the multiplicativity of Legendre symbols. Thus, K can locally compute the PRF evaluation. For the masked version, the idea is similar. We reveal either

Protocol $\Pi_{\text{r-M}}$

Parameters: PRF $\mathsf{PRF}\colon \mathcal{K} \times [m] \rightarrow \mathcal{Y}$.
Init: Each party P_i inputs its share $\langle \mathsf{M} \rangle$ which is interpreted as a list of tuples $(\mathsf{indx}, \langle \mathsf{val} \rangle)$, where $\mathsf{indx} \in [m]$. For $i \in \{1,2,3\}$,

1. All parties call $\mathcal{F}_{\mathsf{DOPRF}}$ with (KGen, P_{i-1}) with P_{i-1} acting as K. P_{i-1} receives k_i.
2. P_{i-1} sends k_i to P_{i+1}.
3. P_{i-1}, P_{i+1} locally compute a list $L_i[j] := \mathsf{PRF}(k_i, j)$, for $j \in [m]$ and sort it.
4. P_{i-1} calls $\mathcal{F}_{\mathsf{3\text{-}OT}}$ with (Init). P_i, P_{i+1} call $\mathcal{F}_{\mathsf{3\text{-}OT}}$ with Init. P_{i-1} receives mask vector as $\boldsymbol{r}_i$.
5. P_{i-1} computes:
   (a) For $j \in [m]$, set $\mathsf{M}'_i[j].\mathsf{indx} := \mathsf{PRF}(k_i, j)$, and $\mathsf{M}'_i[j].\mathsf{val} := \langle \mathsf{M}[j].\mathsf{val} \rangle + \boldsymbol{r}_i[L_i^{-1}(\mathsf{PRF}(k_i, j))]$.
   (b) Sort tuples in M'_i lexicographically with respect to $\mathsf{M}'_i.\mathsf{indx}$.
   (c) Send M'_i to P_i.
6. P_i receives M'_i from P_{i-1}.
7. P_i locally computes list $L_i[j] := \mathsf{M}'_i[j].\mathsf{indx}$, for $j \in [m]$.
8. P_i initializes empty list $\mathsf{adr}_i^{\mathsf{read}}$.
9. Output: P_i outputs k_{i-1}, k_{i+1}, L_i.

Access: Each party P_i inputs additive share $\langle \mathsf{adr}_\mathsf{M} \rangle$, the set $\mathsf{adr}_i^{\mathsf{read}}$ and keys k_{i-1}, k_{i+1}.

1. Each party P_i initializes $\langle v \rangle := 0$.
2. For $i \in \{1,2,3\}$, do:
   (a) P_{i-1} calls $\mathcal{F}_{\mathsf{DOPRF}}$ with inputs $(\mathsf{Eval}, \mathsf{unmasked}, \langle \mathsf{adr}_\mathsf{M} \rangle, k_i)$, and P_i, P_{i+1} call with input $(\mathsf{Eval}, \mathsf{unmasked}, \langle \mathsf{adr}_\mathsf{M} \rangle)$, with P_i acting as R_2. P_i receives $\mathsf{adr}' := \mathsf{PRF}(k_i, \mathsf{adr}_\mathsf{M})$.
   (b) P_i updates $\mathsf{adr}_i^{\mathsf{read}} := \mathsf{adr}_i^{\mathsf{read}} || \mathsf{adr}'$.
   (c) P_i computes $j' := L_i^{-1}(\mathsf{adr}')$.
   (d) P_i sets $\langle x \rangle := \mathsf{M}'_i[j'].\mathsf{val}$ and P_{i-1} sets $\langle x \rangle := 0$.
   (e) P_i calls $\mathcal{F}_{\mathsf{3\text{-}OT}}$ with (Access, j'), and P_{i-1}, P_{i+1} call with Access. P_{i+1} receives $\boldsymbol{r}_i[j']$, and sets $\langle x \rangle := -\boldsymbol{r}_i[j']$.
   (f) Each party computes: $\langle v \rangle := \langle v \rangle + \langle x \rangle$.
3. Output: Each party P_i outputs $(\langle v \rangle, \mathsf{adr}_i^{\mathsf{read}})$.

Fig. 6: Protocol for reading from memory.

Protocol $\Pi_{\text{w-M}}$

Parameters: $\Phi = (\mathsf{Gen}, \mathsf{Eval})$ is a multi-point DPF scheme for functions $[m] \rightarrow \mathbb{F}_p$, and a function PRF. Let $m = n + \sqrt{n}$, where n is the size of M.
Update Memory: Each party P_i, for $i \in \{1,2,3\}$ has input $\langle \mathsf{M} \rangle, L_i$ and $\langle \mathsf{stash} \rangle$. P_i also inputs auxiliary information $\mathbf{adr}_i^{\mathsf{read}}$, and two PRF keys (k_{i-1}, k_{i+1}). M is interpreted as set of tuples $(\mathsf{indx}, \mathsf{val})$.

1. For $i \in \{1,2,3\}$:
   (a) P_{i-1}, P_{i+1} locally compute a list $L_i[j] := \mathsf{PRF}(k_i, j)$, for $j \in [m]$ and sort it.
   (b) P_i locally defines: for $j \in [m]$, set $L_i^{-1}(L_i[j]) := j$.
   (c) P_i defines a multi-point function $f_i\colon [m] \rightarrow \mathbb{F}_p$ such that, for $j \in [\sqrt{n}]$, $f_i(L_i^{-1}(\mathbf{adr}_i^{\mathsf{read}}[j])) := \langle \mathsf{stash}[j].\mathsf{val} \rangle - \langle \mathsf{stash}[j].\mathsf{val}_{\mathsf{old}} \rangle$. It generates multi-point DPF keys $(mk_{i,i-1}, mk_{i,i+1}) \leftarrow \Phi.\mathsf{Gen}(1^\lambda, \mathsf{stat}, f_i)$ and sends them to P_{i-1} and P_{i+1} respectively.
   (d) For $t \in \{1,2,3\} \setminus \{i\}$: P_t locally updates $\langle \mathsf{M} \rangle_t$ as: For $j \in [n]$, $\langle \mathsf{M}[j].\mathsf{val} \rangle := \langle \mathsf{M}[j].\mathsf{val} \rangle + \Phi.\mathsf{Eval}(mk_{i,t}, L_i^{-1}(\mathsf{PRF}(k_i, j)))$
2. All parties call $\mathcal{F}_{\mathsf{Zero}}(n)$ to receive $\langle \mathbf{0} \rangle$.
3. Parties locally compute $\langle \mathsf{M} \rangle := \langle \mathsf{M} \rangle + \langle \mathbf{0} \rangle$.

Fig. 7: Protocol for updating memory with stash entries.

$s^2(k+x)$ or $\alpha \cdot s^2(k+x)$ depending on whether $r = 0$ or 1, where α is a random non-quadratic residue modulo p. Thus when $r = 0$, K locally computes $L_p(s^2(k+x)) = L_p(k+x)$ (from multiplicativity), which is equal to $f(k,x) \oplus r$. On the other hand, when $r = 1$, we get two cases: (i) if $(k+x)$ is not a square, then $\alpha \cdot s^2(k+x)$ is, and, (ii) if $(k+x)$ is a square, then $\alpha \cdot s^2(k+x)$ is not. Observe that in both the cases for $r = 1$, K obtains $f(k,x) \oplus r$. The detailed proof of security is deferred to Appendix B.

Both our protocols can be further optimized (as we do in our implementation) to precompute input independent values. This includes generation of PRG seeds, i.e., Steps 1–5 in Π^m_{DOPRF}, and Steps 1–3 in Π_{DOPRF} (Fig. 22). . Moreover, the PRG seeds and the value α can be set up once and for all. α can be reused in all Π^m_{DOPRF} instantiations as it is always masked by $\boldsymbol{s}$ which is sampled freshly in each instantiation.

Protocol Specific Notations. $\boldsymbol{x} \circ \boldsymbol{y}$ denotes element-wise product. $b \cdot \boldsymbol{a}$, where b is a scalar, denotes the product of b with each element in vector $\boldsymbol{a}$. Wherever $+$ or $-$ operations are between two vectors, it denotes element-wise addition or subtraction. Wherever, there is a $+$ or $-$ operation between a vector $\boldsymbol{x}$ and a scalar a, for example, $(\boldsymbol{x} + a)$, it denotes addition of a to each element in $\boldsymbol{x}$.

Collision Resistance. In our application, we will evaluate PRF function on a small input domain of size $m = n + \sqrt{n}$. For correctness of our scheme, we require that the evaluations should be unique for this input domain, i.e., there are no collisions with high probability. To ensure that collisions happen with probability at most $2^{-\mathsf{stat}}$, where stat is the statistical security parameter, we set the parameter $\ell := 2\log_2(m) + \mathsf{stat}$. When we use the DOPRF in the stash protocol, where the domain size is $\sqrt{n}$, we set $\ell := 2\log_2(\sqrt{n}) + \mathsf{stat}$, accordingly.

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

Parameters: Output length ℓ, a $\mathsf{PRF}\colon \mathcal{K} \times \mathcal{X} \rightarrow \{0,1\}^\ell$.
KeyGen: On receiving (KeyGen, P_i) from all parties, treat P_i as K. Randomly sample $\boldsymbol{k} \leftarrow \mathcal{K}$. Output $\boldsymbol{k}$ to K.
Eval: On receiving $(\mathsf{Eval}, \mathsf{mode}, \langle x \rangle)$ from all parties and receive $\boldsymbol{k}$ from K. Do:

1. Reconstruct x. Compute $\boldsymbol{y} := \mathsf{PRF}(k, x)$.
2. Sample $\boldsymbol{r} \leftarrow \{0,1\}^\ell$.
3. If $\mathsf{mode} = \mathsf{masked}$, output $\boldsymbol{r}$ to $\mathsf{R}_1, \mathsf{R}_2$, and $\boldsymbol{y} \oplus \boldsymbol{r}$ to K.
4. Else, $\mathsf{mode} = \mathsf{unmasked}$ output $\boldsymbol{y}$ to R_2.

Fig. 8: Functionality for a distributed PRF evaluation between parties $\mathsf{R}_1, \mathsf{R}_2, \mathsf{K}$ where K holds the key. In $\mathsf{mode} = \mathsf{unmasked}$ it allows R_2 to obtain PRF evaluation in clear, and in $\mathsf{mode} = \mathsf{masked}$ it allows all parties to obtain secret sharing of the PRF evaluation.

4.6 Cost Analysis

Here, we calculate the overall communication and computation cost for executing Π_{f}. For the sake of analysis, assume that the number of instructions required to execute Π_{f} is $t \cdot \sqrt{n}$, i.e., we have t epochs. Π_{f} makes one call to $\mathcal{F}_{\mathsf{DORAM}}(\mathsf{Init})$ and $\mathcal{F}_{\mathsf{DORAM}}(\mathsf{GetM})$ once per execution. Per instruction, Π_{f} makes one call to $\mathcal{F}_{\mathsf{MPC}}$ and $\mathcal{F}_{\mathsf{DORAM}}(\mathsf{Access})$. When instantiated, the only prohibitive component here are the calls to $\mathcal{F}_{\mathsf{DORAM}}$. We give the per instruction, per party cost (in terms of field elements) for $\mathcal{F}_{\mathsf{DORAM}}(\mathsf{Access})$ and sub-protocols required for instantiating it. We give per execution cost for instantiating $\mathcal{F}_{\mathsf{DORAM}}(\mathsf{Init})$ and $\mathcal{F}_{\mathsf{DORAM}}(\mathsf{GetM})$. The costs are summarized in Table 2.

- $\mathcal{F}_{\mathsf{Zero}}$: Can be implemented naively by each party sending a vector of size m and locally computing shares of 0, which requires $\mathcal{O}(m)$ communication and computation.
- $\mathcal{F}_{\mathsf{Select}}$: Implemented by computing $\mathsf{flag} \cdot x + (1 - \mathsf{flag}) \cdot y$ in MPC with two multiplications, and thus $\mathcal{O}(1)$ communication and computation.
- Π_{DOPRF}: For both versions, this requires computing and sending vectors (of elements from $\mathbb{F}_p$) of length ℓ, where ℓ is at most $2\log_2(m) + \mathsf{stat}$, and $m = n + \sqrt{n}$. Thus, communication and computation is $\mathcal{O}(\log_2(m))$ field elements.
- $\Pi_{\mathsf{3\text{-}OT}}(\mathsf{Init})$: Each party communicates 2 PRF, 1 PRP key. In addition, the party evaluates the mask vector $\boldsymbol{r}$. This requires communication $\mathcal{O}(1)$ and computation $\mathcal{O}(m)$.
- $\Pi_{\mathsf{3\text{-}OT}}(\mathsf{Access})$: Per iteration, each party communicates an index $\mathsf{indx} \in [m]$ and a PRF evaluation a, and locally evaluates one PRF evaluation. This requires communication and computation $\mathcal{O}(1)$.

Protocol Π^{m}_{DOPRF}

Parameters: Output length ℓ, a Prg that expands a seed of length l to l' field elements.
KeyGen: R_1 samples $\boldsymbol{k}_1 \leftarrow \mathbb{F}_p^{\ell}$, and sends it to K. R_2 samples $\boldsymbol{k}_2 \leftarrow \mathbb{F}_p^{\ell}$, and sends it to K. K sets $\boldsymbol{k} := \boldsymbol{k}_1 + \boldsymbol{k}_2$.
Init: Parties sample pairwise PRG seeds: K samples $k_{k,1} \leftarrow \{0,1\}^l$, R_1 samples $k_{1,2} \leftarrow \{0,1\}^l$, and R_2 samples $k_{2,k} \leftarrow \{0,1\}^l$, and sends it to R_1, R_2, and K, respectively. Let α be a fixed quadratic non-residue modulo p known to R_1 and R_2.
Eval: Each party has input $\langle x \rangle$, and pairwise PRG seeds. K in addition has key k.

1. K and R_1 compute $m, \boldsymbol{a}, \boldsymbol{c}_1, \boldsymbol{e} \leftarrow \mathsf{PRG}(k_{k,1})$, where $m \in \mathbb{F}_p, \boldsymbol{a}, \boldsymbol{a}_1, \boldsymbol{e} \in \mathbb{F}_p^{\ell}$.
2. $\mathsf{R}_1, \mathsf{R}_2$ compute $\boldsymbol{s} \leftarrow \mathsf{PRG}(k_{1,2})$, where $\boldsymbol{s} \in \mathbb{F}_p^{\ell}$.
3. R_1 samples $\boldsymbol{r}_1 \leftarrow \{0,1\}^{\ell}$, and sends it to R_2. Similarly, R_2 samples $\boldsymbol{r}_2 \leftarrow \{0,1\}^{\ell}$, and sends it to R_1. Both $\mathsf{R}_1, \mathsf{R}_2$ set $\boldsymbol{r} := \boldsymbol{r}_1 \oplus \boldsymbol{r}_2$.
4. $\mathsf{R}_1, \mathsf{R}_2$ define $\boldsymbol{t}$ as: for $j \in [1, \ell]$, set $\boldsymbol{t}[j] := \boldsymbol{s}^2[j]$ if $\boldsymbol{r}[j] = 0$. Else, set $\boldsymbol{t}[j] := \boldsymbol{s}^2[j] \cdot \alpha$.
5. R_1 defines $\boldsymbol{b} := \boldsymbol{t} - \boldsymbol{e}$, and $\boldsymbol{c}_3 := \boldsymbol{a} \circ \boldsymbol{b} - \boldsymbol{c}_1$. Send $\boldsymbol{c}_3$ to R_2.
6. K computes $y_1 := \langle x \rangle - m$, R_1 computes $y_2 := \langle x \rangle + m$ and R_2 sets $y_3 := \langle x \rangle$. R_1 sends y_2 to R_2.
7. K computes and sends $\boldsymbol{d} := \boldsymbol{k} + y_1 - \boldsymbol{a}$ to R_2.
8. R_2 computes $\boldsymbol{w} := \boldsymbol{t} \cdot (y_2 + y_3)$, and $\boldsymbol{z}_3 := \boldsymbol{d} \circ \boldsymbol{t} + \boldsymbol{c}_3 + \boldsymbol{w}$. It sends $\boldsymbol{z}_3$ to K.
9. K computes $\boldsymbol{z}_1 := \boldsymbol{e} \circ (\boldsymbol{k} + y_1)\boldsymbol{c}_1 - \boldsymbol{d} \circ \boldsymbol{e}$, and $\boldsymbol{z} := \boldsymbol{z}_1 + \boldsymbol{z}_3$.
10. K computes, for $j \in [1, \ell]$, $\boldsymbol{o}[j] := \frac{1}{2}\big(\big(\frac{\boldsymbol{z}[j]}{p}\big) + 1\big) \bmod p$.
11. K outputs $\boldsymbol{o}$, $\mathsf{R}_1, \mathsf{R}_2$ output $\boldsymbol{r}$.

Fig. 9: Protocol for a secure, distributed and oblivious evaluation of PRF.

– $\Pi_{\mathsf{stash}}(\mathsf{Read})$: The cost here is dominated by converting additive to replicative shares which requires $\mathcal{O}(\sqrt{n})$ communication and computation.
– $\Pi_{\mathsf{stash}}(\mathsf{Write})$: The communication is dominated by calling $\mathcal{F}_{\mathsf{Zero}}$, which results in $\mathcal{O}(\sqrt{n})$ communication and computation cost, per iteration.
– $\Pi_{\mathsf{r\text{-}M}}(\mathsf{Init})$: Mainly involves sending a masked, and sorted M, and one run of $\Pi_{\mathsf{3\text{-}OT}}(\mathsf{Init})$. Total communication and computation cost is $\mathcal{O}(m\log(m))$, and per access, it is $\mathcal{O}(\sqrt{n}\log(m))$.
– $\Pi_{\mathsf{r\text{-}M}}(\mathsf{Access})$: Per access, there is one $\mathcal{F}_{\mathsf{DOPRF}}$ evaluation, and one run of $\Pi_{\mathsf{3\text{-}OT}}(\mathsf{Access})$. Thus the communication and computation is $\mathcal{O}(\log_2(m))$.
– $\Pi_{\mathsf{w\text{-}M}}$: The communication and computation cost is dominated by generating and evaluating multi-point DPF keys, which results in per iteration cost of $\mathcal{O}(\sqrt{n})$.
– $\Pi_{\mathsf{DORAM}}(\mathsf{Init})$: This consists of just one call to $\Pi_{\mathsf{r\text{-}M}}(\mathsf{Init})$. Thus, per execution, cost is $\mathcal{O}(m\log(m))$ for both communication and computation. This amounts to $\mathcal{O}(\log(m)/t)$ cost per instruction.
– $\Pi_{\mathsf{DORAM}}(\mathsf{Access})$: Per access, this requires one read and write access to the stash $\mathcal{F}_{\mathsf{Stash}}$, two calls to $\mathcal{F}_{\mathsf{Select}}$, and one $\Pi_{\mathsf{r\text{-}M}}(\mathsf{Access})$. After each $\sqrt{n}$ accesses, there is one call to $\Pi_{\mathsf{w\text{-}M}}$ and $\Pi_{\mathsf{r\text{-}M}}(\mathsf{Init})$. Therefore, per access computation and communication cost is $\mathcal{O}(\sqrt{n}\log(m))$.
– $\Pi_{\mathsf{DORAM}}(\mathsf{GetM})$: Per execution, just one call to $\Pi_{\mathsf{w\text{-}M}}$, which requires $\mathcal{O}(n)$ communication and computation per call. This is called only once after all $t \times \sqrt{n}$ instructions have been executed. Thus, per access, this costs only $\mathcal{O}(\sqrt{n}/t)$.

Cost for one Π_{f} execution. For $t \cdot \sqrt{n}$ instructions, Π_{f} executes $\Pi_{\mathsf{DORAM}}(\mathsf{Init})$ once, $\Pi_{\mathsf{DORAM}}(\mathsf{Access})$ $t \cdot \sqrt{n}$ times, and $\Pi_{\mathsf{DORAM}}(\mathsf{GetM})$ once. This amounts to $\mathcal{O}(\sqrt{n}\log(m))$ computation and communication cost per instruction. Note that all sub-protocols require only constant rounds of communication per iteration.

5 Implementation

To benchmark the performance of our protocols, we implemented them using the Rust programming language. Our implementation is available as open source software at https://github.com/AarhusCrypto/Ramen.

5.1 Setup

Instantiations of Primitives We implement the multi-point distributed point function with the protocol of [SGRR19], and the single-point distributed point function with Half-Tree [GYW+23] using a fixed-

Table 2: Amortized asymptotic costs per party for executing one instruction. Here, n is the number of elements in memory and $m := n + \sqrt{n}$. Each sub-protocol requires $\mathcal{O}(1)$ rounds of communication.

Primitive	Computation	Communication
Π_{DOPRF}	$\mathcal{O}(\log(m))$	$\mathcal{O}(\log(m))$
$\Pi_{\mathsf{3\text{-}OT}}(\mathsf{Init})$	$\mathcal{O}(m)$	$\mathcal{O}(1)$
$\Pi_{\mathsf{3\text{-}OT}}(\mathsf{Access})$	$\mathcal{O}(1)$	$\mathcal{O}(1)$
$\Pi_{\mathsf{stash}}(\mathsf{Read})$	$\mathcal{O}(\sqrt{n})$	$\mathcal{O}(\sqrt{n})$
$\Pi_{\mathsf{stash}}(\mathsf{Write})$	$\mathcal{O}(\sqrt{n})$	$\mathcal{O}(\sqrt{n}\log(p))$
$\Pi_{\mathsf{r\text{-}M}}(\mathsf{Init})$	$\mathcal{O}(m\log(m))$	$\mathcal{O}(m\log(m))$
$\Pi_{\mathsf{r\text{-}M}}(\mathsf{Access})$	$\mathcal{O}(\log(m))$	$\mathcal{O}(\log(m))$
$\Pi_{\mathsf{w\text{-}M}}$	$\mathcal{O}(\sqrt{n})$	$\mathcal{O}(\sqrt{n})$
$\Pi_{\mathsf{DORAM}}(\mathsf{Init})$	$\mathcal{O}(\log(m)/t)$	$\mathcal{O}(\log(m)/t)$
$\Pi_{\mathsf{DORAM}}(\mathsf{Access})$	$\mathcal{O}(\sqrt{n}\log(m))$	$\mathcal{O}(\sqrt{n}\log(m))$
$\Pi_{\mathsf{DORAM}}(\mathsf{GetM})$	$\mathcal{O}(\log(m)/t)$	$\mathcal{O}(\log(m)/t)$

key-AES-based hash function [GKWY20]. For the Legendre PRF, we use the prime field $\mathbb{F}_p$ with $p = 340282366920938462946865773367900766209$. Moreover, we use AES-CTR as a PRG and Blake3 to instantiate a PRF into $\mathbb{F}_p$, where we obtain $\mathbb{F}_p$ via rejection sampling.

Experimental Setup We conducted our experiments on a set of three servers each equipped with an Intel Core i9-9900 CPU having 16 logical cores (i.e., including hyperthreading) and 128 GB memory, connected with 10 Gigabit Ethernet. On average, we measured a bandwidth of 9.4 Gbit/s and a round-trip time (RTT) of 1 ms. Additionally, we also configure Traffic Control in the Linux kernel with the `tc (8)` utility to simulate a WAN setting with 100 Mbit/s bandwidth and a 30 ms RTT, as well as various settings where we either limit the bandwidth or enforce additional latency.

For each experiment with memory size n, we measured the time that the protocol needs for one epoch consisting of $\sqrt{n}$ accesses (until the stash is full) followed by the then necessary refresh (where the stash is written back into the memory). This allows us to give the amortized run-times per access operation, since the accesses early in the epoch are slightly faster because of the lower number of elements stored in the stash. We separate the costs into the input-independent preprocessing phase and the online phase of the protocol.

To compare our work with Duoram [VHG23], we also run experiments with their implementation[6] on the same hardware. For each n, we first run the preprocessing phase to measure the time required to preprocess the necessary DPFs per write, followed by the online phase of 128 writes. We chose to measure the time required for writes, since it similarly to our access – which hides whether it is a read or a write – also consists of a read followed by an update to the memory.

5.2 Benchmarking Results

Memory Size In Table 3, we give the runtimes for memories with sizes from $n = 2^8$ to $n = 2^{26}$ in the LAN and WAN settings, where we run both the preprocessing and the online phase with 16 threads.

In the LAN setting, the runtimes for the online phase increase slowly with the memory size n: We only need 37.56 ms per access in a memory with $n = 2^{26}$ entries. The runtimes are much higher in the WAN setting, starting at about 330 ms, but they still only increase very slowly with n: There is only about 50 ms difference between $n = 2^8$ and $n = 2^{26}$ for the online phase in the WAN setting. Hence, the latency is the dominating factor in this setting, as our protocol needs several (although constant) rounds for each access.

For the preprocessing phase, we clearly see that the runtimes increase linearly in $\sqrt{n}$. Moreover, in the WAN setting, the preprocessing runtimes do not increase much compared to the LAN setting, since the

[6] https://git-crysp.uwaterloo.ca/avadapal/duoram/src/2a88d1a2976d727ceec6915c02d3134149185d3a

Table 3: Amortized runtimes in ms and communication costs in KiB per access for memory sizes $n = 2^8$ to $n = 2^{26}$ in the LAN and WAN setting with 16 threads for preprocessing and online phase.

$\log_2 n$	Time per Access in ms						Communication in KiB		
	LAN			WAN					
	Online	Prep.	Total	Online	Prep.	Total	Online	Prep.	Total
8	2.83	0.43	3.26	328.92	10.09	339.01	12.02	4.90	16.92
9	5.70	0.77	6.47	336.38	7.65	344.03	12.75	5.24	17.99
10	3.21	0.82	4.02	332.12	5.67	337.79	13.61	5.46	19.07
11	5.80	1.13	6.93	331.39	4.43	335.82	14.67	5.91	20.58
12	4.00	1.10	5.10	334.00	3.78	337.78	16.37	7.64	24.01
13	2.28	1.32	3.60	336.34	2.79	339.13	18.69	9.37	28.07
14	5.70	1.76	7.45	334.92	2.93	337.85	21.46	11.38	32.84
15	4.30	1.98	6.28	335.45	3.80	339.26	25.69	14.20	39.89
16	4.48	2.59	7.07	335.87	5.03	340.90	31.15	18.03	49.18
17	10.59	3.81	14.40	336.39	5.86	342.25	39.12	23.47	62.59
18	11.11	5.29	16.40	337.16	7.39	344.55	49.85	31.03	80.88
19	8.75	7.67	16.42	338.23	9.67	347.90	65.29	41.74	107.03
20	8.86	12.15	21.01	339.68	13.15	352.83	86.56	56.77	143.34
21	15.74	17.73	33.47	341.89	19.73	361.62	116.92	78.05	194.97
22	17.86	25.67	43.54	345.39	28.34	373.73	159.29	108.02	267.31
23	21.21	37.16	58.37	350.43	41.10	391.52	219.48	150.43	369.91
24	25.65	53.82	79.48	357.13	59.62	416.75	304.03	210.27	514.30
25	31.28	77.86	109.14	366.67	85.92	452.59	423.91	294.93	718.83
26	37.56	112.53	150.09	378.66	124.23	502.89	592.77	414.52	1 007.29

preprocessing protocol is constant round w.r.t. the number of accesses, and the 100 Mbit/s bandwidth is not a bottleneck.

Comparison with Three-Party Duoram [VHG23] We compare the performance of the two protocols for memory sizes from $n = 2^8$ to $n = 2^{26}$ in the LAN and WAN settings. The resulting runtimes are visualized in Fig. 10. The corresponding data is provided in Table 4 (Appendix D).

Since the Duoram implementation does not support multithreading in the online phase, we used for both protocols 16 threads in the preprocessing phase, and then a single thread in the online phase. Note that in our work every memory entry is an element of a 128 bit prime field, whereas Duoram works by default on 64 bit integers. Hence, it is more costly for us to send elements between the parties, and the arithmetic is more costly as well. On the other hand, we can store about twice the amount of data per element.

The results show that, in the LAN setting, Duoram performs better in the online phase for memory sizes $n \leq 2^{20}$, whereas our protocol is faster for larger memory sizes $n \geq 2^{21}$. At this point the asymptotic difference in computation – our $\mathcal{O}(\sqrt{n})$ vs. Duoram's $\mathcal{O}(n)$ – cost kicks in: While our runtimes increase relatively slowly, those of Duoram deteriorate rapidly, so that our protocol is 14× faster for $n = 2^{26}$. In the WAN setting, this effect sets in a bit later, between $n = 2^{24}$ and $n = 2^{25}$, because of the higher round complexity of our protocol, and our protocol is more than 2× faster for $n = 2^{26}$.

The preprocessing phase of our protocol is even for large memory sizes very light, and takes always less time than the preprocessing of Duoram. With respect to the overall runtime, our protocol is faster for memory sizes $n \geq 2^{18}$ in the LAN setting and $n \geq 2^{19}$ in the WAN setting. For $n = 2^{26}$, we have an improvement of 25× in the LAN setting, and 8× in the WAN setting compared to Duoram.

Summarizing, we can say that our protocol is better suited for large memories, while Duoram performs well on small memories as well as in high-latency network settings.

Network Conditions To see how our protocol performs under different network conditions, we give benchmark results for accesses of an $n = 2^{22}$ element memory with 16 threads in Table 5a and 5b (Appendix D), where we limit the available bandwidth and impose an artificial latency, respectively. The

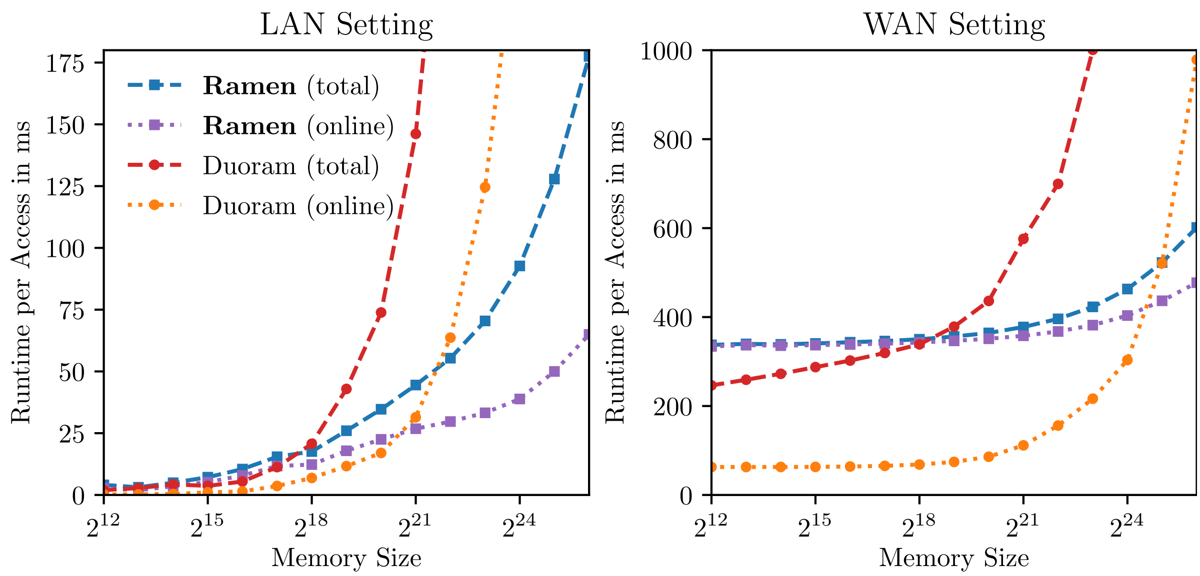

Fig. 10: Comparison of online and total runtimes of *Ramen* (this work) and the three-party Duoram protocol [VHG23] for different memory sizes in the LAN and WAN settings using 16 threads for the preprocessing phase and a single thread for the online phase.

bandwidth has only very little influence on the runtime – more than 50 Mbit/s does not increase the performance very much. Latency, on the other hand, has a large impact on the runtime in the online phase, which increase linearly. The preprocessing phase, however, is unaffected since its round complexity is independent of the number of accesses.

Multithreading To see how much of the computation in our protocol is parallelizable we ran experiments with 1 to 16 threads in the LAN setting with fixed memory size $n = 2^{22}$. In Table 6 (Appendix D) we give the runtimes as well as the achieved speedup and efficiency of the parallelization.[7]

In the online phase, the achieved speedup is limited to $< 2\times$ for any number of threads. This is likely due to the ratio of computation to the rounds of communication in the access protocol. It is to be expected that the efficiency of parallelization drops significantly in the WAN setting, where the parties already spend most of their time waiting for the network.

The preprocessing phase is much better parallelizable since it includes most of the heavy work such as the evaluations of the Legendre PRF on each index of the memory. Here we achieve a speedup of up to 8. Moreover, we see that for up to 8 threads, the efficiency stays above 0.8, but it drops to 0.5 when increasing adding more threads. This effect is likely due to hyperthreading, i.e., executing two logical threads on the same physical CPU core while sharing its resources.

Acknowledgements

This research was supported by the Carlsberg Foundation under the Semper Ardens Research Project CF18-112 (BCM), the European Research Council (ERC) under the European Unions's Horizon 2020 research and innovation programme under grant agreement No 803096 (SPEC), and the Digital Research Centre Denmark (DIREC). We thank Adithya Vadapalli, Ryan Henry, and Ian Goldberg for clarifications regarding Duoram [VHG23], and the COSIC group at KU Leuven for allowing us to use their servers for our experiments.

[7] If t_i is the runtime with i threads, then the speedup for j threads is defined as $s_j := t_1/t_j$ and the efficiency is $e_j = j/s_j$ measures how close it is to the ideal linear speedup.

References

BCGI18. Elette Boyle, Geoffroy Couteau, Niv Gilboa, and Yuval Ishai. Compressing vector OLE. In David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang, editors, *ACM CCS 2018: 25th Conference on Computer and Communications Security*, pages 896–912, Toronto, ON, Canada, October 15–19, 2018. ACM Press.

Bea95. Donald Beaver. Precomputing oblivious transfer. In Don Coppersmith, editor, *Advances in Cryptology – CRYPTO'95*, volume 963 of *Lecture Notes in Computer Science*, pages 97–109, Santa Barbara, CA, USA, August 27–31, 1995. Springer, Heidelberg, Germany.

BGW88. Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation (extended abstract). In *20th Annual ACM Symposium on Theory of Computing*, pages 1–10, Chicago, IL, USA, May 2–4, 1988. ACM Press.

BKKO20. Paul Bunn, Jonathan Katz, Eyal Kushilevitz, and Rafail Ostrovsky. Efficient 3-party distributed ORAM. In Clemente Galdi and Vladimir Kolesnikov, editors, *SCN 20: 12th International Conference on Security in Communication Networks*, volume 12238 of *Lecture Notes in Computer Science*, pages 215–232, Amalfi, Italy, September 14–16, 2020. Springer, Heidelberg, Germany.

BPRS23. Lennart Braun, Mahak Pancholi, Rahul Rachuri, and Mark Simkin. Ramen: Souper fast three-party computation for ram programs. In *ACM CCS 2023: 30th Conference on Computer and Communications Security*. ACM Press, November 2023.

Can01. Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *42nd Annual Symposium on Foundations of Computer Science*, pages 136–145, Las Vegas, NV, USA, October 14–17, 2001. IEEE Computer Society Press.

CCD88. David Chaum, Claude Crépeau, and Ivan Damgård. Multiparty unconditionally secure protocols (extended abstract). In *20th Annual ACM Symposium on Theory of Computing*, pages 11–19, Chicago, IL, USA, May 2–4, 1988. ACM Press.

Dam90. Ivan Damgård. On the randomness of Legendre and Jacobi sequences. In Shafi Goldwasser, editor, *Advances in Cryptology – CRYPTO'88*, volume 403 of *Lecture Notes in Computer Science*, pages 163–172, Santa Barbara, CA, USA, August 21–25, 1990. Springer, Heidelberg, Germany.

DMN11. Ivan Damgård, Sigurd Meldgaard, and Jesper Buus Nielsen. Perfectly secure oblivious RAM without random oracles. In Yuval Ishai, editor, *TCC 2011: 8th Theory of Cryptography Conference*, volume 6597 of *Lecture Notes in Computer Science*, pages 144–163, Providence, RI, USA, March 28–30, 2011. Springer, Heidelberg, Germany.

Ds17. Jack Doerner and abhi shelat. Scaling ORAM for secure computation. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *ACM CCS 2017: 24th Conference on Computer and Communications Security*, pages 523–535, Dallas, TX, USA, October 31 – November 2, 2017. ACM Press.

FJKW15. Sky Faber, Stanislaw Jarecki, Sotirios Kentros, and Boyang Wei. Three-party ORAM for secure computation. In Tetsu Iwata and Jung Hee Cheon, editors, *Advances in Cryptology – ASIACRYPT 2015, Part I*, volume 9452 of *Lecture Notes in Computer Science*, pages 360–385, Auckland, New Zealand, November 30 – December 3, 2015. Springer, Heidelberg, Germany.

GGMP16. Sanjam Garg, Divya Gupta, Peihan Miao, and Omkant Pandey. Secure multiparty RAM computation in constant rounds. In Martin Hirt and Adam D. Smith, editors, *TCC 2016-B: 14th Theory of Cryptography Conference, Part I*, volume 9985 of *Lecture Notes in Computer Science*, pages 491–520, Beijing, China, October 31 – November 3, 2016. Springer, Heidelberg, Germany.

GHL+14. Craig Gentry, Shai Halevi, Steve Lu, Rafail Ostrovsky, Mariana Raykova, and Daniel Wichs. Garbled RAM revisited. In Phong Q. Nguyen and Elisabeth Oswald, editors, *Advances in Cryptology – EUROCRYPT 2014*, volume 8441 of *Lecture Notes in Computer Science*, pages 405–422, Copenhagen, Denmark, May 11–15, 2014. Springer, Heidelberg, Germany.

GI14. Niv Gilboa and Yuval Ishai. Distributed point functions and their applications. In Phong Q. Nguyen and Elisabeth Oswald, editors, *Advances in Cryptology – EUROCRYPT 2014*, volume 8441 of *Lecture Notes in Computer Science*, pages 640–658, Copenhagen, Denmark, May 11–15, 2014. Springer, Heidelberg, Germany.

GKK+12. S. Dov Gordon, Jonathan Katz, Vladimir Kolesnikov, Fernando Krell, Tal Malkin, Mariana Raykova, and Yevgeniy Vahlis. Secure two-party computation in sublinear (amortized) time. In Ting Yu, George Danezis, and Virgil D. Gligor, editors, *ACM CCS 2012: 19th Conference on Computer and Communications Security*, pages 513–524, Raleigh, NC, USA, October 16–18, 2012. ACM Press.

GKWY20. Chun Guo, Jonathan Katz, Xiao Wang, and Yu Yu. Efficient and secure multiparty computation from fixed-key block ciphers. In *2020 IEEE Symposium on Security and Privacy*, pages 825–841, San Francisco, CA, USA, May 18–21, 2020. IEEE Computer Society Press.

GLO15. Sanjam Garg, Steve Lu, and Rafail Ostrovsky. Black-box garbled RAM. In Venkatesan Guruswami, editor, *56th Annual Symposium on Foundations of Computer Science*, pages 210–229, Berkeley, CA, USA, October 17–20, 2015. IEEE Computer Society Press.

GLOS15. Sanjam Garg, Steve Lu, Rafail Ostrovsky, and Alessandra Scafuro. Garbled RAM from one-way functions. In Rocco A. Servedio and Ronitt Rubinfeld, editors, *47th Annual ACM Symposium on Theory of Computing*, pages 449–458, Portland, OR, USA, June 14–17, 2015. ACM Press.

GMW87. Oded Goldreich, Silvio Micali, and Avi Wigderson. How to play any mental game or A completeness theorem for protocols with honest majority. In Alfred Aho, editor, *19th Annual ACM Symposium on Theory of Computing*, pages 218–229, New York City, NY, USA, May 25–27, 1987. ACM Press.

GO96. Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious rams. *J. ACM*, 1996.

GRR+16. Lorenzo Grassi, Christian Rechberger, Dragos Rotaru, Peter Scholl, and Nigel P. Smart. MPC-friendly symmetric key primitives. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *ACM CCS 2016: 23rd Conference on Computer and Communications Security*, pages 430–443, Vienna, Austria, October 24–28, 2016. ACM Press.

GYW+23. Xiaojie Guo, Kang Yang, Xiao Wang, Wenhao Zhang, Xiang Xie, Jiang Zhang, and Zheli Liu. Half-tree: Halving the cost of tree expansion in COT and DPF. In Carmit Hazay and Martijn Stam, editors, *Advances in Cryptology – EUROCRYPT 2023, Part I*, volume 14004 of *Lecture Notes in Computer Science*, pages 330–362, Lyon, France, April 23–27, 2023. Springer, Heidelberg, Germany.

HKO22. David Heath, Vladimir Kolesnikov, and Rafail Ostrovsky. EpiGRAM: Practical garbled RAM. In Orr Dunkelman and Stefan Dziembowski, editors, *Advances in Cryptology – EUROCRYPT 2022, Part I*, volume 13275 of *Lecture Notes in Computer Science*, pages 3–33, Trondheim, Norway, May 30 – June 3, 2022. Springer, Heidelberg, Germany.

HV21. Ariel Hamlin and Mayank Varia. Two-server distributed ORAM with sublinear computation and constant rounds. In Juan Garay, editor, *PKC 2021: 24th International Conference on Theory and Practice of Public Key Cryptography, Part II*, volume 12711 of *Lecture Notes in Computer Science*, pages 499–527, Virtual Event, May 10–13, 2021. Springer, Heidelberg, Germany.

JW18. Stanislaw Jarecki and Boyang Wei. 3PC ORAM with low latency, low bandwidth, and fast batch retrieval. In Bart Preneel and Frederik Vercauteren, editors, *ACNS 18: 16th International Conference on Applied Cryptography and Network Security*, volume 10892 of *Lecture Notes in Computer Science*, pages 360–378, Leuven, Belgium, July 2–4, 2018. Springer, Heidelberg, Germany.

KY18. Marcel Keller and Avishay Yanai. Efficient maliciously secure multiparty computation for RAM. In Jesper Buus Nielsen and Vincent Rijmen, editors, *Advances in Cryptology – EUROCRYPT 2018, Part III*, volume 10822 of *Lecture Notes in Computer Science*, pages 91–124, Tel Aviv, Israel, April 29 – May 3, 2018. Springer, Heidelberg, Germany.

LO13. Steve Lu and Rafail Ostrovsky. How to garble RAM programs. In Thomas Johansson and Phong Q. Nguyen, editors, *Advances in Cryptology – EUROCRYPT 2013*, volume 7881 of *Lecture Notes in Computer Science*, pages 719–734, Athens, Greece, May 26–30, 2013. Springer, Heidelberg, Germany.

OS97. Rafail Ostrovsky and Victor Shoup. Private information storage (extended abstract). In *29th Annual ACM Symposium on Theory of Computing*, pages 294–303, El Paso, TX, USA, May 4–6, 1997. ACM Press.

SGRR19. Phillipp Schoppmann, Adrià Gascón, Leonie Reichert, and Mariana Raykova. Distributed vector-OLE: Improved constructions and implementation. In Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz, editors, *ACM CCS 2019: 26th Conference on Computer and Communications Security*, pages 1055–1072, London, UK, November 11–15, 2019. ACM Press.

VHG23. Adithya Vadapalli, Ryan Henry, and Ian Goldberg. Duoram: A bandwidth-efficient distributed oram for 2- and 3-party computation. In *USENIX Security 2023: 32nd USENIX Security Symposium*, Anaheim, CA, USA, August 2023. USENIX Association.

Yao82. Andrew Chi-Chih Yao. Protocols for secure computations (extended abstract). In *23rd Annual Symposium on Foundations of Computer Science*, pages 160–164, Chicago, Illinois, November 3–5, 1982. IEEE Computer Society Press.

Yao86. Andrew Chi-Chih Yao. How to generate and exchange secrets (extended abstract). In *27th Annual Symposium on Foundations of Computer Science*, pages 162–167, Toronto, Ontario, Canada, October 27–29, 1986. IEEE Computer Society Press.

ZWR+16. Samee Zahur, Xiao Shaun Wang, Mariana Raykova, Adria Gascón, Jack Doerner, David Evans, and Jonathan Katz. Revisiting square-root ORAM: Efficient random access in multi-party computation. In *2016 IEEE Symposium on Security and Privacy*, pages 218–234, San Jose, CA, USA, May 22–26, 2016. IEEE Computer Society Press.

A Protocols and Security Proofs

A.1 Security Model

We prove the security of our protocols in the universal composability framework of [Can01]. The security requirements are captured by defining the real-world and the ideal-world. The real-world experiment is defined with respect to a protocol Π (run by parties P_1, P_2, P_3), an adversary $\mathcal{A}$ and an environment $\mathcal{Z}$. The environment $\mathcal{Z}$ can write inputs to all parties, read outputs of all parties, and additionally interact with $\mathcal{A}$ throughout execution. After $\mathcal{A}$ is activated, it can corrupt one of the parties in the system, and from then on gets read-only access to the internal state of the corrupted party. It also gets to change any input for the corrupt party, which is modeled as follows: When $\mathcal{Z}$ wants to write input x for a corrupted party, it first interacts with $\mathcal{A}$ who might change it to x', after which $\mathcal{Z}$ writes x' on the input tape of the corrupt party. Let $\mathsf{Real}_{\Pi,\mathcal{A},\mathcal{Z}}$ denote the output of the distinguisher $\mathcal{Z}$ when interacting in the real-world experiment. The ideal-world experiment is defined with respect to an ideal functionality $\mathcal{F}$, ideal adversary $\mathcal{S}$, the environment $\mathcal{Z}$ and a set of dummy parties $\tilde{P}_1, \tilde{P}_2, \tilde{P}_3$. Similar to the previous case, $\mathcal{Z}$ writes to the input tapes, and read from the output tapes of all dummy parties. As in the real-world experiment, here too $\mathcal{Z}$ interacts with the adversary $\mathcal{S}$ throughout the execution. When $\mathcal{S}$ is activated for the first time, it activates an instance of $\mathcal{A}$ internally, and relays communication between $\mathcal{A}$ and $\mathcal{Z}$. This is useful when $\mathcal{Z}$ wants to interact with $\mathcal{A}$ to decide corrupt party's inputs; it does so via $\mathcal{S}$. This means that $\mathcal{S}$ in the passive case is aware of the inputs of the corrupted party even if they might be modified by $\mathcal{A}$. Finally, when each dummy party receives its input, it sends it to $\mathcal{F}$ to receive outputs. The corrupt party's output is received by $\mathcal{S}$ who writes it on its output tape to be read by $\mathcal{Z}$. Let $\mathsf{Ideal}_{\mathcal{F},\mathcal{S},\mathcal{Z}}$ denote the output of the distinguisher $\mathcal{Z}$ when interacting in the ideal-world experiment.

Definition 2. *Let $\mathcal{F}$ be a three-party functionality and Π be a three-party protocol. We say that Π securely realizes functionality $\mathcal{F}$ in the presence of one passive corruption, if for all PPT adversaries $\mathcal{A}$, there exists a PPT algorithm $\mathcal{S}$, such that, for all PPT $\mathcal{Z}$,*

$$\{\mathsf{Ideal}_{\mathcal{F},\mathcal{S},\mathcal{Z}}(\boldsymbol{x}, \lambda, z)\}_{\boldsymbol{x},\lambda,z} \stackrel{c}{\equiv} \{\mathsf{Real}_{\Pi,\mathcal{A},\mathcal{Z}}(\boldsymbol{x}, \lambda, z)\}_{\boldsymbol{x},\lambda,z}$$

where Ideal *and* Real *are experiments as described earlier, $\boldsymbol{x} = (x_0, x_1, x_2)$, and $x_i \in \{0,1\}^*$ is the input of party P_i, $z \in \{0,1\}^*$ is the auxiliary input for $\mathcal{A}$, and λ is the security parameter.*

Each proof of security will follow the same pattern. First the simulator $\mathcal{S}$ starts by invoking an instance of $\mathcal{A}$ and runs a simulated interaction of $\mathcal{A}$ with $\mathcal{Z}$ (by simply relaying messages between them). $\mathcal{S}$ receives the adversary's inputs from $\mathcal{Z}$, and it writes them on $\mathcal{A}$'s input tape. Similarly, every output value written by $\mathcal{A}$ is copied by $\mathcal{S}$ on its output tape to be read by $\mathcal{Z}$. The simulator $\mathcal{S}$ calls $\mathcal{F}$ on the adversary's input to receive the function's output. It then continues to simulate $\mathcal{A}$'s view using this output. Since this is a common strategy for all simulators, we skip the explicit invocation of $\mathcal{A}$ in the proofs. As is common practice, each new invocation of a functionality happens with a new session id. We, however, will not explicitly mention this in our protocol descriptions or proofs.

A.2 Security Proof of $\Pi_{\mathsf{3\text{-}OT}}$

Theorem 1. *Protocol $\Pi_{\mathsf{3\text{-}OT}}$ UC-securely implements functionality $\mathcal{F}_{\mathsf{3\text{-}OT}}$ in presence of one passive corruption.*

Proof. The correctness of the protocol is immediate. Next, we will argue that our protocol is secure. Init for $\Pi_{\mathsf{3\text{-}OT}}$ is run once, while Access can run up to ℓ times. We consider all possible corruption scenario and construct a simulator $\mathcal{S}$ for each of the case.

S *corrupt:* From $\mathcal{Z}$, $\mathcal{S}$ receives command Init. It receives keys $k_{\mathsf{C}}, k_{\mathsf{R}}, k_{\mathsf{PRP}}$ from S on behalf of C, R, computes vector $\boldsymbol{r}$ just as S would in the protocol $\Pi_{\mathsf{3\text{-}OT}}$, and calls $\mathcal{F}_{\mathsf{3\text{-}OT}}$ on $\boldsymbol{r}$. It receives no output in return.

C *corrupt:* On receiving command (Access, indx) from $\mathcal{Z}$, call $\mathcal{F}_{\mathsf{3\text{-}OT}}$. $\mathcal{S}$ receives no output in return. Simulate the view as follows. It simulates the transcript in the Init phase by sending randomly sampled k_{PRP} and k_{C} to C. Note that, these keys are generated exactly as in the real world and hence are indistinguishable. Since C receives no output and no incoming messages in the Access phase, $\mathcal{S}$ simulates the transcript by accepting messages sent by C.

R *corrupt:* On receiving command (Init) from $\mathcal{Z}$, $\mathcal{S}$ calls $\mathcal{F}_{\text{3-OT}}$. TO simulate the view, it samples and returns a random key $k_{\mathsf{R}} \leftarrow \mathcal{K}_{\mathsf{PRF}}$. This is exactly as in the real world and hence is indistinguishable. For each iteration of Access, on receiving command (Access) from $\mathcal{Z}$, the simulator works as follows:

1. Since R has no input, $\mathcal{S}$ simply calls $\mathcal{F}_{\text{3-OT}}$ on input Access and receives x which is the output of the current iteration.
2. Pick a random index $\mathsf{indx}' \in [\ell]$ that has not been picked in any of the previous iterations.
3. Set $a := x - \mathsf{PRF}(k_{\mathsf{R}}, \mathsf{indx}')$.
4. Output (indx', a) as the message that R receives from C.

This is indistinguishable from the real world view. We describe hybrids for one Init operation followed by one iteration of Access, which can be replicated for all iterations.

- Hyb_0: The real world view obtained by executing steps in $\Pi_{\text{3-OT}}$.
- Hyb_1: Same as Hyb_0 except that $\mathcal{S}$ receives x from $\mathcal{F}_{\text{3-OT}}$ and sets a as $a := x - \mathsf{PRF}(k_{\mathsf{R}}, \pi(\mathsf{indx}))$. This is indistinguishable because of correctness of the protocol.
- Hyb_2: Same as Hyb_1 except that $\pi(\mathsf{indx})$ is replaced by a uniformly random index $\mathsf{indx}' \in [\ell]$ such that indx' was not picked in any past iteration. This is indistinguishable from the previous hybrid because of security of PRP scheme.

Observe that Hyb_2 is exactly how $\mathcal{S}$ behaves for a corrupt R. □

A.3 Security Proof of Π_{f}

Protocol Π_{f}

Parameters: A next-step function NS. Each party inputs shares $\langle x\rangle, \langle \mathsf{M}\rangle$, and gets output $\langle \mathsf{st}_{\mathsf{NS}}\rangle, \langle \mathsf{M}\rangle, \mathsf{rnd}$.

Run: 1. All parties initialize $\langle \mathsf{st}_{\mathsf{NS}}\rangle := (\mathsf{start}, \langle x\rangle)$, $\langle d\rangle := 0$ and $\mathsf{rnd} := 0$. Call $\mathcal{F}_{\mathsf{DORAM}}$ with $(\mathsf{Init}, \langle \mathsf{M}\rangle)$.

2. Do:
   (a) The parties call $\mathcal{F}_{\mathsf{MPC}}$ with inputs $(\langle \mathsf{st}_{\mathsf{NS}}\rangle, \langle d\rangle)$ to securely compute $(\langle \mathsf{st}_{\mathsf{NS}}\rangle, \langle \mathsf{I}\rangle) \leftarrow \mathsf{NS}(\langle \mathsf{st}_{\mathsf{NS}}\rangle, \langle d\rangle)$.
   (b) Parse $\langle \mathsf{st}_{\mathsf{NS}}\rangle$ as $(\langle z\rangle, \cdot)$.
   (c) Parties z as: For $i \in \{1,2,3\}$, P_i sends $\langle z\rangle$ to P_{i-1}, P_{i+1}. If $z = \mathsf{stop}$, then break. Else if $z = \mathsf{continue}$, continue.
   (d) Call $\mathcal{F}_{\mathsf{DORAM}}$ with input $(\mathsf{Access}, \langle \mathsf{I}\rangle)$ to receive output $\langle d\rangle$.
   (e) Update $\mathsf{rnd} := \mathsf{rnd} + 1$.
3. Call $\mathcal{F}_{\mathsf{DORAM}}$ with input (GetM) to receive output $\langle \mathsf{M}\rangle$.
4. Each party locally outputs $(\langle \mathsf{st}_{\mathsf{NS}}\rangle, \langle \mathsf{M}\rangle, \mathsf{rnd})$.

Fig. 11: Protocol for secure evaluation of a RAM program f.

Theorem 2. *Protocol Π_{f} UC-securely instantiates functionality $\mathcal{F}_{\mathsf{f}}$ in presence of one passive corruption in the $(\mathcal{F}_{\mathsf{DORAM}}, \mathcal{F}_{\mathsf{MPC}}, \mathcal{F}_{\mathsf{Check}})$-hybrid model.*

Proof. The correctness of Π_{f} is clear from inspection. We argue security against a corrupt P_1. The simulation strategy for other corruption cases is analogous.

- Hyb_0: Same as the real world execution except that, when $\mathcal{F}_{\mathsf{DORAM}}$ is called with input GetM, reply with $\langle \mathsf{M}_{\mathsf{out}}\rangle$. This is indistinguishable because correctness.
- Hyb_1 : When $\mathsf{crnd} = \mathsf{rnd}$, $\mathcal{S}_f$ returns random shares for $\langle \mathsf{I}\rangle$, and $\langle \mathsf{st}_{\Pi,\mathsf{out}}\rangle$, where $\langle \mathsf{st}_{\Pi,\mathsf{out}}\rangle$ is received from the functionality. This is indistinguishable because security of additive sharing, and correctness of Π_{f}.
- Hyb_2 : When $\mathsf{crnd} \neq \mathsf{rnd}$, $\mathcal{S}_f$ returns random shares for $\langle \mathsf{st}_{\Pi}\rangle$ and $\langle \mathsf{I}\rangle$ rather than the correctly computed shares. Moreover, it returns calls to $\mathcal{F}_{\mathsf{DORAM}}(\mathsf{Access})$ with a random $\langle d\rangle$. This is indistinguishable from the previous hybrid because of the security of the additive shares.

Notice that Hyb_2 is the view generated by $\mathcal{S}_{\mathsf{f}}$. □

Simulator $\mathcal{S}_{\mathsf{f}}$

1. On receiving inputs of the corrupt party $\langle \mathsf{M} \rangle, \langle x \rangle$, call $\mathcal{F}_{\mathsf{f}}$ to receive outputs outputs $(\langle \mathsf{st}_{\mathsf{NS,out}} \rangle, \langle \mathsf{M}_{\mathsf{out}} \rangle, \mathsf{rnd})$. Simulate the view as follows.
2. Receive call to $\mathcal{F}_{\mathsf{DORAM}}(\mathsf{Init})$ from P_1. Return nothing in response. Initialize current counter $\mathsf{crnd} := 0$.
3. For $\mathsf{crnd} < \mathsf{rnd}$, do:
   (a) Receive call to $\mathcal{F}_{\mathsf{MPC}}$ from P_1. Sample and return random $(\langle \mathsf{st}_{\mathsf{NS}} \rangle, \langle \mathsf{I} \rangle)$ (of appropriate length).
   (b) Receive $\langle z \rangle$ from P_1. Sample $\langle z \rangle_2, \langle z \rangle_3$ such that $z = \mathsf{continue}$. Send $\langle z \rangle_2, \langle z \rangle_3$ to P_1 on the behalf of P_2, P_3.
   (c) Receive call to $\mathcal{F}_{\mathsf{DORAM}}(\mathsf{Access})$ from P_1. Return $\langle d \rangle \leftarrow \mathbb{F}_p$.
   (d) Update $\mathsf{crnd} := \mathsf{crnd} + 1$.
4. If $\mathsf{crnd} = \mathsf{rnd}$, do:
   (a) Receive call to $\mathcal{F}_{\mathsf{MPC}}$ from P_1. Sample $\langle \mathsf{I} \rangle$ uniformly at random of appropriate length and return $(\langle \mathsf{st}_{\mathsf{NS,out}} \rangle, \langle \mathsf{I} \rangle)$.
   (b) Receive call to $\mathcal{F}_{\mathsf{Check}}$ from P_1. Return stop to P_1.
5. Receive call to $\mathcal{F}_{\mathsf{DORAM}}(\mathsf{GetM})$ from P_1 and return $\langle \mathsf{M}_{\mathsf{out}} \rangle$.

Fig. 12: Simulator for Theorem 2 in the case of corrupt P_1.

A.4 Security Proof of Π_{DORAM}

$\mathcal{F}_{\mathsf{DORAM}}$

Init: On receiving $(\mathsf{Init}, \langle \mathsf{M} \rangle)$ from all parties, reconstruct and store M locally.
Access: On receiving $(\mathsf{Access}, \langle \mathsf{I} \rangle)$ from all parties, reconstruct instruction $\mathsf{I} := (\mathsf{op}, \mathsf{adr}, \mathsf{val})$, and do:
1. If $\mathsf{I} = (\mathsf{read}, \mathsf{adr}, 0)$, set $d := \mathsf{M}[\mathsf{adr}].\mathsf{val}$.
2. Else if, $\mathsf{I} = (\mathsf{write}, \mathsf{adr}, \mathsf{val})$, set $d := \mathsf{M}[\mathsf{adr}].\mathsf{val}$ and $\mathsf{M}[\mathsf{adr}] := \mathsf{I}.\mathsf{val}$.
3. Output additive share $\langle d \rangle_i$ to party P_i.

Get Memory : On receiving (GetM) from all parties, output additive shares $\langle \mathsf{M} \rangle_i$ to party P_i.

Fig. 13: Functionality for distributed ORAM between three parties P_1, P_2, P_3 and executing instruction I on M obliviously.

Theorem 3. *Protocol Π_{DORAM} UC-securely instantiates functionality $\mathcal{F}_{\mathsf{DORAM}}$ in presence of one passive corruption in the $(\mathcal{F}_{\mathsf{r\text{-}M}}, \mathcal{F}_{\mathsf{Stash}}, \mathcal{F}_{\mathsf{Select}}, \mathcal{F}_{\mathsf{w\text{-}M}})$-hybrid model.*

Proof. Correctness and security for the simulator are argued as follows.

Correctness. Note that the very first execution of Refresh in Π_{DORAM} acts as the instantiation of Init command in $\mathcal{F}_{\mathsf{DORAM}}$. The correctness of Init phase is obvious from inspection. Correctness of read operations follow from the fact that the most up-to-date value is stored either in the stash or in M and the protocol reads from both and obliviously selects the actual value. For write operations, correctness follows from maintaining the updates into the stash. Since, the updates are pushed into M, correctness of GetM is also immediate.

Security. We argue the case where P_2 is corrupt. The simulation strategy for the other two cases is identical. The simulator for this case is in Fig.15. Observe that, the Init and GetM commands are invoked only once in both real and ideal world, while Access can queried multiple times. However, we will describe hybrids and argue indistinguishability considering Init, *a single* invocation of Access, followed by GetM. This can be easily generalized by simply repeating the simulation steps and the indistinguishability argument for multiple invocations of Access. The only constraint that will be maintained by the simulator when repeating its steps for multiple iterations is that it will pick the new random address to be added to the set $\mathsf{adr}_1^{\mathsf{read}}$ such that it is distinct from all previous addresses in the set. All other values are picked

Π_{DORAM}

Parameters: A PRF : $\mathcal{K} \times [1, m] \to \mathcal{Y}$, where $m = n + \sqrt{n}$.
Init/Refresh: Each party has input $\langle \mathsf{M} \rangle$ which is parsed as set of tuples $(\mathsf{indx}, \langle \mathsf{val} \rangle)$, where $\mathsf{indx} \in [1, n]$.

1. Set access counter $c := 1$, $\langle \mathsf{stash} \rangle := [(0,0,0)]^{\sqrt{n}}$.
2. Set $\langle \mathsf{st_{stash}} \rangle := \perp$, where $\langle \mathsf{st_{stash}} \rangle$ is parsed as $(\langle \mathsf{flag} \rangle, \langle \mathsf{loc} \rangle, \langle \mathsf{val_{st}} \rangle)$.
3. Define $\langle \mathsf{M} \rangle := \langle \mathsf{M} \rangle \,||\{(n+1, 0), \ldots, (m, 0)\}$.
4. Call $\mathcal{F}_{\mathsf{r\text{-}M}}$ with $(\mathsf{Init}, \langle \mathsf{M} \rangle)$.Each party P_i receives (k_{i-1}, k_{i+1}), and list L_i from $\mathcal{F}_{\mathsf{r\text{-}M}}$.
5. Each party P_i sets the list of addresses read $\mathsf{adr}_i^{\mathsf{read}} := \perp$.

Access: Each party inputs $\langle \mathsf{I} \rangle$, and gets output $\langle d \rangle$ at the end. In iteration c,

1. Call $\mathcal{F}_{\mathsf{Stash}}$ with input $(\mathsf{Read}, c, \langle \mathsf{stash} \rangle, \langle \mathsf{I} \rangle)$ and receive $\langle \mathsf{st_{stash}} \rangle$.
2. Set $\langle c + n \rangle$ as: P_1 sets its share as $c + n$ and P_2, P_3 set their shares to be 0. Parties call $\mathcal{F}_{\mathsf{Select}}$ with $(\langle \mathsf{flag} \rangle, \langle c + n \rangle, \langle \mathsf{I.adr} \rangle)$ to receive $\langle \mathsf{adr_M} \rangle$.
3. Each party P_i calls $\mathcal{F}_{\mathsf{r\text{-}M}}$ with input $(\mathsf{Access}, \langle \mathsf{adr_M} \rangle, \mathsf{adr}_i^{\mathsf{read}}, k_{i-1}, k_{i+1})$ and receives $(\langle \mathsf{val_M} \rangle, \mathsf{adr}_i^{\mathsf{read}})$.
4. Call $\mathcal{F}_{\mathsf{Stash}}$ with $(\mathsf{Write}, \langle \mathsf{val_M} \rangle, \langle \mathsf{adr_M} \rangle, c, \langle \mathsf{stash} \rangle, \langle \mathsf{st_{stash}} \rangle, \langle \mathsf{I} \rangle)$. All parties receive an updated $\langle \mathsf{stash} \rangle$.
5. Call $\mathcal{F}_{\mathsf{Select}}$ with $(\langle \mathsf{flag} \rangle, \langle \mathsf{val_{st}} \rangle, \langle \mathsf{val_M} \rangle)$ to receive $\langle d \rangle$.
6. If $c = \sqrt{n}$:
   (a) Each party P_i calls $\mathcal{F}_{\mathsf{w\text{-}M}}$ with $(\langle \mathsf{M}|_n \rangle, \langle \mathsf{stash} \rangle, L_i, \mathsf{adr}_i^{\mathsf{read}}, k_{i-1}, k_{i+1})$ to receive updated shares of $\langle \mathsf{M} \rangle$. Here, $\langle \mathsf{M}|_n \rangle$ represents the first n tuples in $\langle \mathsf{M} \rangle$.
   (b) Parties run $\mathsf{Refresh}(\langle \mathsf{M} \rangle)$.
7. Increment $c := c + 1$ and locally output $\langle d \rangle$.

Get Memory: Each party receives output the updated share $\langle \mathsf{M} \rangle$ at the end.

1. Each party P_i calls $\mathcal{F}_{\mathsf{w\text{-}M}}$ with $(\langle \mathsf{M}|_n \rangle, \langle \mathsf{stash} \rangle, L_i, \mathsf{adr}_i^{\mathsf{read}}, k_{i-1}, k_{i+1})$ to receive $\langle \mathsf{M} \rangle$. Here, $\langle \mathsf{M}|_n \rangle$ represents the first n tuples in $\langle \mathsf{M} \rangle$.
2. Parties locally output $\langle \mathsf{M} \rangle$.

Fig. 14: Protocol for distributed ORAM, realizing $\mathcal{F}_{\mathsf{DORAM}}$.

independently of the previous iterations. Additionally, in case of multiple iterations, the steps for GetM are executed before Access, and the i-th iteration Access is simulated is simulated before $i - 1$. The hybrids are as follows.

- Hyb_0 : Same as the real world execution except that the outputs are set differently by $\mathcal{S}_{\mathsf{DORAM}}$. It replies to the call to $\mathcal{F}_{\mathsf{w\text{-}M}}$ with $\langle \mathsf{M_{out}} \rangle$. Additionally, it replies to the call for $\mathcal{F}_{\mathsf{Select}}$ (Step 5) with d_{out}, where d_{out} is the current iteration output received from the functionality. This is indistinguishable because of correctness.
- Hyb_1: $\mathcal{S}_{\mathsf{DORAM}}$ answers call to $\mathcal{F}_{\mathsf{r\text{-}M}}$ with randomly sampled $\langle \mathsf{val_M} \rangle$ and a randomly sampled address from $L_2 \setminus \mathsf{adr}_i^{\mathsf{read}}$. This is indistinguishable because of the security of PRF function.
- Hyb_2: $\mathcal{S}_{\mathsf{DORAM}}$ replies with random additive shares in Access phase, i.e. a random $\mathsf{st_{stash}}$, $\mathsf{adr_M}$, $\langle \mathsf{stash} \rangle$, and $\langle \mathsf{M} \rangle$ (when $c = \sqrt{n} - 1$). This is indistinguishable because of security of additive sharing.
- Hyb_3 : During Init, the simulator samples list L_2 at random from $\mathcal{Y}$ (without replacement) rather than performing PRF evaluation on $[m]$. This is indistinguishable because of the security of PRF scheme, and since it is assumed to have negligible probability of collision.
- Hyb_4 : $\mathcal{S}_{\mathsf{DORAM}}$ replies calls to $\mathcal{F}_{\mathsf{r\text{-}M}}(\mathsf{Init})$ by returning two randomly sampled PRF keys (k_1, k_3). This is identical to the previous hybrid.

Observe that the view produced in the last hybrid is exactly the one generated by $\mathcal{S}_{\mathsf{DORAM}}$. □

A.5 Security Proof of Π_{stash}

The functionality for reading and writing to stash ($\mathcal{F}_{\mathsf{Stash}}$) appears in Fig. 16.

Theorem 4. *Protocol Π_{stash} UC-securely instantiates functionality $\mathcal{F}_{\mathsf{Stash}}$ in presence of one passive corruption in the $(\mathcal{F}_{\mathsf{DOPRF}}, \mathcal{F}_{\mathsf{Select}}, \mathcal{F}_{\mathsf{Zero}})$-hybrid model.*

Proof. Correctness and security for the simulator are argued as follows.

$\mathcal{S}_{\mathsf{DORAM}}$

Init/Refresh: On receiving corrupt party's inputs $\langle \mathsf{M} \rangle$, call $\mathcal{F}_{\mathsf{DORAM}}$ on $(\mathsf{Init}, \langle \mathsf{M} \rangle)$. Simulate transcript as follow.

1. On behalf of P_1, P_3, set access counter $c := 0$, $\langle \mathsf{stash} \rangle := [(0,0,0)]^{\sqrt{n}}$, and set $\langle \mathsf{st}_{\mathsf{stash}} \rangle := \perp$, where $\mathsf{st}_{\mathsf{stash}} : (\mathsf{flag}, \mathsf{loc}, \mathsf{val}_{\mathsf{st}})$.
2. From P_2, receive call to $\mathcal{F}_{\mathsf{r\text{-}M}}$ with $(\mathsf{Init}, \langle \mathsf{M} \rangle)$. Sample two PRF keys $k_1, k_3 \leftarrow \mathcal{K}$.
3. Sample without replacement a list $L_2 \leftarrow \mathcal{Y}^m$. Sort L_2. Reply with $(k_1, k_3), L_2$.
4. Set the list of addresses read $\mathbf{adr}_1^{\mathsf{read}}, \mathbf{adr}_3^{\mathsf{read}} := \perp$.

Access: On receiving inputs of the corrupt party for the current iteration, $\langle \mathsf{I} \rangle$, call $\mathcal{F}_{\mathsf{DORAM}}$ on Access to receive output d_{out}. Simulate transcript messages as follows:

1. From P_2, receive call to $\mathcal{F}_{\mathsf{Stash}}$ with $(\mathsf{Read}, c, \langle \mathsf{stash} \rangle, \langle \mathsf{I} \rangle)$. Sample $\langle \mathsf{st}_{\mathsf{stash}} \rangle$ uniformly at random from the appropriate domain. Reply with $\langle \mathsf{st}_{\mathsf{stash}} \rangle$.
2. From P_2, receive call to $\mathcal{F}_{\mathsf{Select}}$ with $(\langle \mathsf{flag} \rangle, \langle c+n \rangle, \langle \mathsf{I.adr} \rangle)$. Set $\langle \mathsf{adr}_{\mathsf{M}} \rangle \leftarrow \mathcal{Y}$, and reply with $\langle \mathsf{adr}_{\mathsf{M}} \rangle$.
3. Receive call to $\mathcal{F}_{\mathsf{r\text{-}M}}$ with $(\mathsf{Access}, \langle \mathsf{adr}_{\mathsf{M}} \rangle, \mathbf{adr}_2^{\mathsf{read}}, k_1, k_3)$, and do:
   (a) Sample $\mathsf{adr} \leftarrow L_2 \setminus \mathbf{adr}_2^{\mathsf{read}}$.
   (b) Update $\mathbf{adr}_2^{\mathsf{read}} := \mathbf{adr}_2^{\mathsf{read}} \cup \mathsf{adr}$.
   (c) Sample $\langle \mathsf{val}_{\mathsf{M}} \rangle \leftarrow \mathbb{F}_p$.
   (d) Reply with $(\langle \mathsf{val}_{\mathsf{M}} \rangle, \mathbf{adr}_2^{\mathsf{read}})$.
4. Receive call to $\mathcal{F}_{\mathsf{Stash}}$ with $(\mathsf{Write}, \langle \mathsf{val}_{\mathsf{M}} \rangle, \langle \mathsf{adr}_{\mathsf{M}} \rangle, c, \langle \mathsf{stash} \rangle, \langle \mathsf{st}_{\mathsf{stash}} \rangle, \langle \mathsf{I} \rangle)$ from P_2. Sample $\langle \mathsf{stash} \rangle$ uniformly at random from the appropriate domain.
5. Receive call to $\mathcal{F}_{\mathsf{Select}}$ with $(\langle \mathsf{flag} \rangle, \langle \mathsf{val}_{\mathsf{st}} \rangle, \langle \mathsf{val}_{\mathsf{M}} \rangle)$ and reply with d_{out}.
6. If $c = \sqrt{n} - 1$, do:
   (a) Receive call to $\mathcal{F}_{\mathsf{w\text{-}M}}$ with $(\langle \mathsf{M} \rangle, \langle \mathsf{stash} \rangle, L_2, \mathbf{adr}_2^{\mathsf{read}}, k_1, k_3))$ from P_2.
   (b) Sample $\langle \mathsf{M} \rangle \leftarrow \mathbb{F}_p^n$ and send it.
   (c) Run simulation steps for $\mathsf{Refresh}$.
7. Increment $c := c + 1$.

Get Memory: On receiving command GetM for the corrupt party, call $\mathcal{F}_{\mathsf{DORAM}}$ on GetM and receive updated share $\langle \mathsf{M}_{\mathsf{out}} \rangle$. Simulate transcript as follows. Receive call to $\mathcal{F}_{\mathsf{w\text{-}M}}$ with inputs $(\langle \mathsf{M} \rangle, \langle \mathsf{stash} \rangle, \mathbf{adr}_2^{\mathsf{read}}, k_1, k_3)$ from P_2. Reply with $\langle \mathsf{M}_{\mathsf{out}} \rangle$.

Fig. 15: Simulator for Theorem 3 in the case of corrupted P_2.

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

Parameters: Stash size $\sqrt{n}$, a memory size n.

Read Stash: On receiving $(\mathsf{Read}, c, \langle \mathsf{stash} \rangle, \langle \mathsf{I} \rangle)$ from all the parties,

1. Reconstruct stash and I.
2. If $\exists j$ such that $\mathsf{stash}[j].\mathsf{adr} = \mathsf{I.adr}$, then set $\mathsf{flag} := 1$, $\mathsf{val}_{\mathsf{st}} := \mathsf{stash}[j].\mathsf{val}$, and $\mathsf{loc} := j$. Else, set $\mathsf{flag} := 0$, $\mathsf{val}_{\mathsf{st}} := 0$, and $\mathsf{loc} := c$.
3. Set $\mathsf{st}_{\mathsf{stash}} = (\mathsf{flag}, \mathsf{loc}, \mathsf{val}_{\mathsf{st}})$ and output $\langle \mathsf{st}_{\mathsf{stash}} \rangle$ to all the parties.

Write in Stash: On receiving $(\mathsf{Write}, \langle \mathsf{val}_{\mathsf{M}} \rangle, \langle \mathsf{adr}_{\mathsf{M}} \rangle, c, \langle \mathsf{stash} \rangle, \langle \mathsf{st}_{\mathsf{stash}} \rangle, \langle \mathsf{I} \rangle)$ from all parties,

1. Reconstruct $\mathsf{val}_{\mathsf{M}}, \mathsf{adr}_{\mathsf{M}}, \mathsf{stash}, \mathsf{st}_{\mathsf{stash}}$, and I. Parse $\mathsf{st}_{\mathsf{stash}}$ as $(\mathsf{flag}, \mathsf{loc}, \mathsf{val}_{\mathsf{st}})$.
2. Update $\mathsf{stash}[c].\mathsf{adr} := \mathsf{adr}_{\mathsf{M}}$, $\mathsf{stash}[c].\mathsf{val} := \mathsf{val}_{\mathsf{M}}$, and $\mathsf{stash}[c].\mathsf{val}_{\mathsf{old}} := \mathsf{val}_{\mathsf{M}}$.
3. If $\mathsf{I.op} = \mathsf{write}$, update $\mathsf{stash}[\mathsf{loc}].\mathsf{val} := \mathsf{I.val}$.
4. Output shares $\langle \mathsf{stash} \rangle$ to the parties.

Fig. 16: Functionality for three parties, P_1, P_2, P_3, to read and write to the stash.

Correctness. First consider the correctness of read operation. The list L_{stash} maintained by P_2, P_3 is a list of PRF evaluations of addresses that were read previously from the memory, i.e., $\mathsf{adr}_{\mathsf{M}}$, under key k_{stash} that is known to P_1. The correctness of L_{stash} is guaranteed by correctness of write operation. In each iteration a new address is added to L_{stash}. In the start of read operation, suppose address adr needs to be read. In case it was previously read, $\mathsf{PRF}(k_{\mathsf{stash}}, \mathsf{adr})$ is present in L_{stash}. $\mathsf{adr}' := \mathsf{PRF}(k_{\mathsf{stash}}, \mathsf{adr}) + R$ is revealed to P_1, where P_2, P_3 know R. P_1 generates DPF keys which P_2, P_3 evaluate on L_{stash} after shifting each entry in L_{stash} by R. Thus if $\mathsf{PRF}(k_{\mathsf{stash}}, \mathsf{adr})$ is present in L_{stash}, so is $\mathsf{PRF}(k_{\mathsf{stash}}, \mathsf{adr}) + R$, and evaluating the key generates additive secret shares of 1 between P_2, P_3. Otherwise, it generates shares of 0. This gives correct shares for flag. Similarly, since the evaluation of keys at each entry in L_{stash} is multiplied by the corresponding index, if the address is present (say at index j) in L_{stash}, P_2, P_3 get secret shares of $\mathsf{loc} := j$, otherwise they receive shares of 0. Since, they select between this and c based on flag, this gives them the correct shares of loc.

Before computing shares of $\mathsf{val_{st}}$, parties convert additive shares of $\mathsf{stash.val}$ to replicative shares. In the following, let us consider the protocol from point of view of P_2. After replicated sharing, P_1, P_3 hold a common share $\langle \mathsf{stash.val} \rangle_3$. Suppose address to be read is present in stash. Then from previous argument, the parties hold secret shares of $\mathsf{loc} := j$, otherwise, they hold shares of $\mathsf{loc} := c$. A shifted $\mathsf{loc}_2 := \mathsf{loc}_2 + r_2$ is revealed to P_2, where P_1, P_3 know r_1. P_2 generates DPF keys for loc, and P_1, P_3 execute it on their common share $\langle \mathsf{stash.val} \rangle_3$ after shifting each index by r_i. If $\mathsf{loc} := j$, then they hold additive shares of just one out of three shares of $\langle \mathsf{val} \rangle$. Otherwise, it is a share of 0. This is done thrice, to obtain the final share val.

For write operation, first L_{stash} is updated with the newest read address $\mathsf{adr_M}$ by calling $\mathcal{F}_{\mathsf{DOPRF}}$. Since a new address $\mathsf{adr_M}$ is always made, parties add this in stash at the fixed location c. If adr was read in a previous iteration, from correctness of read (as argued above), $\mathsf{flag} := 1, \mathsf{loc} := j, \mathsf{val_{st}} := \mathsf{stash}[j].\mathsf{val}$, and, $\mathsf{flag} := 0, \mathsf{loc} := c, \mathsf{val_{st}} := 0$, otherwise. If $\mathsf{op} = \mathsf{write}$, then $\mathsf{stash}[j].\mathsf{val}$ should be updated to $\mathsf{I.val}$ (if $\mathsf{flag} = 1$), and otherwise, $\mathsf{stash}[c].\mathsf{val}$ should be update to $\mathsf{I.val}$ (if $\mathsf{flag} = 0$). In the first case $\Delta := \mathsf{I.val} - \mathsf{stash}[j].\mathsf{val}$, while in the second case, $\Delta := \mathsf{I.val} - \mathsf{val_M}$. Moreover, if $\mathsf{op} = \mathsf{read}$, then $\mathsf{stash.val}$ should remain unchanged, i.e., $\Delta := 0$. To compute Δ, parties obtain the *old* value ($\mathsf{val_{st}} := \mathsf{stash}[j].\mathsf{val}$ or $\mathsf{val_M}$) using the first select operation with flag. Using the second $\mathcal{F}_{\mathsf{Select}}$, they obtain the *new* value ($\mathsf{I.val}$ or $\mathsf{val_{old}}$) depending on op. Observe that, if $\mathsf{op} = \mathsf{write}$, we have $\mathsf{val_{old}} = \mathsf{stash}[j].\mathsf{val}$ (for $\mathsf{flag} = 1$), and $\mathsf{val_{old}} = \mathsf{val_M}$ (for $\mathsf{flag} = 0$), and $\mathsf{val_{new}} = \mathsf{I.val}$. This gives the right Δ for $\mathsf{op} = \mathsf{write}$. On the other hand, if $\mathsf{op} = \mathsf{read}$, we have $\mathsf{val_{new}} = \mathsf{val_{old}}$ which makes $\Delta = 0$. This gives the correct values to generate DPF keys.

Security. The simulator is described in Fig. 17. We argue security for the case when P_2 is corrupt. The case for P_3 is analogous. In the hybrids, we start with replacing real protocol steps with simulation steps for write, followed by steps for read. This is to maintain correctness of simulation output in each hybrid. We give hybrids for just one read, and write operation. These steps can be repeated for multiple, interleaved read and write operations. While repeating steps for multiple iteration, $\mathcal{S}_{\mathsf{stash}}$ will internally maintain the set L_{stash}, and in each iteration, it will sample the response for $\mathcal{F}_{\mathsf{DOPRF}}$ call (in Step 1 of write) by excluding entries in L_{stash}. Other than this, $\mathcal{S}_{\mathsf{stash}}$ does not need to record any other value across iterations. Additionally, in case of interleaved read, write operations, it will first simulate write and then read.

- Hyb_0: Same as the real world execution except that $\mathcal{S}_{\mathsf{stash}}$ computes outputs differently (steps 10 and 11). It answers calls to $\mathcal{F}_{\mathsf{Zero}}$ with $\langle \mathsf{stash_{out}} \rangle - \boldsymbol{\delta}_i - \langle \mathsf{stash.val} \rangle$, where $\boldsymbol{\delta}_i$ and $\langle \mathsf{stash.val} \rangle$ are exactly as in the real world execution.
- Hyb_1: $\mathcal{S}_{\mathsf{stash}}$ answers calls to $\mathcal{F}_{\mathsf{DOPRF}}$ by sampling a random x' with the constraint that it was not sampled in previous executions of write (Step 1). This is indistinguishable because PRF evaluations look indistinguishable form random.
- Hyb_2: $\mathcal{S}_{\mathsf{stash}}$ replies to the calls to $\mathcal{F}_{\mathsf{Select}}$ with randomly sampled y' and z' (steps 2 and 3). This is identical to the previous hybrid as the functionalities return additive shares which look indistinguishable from a random element.
- Hyb_3 : $\mathcal{S}_{\mathsf{stash}}$ sends random w_0, w_2 (step 5) instead of correctly masked values. This is indistinguishable because the masks are random and unknown to P_1.
- Hyb_4 : $\mathcal{S}_{\mathsf{stash}}$ generates DPF keys by running the simulator $\mathcal{S}$ (Step 6). This is indistinguishable from real keys because of the security of the DPF scheme.
- Hyb_5: $\mathcal{S}_{\mathsf{stash}}$ computes outputs differently. $\mathcal{S}_{\mathsf{stash}}$ answers the call to $\mathcal{F}_{\mathsf{Select}}$ and replies with $\langle \mathsf{loc} \rangle$ (Step 5). It then computes y as P_1 would have (Step 5 in Π_{stash}), and then answers call to $\mathcal{F}_{\mathsf{Zero}}$ with $\langle \mathsf{flag} \rangle - y$. It continues executing steps in $\Pi_{\mathsf{r\text{-}st}}$ and computes v as P_1 would have (Step 10d in Π_{stash}). It answers call $\mathcal{F}_{\mathsf{Zero}}$ with $\langle \mathsf{val} \rangle - v$ (Step 14). These changes are indistinguishable because of correctness.
- Hyb_6: $\mathcal{S}_{\mathsf{stash}}$ answers the call to $\mathcal{F}_{\mathsf{DOPRF}}$ by sampling a random R (Step 2). This is indistinguishable because of the security of additive shares.
- Hyb_7: $\mathcal{S}_{\mathsf{stash}}$ sends random u_0, u_2 (Step 10) instead of correctly masked values. This is indistinguishable because the masks are random and unknown to P_1.

– Hyb_8: $\mathcal{S}_{\mathsf{stash}}$ sends a random vector $\boldsymbol{z}$ to P_1 (Step 8) This is indistinguishable, again, because of the additive secret sharing.
– Hyb_9: $\mathcal{S}_{\mathsf{stash}}$ generates DPF keys by running the simulator $\mathcal{S}$ (steps 4 and 11). This is indistinguishable from real keys because of the security of the DPF scheme.

Observe that the view generated in the last hybrid is identical to the one generated by $\mathcal{S}_{\mathsf{stash}}$.

□

$\mathcal{S}_{\mathsf{stash}}$

Parameters: Stash size $\sqrt{n}$, memory size n. A PRF: $\mathcal{K} \times [m] \to \mathcal{Y}$. Three DPF schemes: Φ_1 for functions $\mathcal{Y} \to \{0,1\}$, Φ_2 for $\mathbb{F}_p \to \{0,1\}$, and Φ_3 for $\mathbb{F}_p \to \mathbb{F}_p$. $\mathcal{S}$ is the simulator for the DPF schemes.
Read in Stash: On receiving corrupt party's inputs $c, \langle \mathsf{stash} \rangle, \langle \mathsf{I} \rangle$, call $\mathcal{F}_{\mathsf{Stash}}$ with Read and the inputs to receive outputs $\langle \mathsf{st}_{\mathsf{stash}} \rangle = (\langle \mathsf{flag} \rangle, \langle \mathsf{loc} \rangle, \langle \mathsf{val}_{\mathsf{st}} \rangle)$. Parse $\langle \mathsf{I} \rangle$ as $(\langle \mathsf{op} \rangle, \langle \mathsf{adr} \rangle, \langle \mathsf{val} \rangle)$. Simulate transcript as follows:
1. If $c = 1$: $\mathcal{S}_{\mathsf{stash}}$ initializes an empty list L_{stash} on behalf of P_3. Receives call to $\mathcal{F}_{\mathsf{DOPRF}}$ from P_2 and replies nothing. If P_1 is corrupt, sample a PRF key $k_{\mathsf{stash}} \leftarrow \mathcal{Y}$ and reply to P_1 with k_{stash}.
2. Receive call to $\mathcal{F}_{\mathsf{DOPRF}}$ with input $(\mathsf{Eval}, \mathsf{masked}, \langle \mathsf{adr} \rangle)$ from P_2, sample $R \leftarrow F_p$ and send it. If P_1 was corrupt, receive k_{stash} in addition and send a random $x \leftarrow F_p$.
3. Receive $x + R_1$ from P_2 on behalf of P_1. If P_1 is corrupt, sample $w_1, w_2 \leftarrow \mathbb{F}_p$ and send it to P_1 on behalf of P_2, P_3.
4. Run simulator $k^1_{\mathsf{DPF}} \leftarrow \mathcal{S}(1^\lambda, \mathcal{Y}_{\mathsf{PRF}}, \{0,1\})$. Send k^1_{DPF} to P_2.
5. Receive call to $\mathcal{F}_{\mathsf{Select}}$ and reply with $\langle \mathsf{loc} \rangle$ received from $\mathcal{F}_{\mathsf{Stash}}$.
6. For $j \in [|\mathsf{stash}|]$ compute $y := x + \mathsf{Eval}(k_1, L_{\mathsf{stash}}[j] + R)$.
7. Receive call to $\mathcal{F}_{\mathsf{Zero}}$ and reply with $\langle 0 \rangle := \langle \mathsf{flag} \rangle - y$, where $\langle \mathsf{flag} \rangle$ is the value received from $\mathcal{F}_{\mathsf{Stash}}$.
8. Receive $\langle \mathsf{stash.val} \rangle_1$ from P_2 on behalf of P_3. Sample $\boldsymbol{z} \leftarrow \mathbb{F}_p^{|\mathsf{stash}|}$ and send it to P_2.
9. Sample $r_{3,1}, r_{2,1} \leftarrow \mathbb{F}_p^2$ and send it to P_2 on behalf of P_3. Receive $r_{2,3}, r_{1,3}$ from P_2 on behalf of P_1.
10. Send $u_1, u_3 \leftarrow \mathbb{F}_p^2$ to P_2 on behalf of P_1, P_3, respectively. Receive $\langle \mathsf{loc} \rangle_1 + r_{2,1}$, $\langle \mathsf{loc} \rangle_1 + r_{2,3}$ from P_2 on behalf of P_1, P_3, respectively.
11. Run simulator $k_{1,2} \leftarrow \mathcal{S}(1^\lambda, \mathbb{F}_p, \{0,1\})$, and $k_{3,2} \leftarrow \mathcal{S}(1^\lambda, \mathbb{F}_p, \{0,1\})$. Send $k_{1,2}, k_{3,2}$ to P_2 on behalf of P_1, P_3. Receive $k_{2,1}, k_{2,3}$ from P_2 on behalf of P_1, P_3, respectively.
12. Define $r_1 := r_{3,1} + r_{2,1}$ and $r_3 := r_{2,3} + r_{1,3}$.
13. For $j \in [|\mathsf{stash}|]$, compute $v := v + (\mathsf{Eval}(k_{1,2}, j + r_1) \times \langle \mathsf{stash}[j].\mathsf{val} \rangle_1) + (\mathsf{Eval}(k_{3,2}, j + r_3) \times \boldsymbol{z}[j])$.
14. Receive call to $\mathcal{F}_{\mathsf{Zero}}$ from P_1. Reply with $\langle \mathsf{val}_{\mathsf{st}} \rangle - v$ where $\langle \mathsf{val}_{\mathsf{st}} \rangle$ is received from $\mathcal{F}_{\mathsf{Stash}}$.

Write in Stash: On receiving corrupt party's inputs $\langle \mathsf{val}_{\mathsf{M}} \rangle, \langle \mathsf{adr}_{\mathsf{M}} \rangle, c, \langle \mathsf{stash} \rangle, \langle \mathsf{st}_{\mathsf{stash}} \rangle, \langle \mathsf{I} \rangle$, call $\mathcal{F}_{\mathsf{Stash}}$ with the inputs to receive output $\langle \mathsf{stash}_{\mathsf{out}} \rangle$. Simulate transcript as follows:
1. Receive call to $\mathcal{F}_{\mathsf{DOPRF}}$ from P_2, and reply with $x' \leftarrow \mathcal{Y} \setminus L_{\mathsf{stash}}$.
2. Receive call to $\mathcal{F}_{\mathsf{Select}}$ from P_2 and reply with $y' \leftarrow \mathbb{F}_p$.
3. Receive call to $\mathcal{F}_{\mathsf{Select}}$ from P_2 and reply with $z' \leftarrow \mathbb{F}_p$.
4. Sample $\rho_{3,1}, \rho_{2,1} \leftarrow \mathbb{F}_p^2$ and send it to P_2 on behalf of P_3. Receive $\rho_{2,3}, \rho_{1,3}$ from P_2 on behalf of P_1.
5. Send $w_1, w_3 \leftarrow \mathbb{F}_p^2$ to P_2 on behalf of P_1, P_3, respectively. Receive $\langle \mathsf{loc} \rangle_1 + \rho_{2,1}$, $\langle \mathsf{loc} \rangle_1 + \rho_{2,3}$ from P_2 on behalf of P_1, P_3, respectively.
6. Run simulator $k_{1,2} \leftarrow \mathcal{S}(1^\lambda, \mathbb{F}_p, \{0,1\})$, and $k_{3,2} \leftarrow \mathcal{S}(1^\lambda, \mathbb{F}_p, \{0,1\})$. Send $k_{1,2}, k_{3,2}$ to P_2 on behalf of P_1, P_3. Receive $k_{2,1}, k_{2,3}$ from P_2 on behalf of P_1, P_3, respectively.
7. Initialize a vector $\boldsymbol{\delta}$ of length $|\mathsf{stash}|$ as $(0, \ldots, 0)$.
8. Define $\rho_1 := \rho_{2,1} + \rho_{3,1}$, and $\rho_3 := \rho_{2,3} + \rho_{1,3}$.
9. For $j \in [|\mathsf{stash}|]$, compute $\boldsymbol{\delta}[j] := \boldsymbol{\delta}[j] + \mathsf{Eval}(k_{1,2}, j + \rho_1) + \mathsf{Eval}(k_{3,2}, j + \rho_3)$.
10. Receive call to $\mathcal{F}_{\mathsf{Zero}}$ from P_2. Reply with $\langle \mathsf{stash}_{\mathsf{out}}.\mathsf{val} \rangle - \boldsymbol{\delta} - \langle \mathsf{stash.val} \rangle$.
11. Receive two calls to $\mathcal{F}_{\mathsf{Zero}}$ from P_2. Reply with $\langle \mathsf{stash}_{\mathsf{out}}.\mathsf{adr} \rangle - \langle \mathsf{stash.adr} \rangle$ and $\langle \mathsf{stash}_{\mathsf{out}}.\mathsf{val}_{\mathsf{old}} \rangle - \langle \mathsf{stash.val}_{\mathsf{old}} \rangle$.

Fig. 17: Simulator for Theorem 4 for reading and writing to stash.

A.6 Security Proof of $\Pi_{\mathsf{r\text{-}M}}$

Theorem 5. *Protocol $\Pi_{\mathsf{r\text{-}M}}$ UC-securely instantiates functionality $\mathcal{F}_{\mathsf{r\text{-}M}}$ in the presence of one passive corruption in the $(\mathcal{F}_{\mathsf{DOPRF}}, \mathcal{F}_{\mathsf{3\text{-}OT}})$-hybrid model.*

Functionality $\mathcal{F}_{\text{r-M}}$

Parameters: A $\text{PRF} : \mathcal{K} \times [m] \to \mathcal{Y}$.
Init: On receiving command $(\text{Init}, \langle \mathsf{M} \rangle)$ as input from all the parties,
1. Reconstruct and store tuple (M) locally.
2. For $i \in \{1, 2, 3\}$,
   (a) Sample a PRF key k_i.
   (b) Compute list $L_i := \text{PRF}(k_i, j)$ for $j \in [m]$, and sort it to obtain L_i.
   (c) Output k_i to parties P_{i-1}, P_{i+1}, and L_i to P_i.

Access: On receiving $(\text{Access}, \langle \text{adr}_{\mathsf{M}} \rangle)$ from all parties, and $\textbf{adr}_i^{\text{read}}, k_{i-1}, k_{i+1}$ from each party P_i,
1. Reconstruct adr_{M}.
2. $d := \mathsf{M}[\text{adr}_{\mathsf{M}}].\text{val}$, where M is stored at id.
3. For $i \in \{1, 2, 3\}$, update $\textbf{adr}_i^{\text{read}} := \textbf{adr}_i^{\text{read}} \cup \text{PRF}(k_i, \text{adr}_{\mathsf{M}})$. Output $(\langle d \rangle, \textbf{adr}_i^{\text{read}})$ to party P_i.

Fig. 18: Functionality for parties P_1, P_2, P_3 to read from a secret address in memory M obliviously.

Proof. Correctness and Security are argued below.

Correctness. It follows from the correctness of $\mathcal{F}_{\text{3-OT}}$, $\mathcal{F}_{\text{DOPRF}}$, and that there are no collisions in PRF evaluation of $\{1, \ldots, m\}$.

Security. Now, we argue security. W.l.o.g assume P_2 is corrupt. The simulator for this case appears in Fig. 19. In the real protocol execution, Init is called only once and Access can be called a number of times. However, we describe the hybrids only for Init and *a single* invocation of Access, which can be replicated for multiple invocations in a straight forward way. The simulator for the case of corrupt P_2 appears in Fig. 6. $\mathcal{S}_{\text{r-M}}$ generates an indistinguishable transcript because of the following hybrids.

- Hyb_0: Same as the real world execution except that the simulator sets $\langle x \rangle$ as $\langle d \rangle - y$, where $\langle d \rangle$ is obtained from $\mathcal{F}_{\text{r-M}}$. This is indistinguishable as in the real execution $\langle d \rangle := x + y$ as well.
- Hyb_1: Reply to $\mathcal{F}_{\text{DOPRF}}$ is set as $\text{adr}' = \text{adr}^{\text{read}} \setminus \text{adr}_2^{\text{read}}$, where adr^{read} is obtained form $\mathcal{F}_{\text{r-M}}$. Again, this is indistinguishable from the previous hybrid because there too this set is updated similarly.
- Hyb_3: Same as Hyb_1 except M^* is picked at random. This is indistinguishable from Hyb_1 because the output of $\mathcal{F}_{\text{3-OT}}$ is a random vector which is unknown to P_2 (since P_1 acts as S and P_1 is honest).

Observe that the view generated in the last hybrid is exactly the one generated by $\mathcal{S}_{\text{r-M}}$. □

$\mathcal{S}_{\text{r-M}}$

Parameters: A $\text{PRF} : \mathcal{K} \times [m] \to \mathcal{Y}$.
Init: On receiving corrupt party's inputs $\langle \mathsf{M} \rangle$, call $\mathcal{F}_{\text{r-M}}$ on command Init and input to receive output $(k_1, k_3), L_2$. Simulate transcript messages as follows:
1. Receive calls to $\mathcal{F}_{\text{DOPRF}}$ with input (KeyGen, P_1) from P_2. If P_2 acts as K in call to $\mathcal{F}_{\text{DOPRF}}$, then return nothing. Else, return $(k_1, k_3), L_2$ to P_2.
2. Receive call to $\mathcal{F}_{\text{3-OT}}$ with input Init. If P_2 acts as the sender S, accept vector $\boldsymbol{r}_3$.
3. From P_2, accept $\langle \mathsf{M}' \rangle$ on behalf of P_3. Sample $\mathsf{M}^* \leftarrow \mathbb{F}_p^m$. Send $(L_2[j], \mathsf{M}^*[j])$ to P_2, for $j \in [m]$.

Access: On receiving corrupt party's inputs $\langle \text{adr}_{\mathsf{M}} \rangle, \text{adr}_2^{\text{read}}, k_1, k_3$, call $\mathcal{F}_{\text{r-M}}$ on Access and inputs to receive outputs $(\langle d \rangle, \text{adr}^{\text{read}})$. Simulate transcript as follows:
1. Receive call to $\mathcal{F}_{\text{DOPRF}}$ with $(\text{Eval}, \text{unmasked}, \langle \text{adr}_{\mathsf{M}} \rangle)$ from P_2. If P_2 acts as R_2, reply $\text{adr}' := \text{adr}^{\text{read}} \setminus \text{adr}_2^{\text{read}}$. Else, return nothing.
2. Receive calls to $\mathcal{F}_{\text{3-OT}}$ with input Online. If P_2 acts as the receiver R, set $y := \mathsf{M}^*[L_2^{-1}(\text{adr}')]$. Return $\langle x \rangle := \langle d \rangle - y$, where $\langle d \rangle$ is received form $\mathcal{F}_{\text{r-M}}$. Else, return nothing.

Fig. 19: Simulator for Theorem 5 for reading from memory.

A.7 Security Proof of $\Pi_{\text{w-M}}$

Theorem 6. *Protocol $\Pi_{\text{w-M}}$ UC-securely instantiates functionality $\mathcal{F}_{\text{w-M}}$ in the presence of one passive corruption in the ($\mathcal{F}_{\text{Zero}}$)-hybrid model.*

Functionality $\mathcal{F}_{\text{w-M}}$

Update: On receiving $(\langle \mathsf{M} \rangle, \langle \mathsf{stash} \rangle, L_i, \mathbf{adr}_i^{\text{read}}, k_{i-1}, k_{i+1})$ from each P_i,

1. Reconstruct M and stash.
2. For $j \in [\sqrt{n}]$, set $\mathsf{adr} := \mathsf{stash}[j].\mathsf{adr}$ and $\mathsf{val} := \mathsf{stash}[j].\mathsf{val}$. Update $\mathsf{M}[\mathsf{adr}].\mathsf{val} := \mathsf{val}$.
3. Output $\langle \mathsf{M} \rangle$ to all parties.

Fig. 20: Functionality for parties P_1, P_2, P_3 for updating memory M with stash entries.

$\mathcal{S}_{\text{w-M}}$

Parameter: An MPDPF scheme $\Phi = (\mathsf{Gen}, \mathsf{Eval})$ for input domain $[m]$ and output domain $\mathbb{F}_p$. Let $\mathcal{S}$ be the simulator for this scheme.

Update: On receiving corrupt party's inputs $\langle \mathsf{M} \rangle, \langle \mathsf{stash} \rangle, \mathbf{adr}_2^{\text{read}}, L_2, k_1, k_3$, call $\mathcal{F}_{\text{w-M}}$ on inputs to receive output $\langle \mathsf{M} \rangle$. Simulate transcript as follows.

1. Accept MPDPF keys $mk_{2,1}, mk_{2,3}$ on behalf of P_1 and P_3.
2. For $i \in \{1, 3\}$ do:
   (a) Run simulator $mk_{2,i} \leftarrow \mathcal{S}(1^\lambda, m, \mathbb{F}_p)$.
   (b) Send key $mk_{2,i}$ to P_2 on behalf of P_i.
3. Receive call to $\mathcal{F}_{\text{Zero}}(n)$,
   (a) For $j \in [n]$, evaluate: $\mathbf{x}[j] := \Phi.\mathsf{Eval}(mk_{1,2}, L_1^{-1}(\mathsf{PRF}(k_1, j)))$ and $\mathbf{y}[j] := \Phi.\mathsf{Eval}(mk_{3,2}, L_3^{-1}(\mathsf{PRF}(k_3, j)))$.
   (b) For $j \in [n]$, set $\boldsymbol{\alpha}_2[j] := \langle \mathsf{M} \rangle_2 [j] - \langle \mathsf{M} \rangle_2 [j] - \boldsymbol{x}[j] - \boldsymbol{y}[j]$.
   (c) Return $\boldsymbol{\alpha}_2$.

Fig. 21: Simulator for Theorem 6 in the case of corrupted P_2.

Proof. Correctness and security are argued as follows.

Correctness. It follows if there are no collisions in the PRF evaluations of the addresses, and from the correctness of the MPDPF scheme. Consider updating the memory with respect to just one party's share (say P_i's share). Because of correctness of MPDPF primitive, the update for an actual address adr is stored at (a unique) index $L_i^{-1}(\mathsf{PRF}(k_i, \mathsf{adr}))$ in the vector $\{\Phi.\mathsf{Eval}(mk_{i-1,i}, L_i^{-1}(\mathsf{PRF}(k_i, 0))) + \Phi.\mathsf{Eval}(mk_{i+1,i}, L_i^{-1}(\mathsf{PRF}(k_i, 0))), \ldots, \Phi.\mathsf{Eval}(mk_{i-1,i}, L_i^{-1}(\mathsf{PRF}(k_i, m-1))) + \Phi.\mathsf{Eval}(mk_{i+1,i}, L_i^{-1}(\mathsf{PRF}(k_i, m-1)))\}$. Also because of the correctness of MPDPF, this value is 0 in case this address was never accessed, and the corresponding update otherwise. Finally, because of uniqueness of L^{-1} all updates are recorded in this vector. Update to position j in M with one share, is then $\mathsf{M}[j].\mathsf{val} + \{\Phi.\mathsf{Eval}(mk_{i-1,i}, L_i^{-1}(\mathsf{PRF}(k_i, j))) + \Phi.\mathsf{Eval}(mk_{i+1,i}, L_i^{-1}(\mathsf{PRF}(k_i, j)))\}$, which is exactly what happens in the protocol. Repeating the above thrice updates the memory fully.

Security. Now we argue security of the protocol. We assume that P_2 is corrupt. The simulator $\mathcal{S}_{\text{w-M}}$ appears in Fig. 21, and we argue indistinguishability with the following hybrids.

- Hyb_0 : Same as the real world execution except that the output of the call to the functionality $\mathcal{F}_{\text{Zero}}$ is answered by setting $\boldsymbol{\alpha}_2$ as $\langle \mathsf{M}' \rangle_2 - \langle \mathsf{M} \rangle_2 - \boldsymbol{x} - \boldsymbol{y}$, where $\langle \mathsf{M}' \rangle_2$ is the value received from $\mathcal{F}_{\text{w-M}}$. This is indistinguishable since in the real protocol as well, $\langle \mathsf{M}' \rangle_2 := \boldsymbol{\alpha}_2 + \langle \mathsf{M} \rangle_2 + \boldsymbol{x} + \boldsymbol{y}$.

– Hyb_2 : Same as before except that the MPDPF simulator is called to generate keys $mk_{1,0}, mk_{1,2}$ instead of generating them keys honestly. This is indistinguishable because of the security of the MPDPF scheme.

Observe that the view generated in the last hybrid is exactly as the one generated by $\mathcal{S}_{\text{w-M}}$. □

B Distributed PRF Evaluation

Protocol Π_{DOPRF}

Parameters : Output length ℓ, a Prg that expands a seed of length l to l' field elements.
KeyGen: R_1 samples $\boldsymbol{k}_1 \leftarrow \mathbb{F}_p^\ell$, and sends it to K. R_2 samples $\boldsymbol{k}_2 \leftarrow \mathbb{F}_p^\ell$, and sends it to K. K sets $\boldsymbol{k} := \boldsymbol{k}_1 + \boldsymbol{k}_2$.
Init: Parties sample pairwise PRG seeds: K samples $k_{k,1} \leftarrow \{0,1\}^l$, R_1 samples $k_{1,2} \leftarrow \{0,1\}^l$, and R_2 samples $k_{2,k} \leftarrow \{0,1\}^l$, and sends it to R_1, R_2, and K, respectively.
Eval: Each party has input $\langle x \rangle$, and pairwise PRG seeds. K in addition has key k.

1. K and R_1 compute $\boldsymbol{s} \leftarrow \mathsf{PRG}(k_{k,1})$, where $\boldsymbol{s} \in \mathbb{F}_p^\ell$.
2. $\mathsf{R}_1, \mathsf{R}_2$ compute $m, b, \boldsymbol{d}, \boldsymbol{c}_3 \leftarrow \mathsf{PRG}(k_{1,2})$, where $m, b \in \mathbb{F}_p, \boldsymbol{d}, \boldsymbol{c}_3 \in \mathbb{F}_p^\ell$.
3. R_1 computes $\boldsymbol{a} := \boldsymbol{s} \circ \boldsymbol{s} - \boldsymbol{d}$, and $\boldsymbol{c}_1 := b \cdot \boldsymbol{a} - \boldsymbol{c}_3$. Send $\boldsymbol{c}_1$ to K.
4. R_1 computes and sends $y_2 := \langle x \rangle + m$ to K.
5. R_2 computes $y_3 := \langle x \rangle - m$, $e := y_3 - b$. R_2 sends e to K.
6. K computes $\boldsymbol{w} := (\boldsymbol{s} \circ \boldsymbol{s}) \cdot (\boldsymbol{k} + \langle x \rangle + y_2)$, and $\boldsymbol{z}_1 := e \cdot (\boldsymbol{s} \circ \boldsymbol{s}) + \boldsymbol{c}_1 + \boldsymbol{w}$. K sends $\boldsymbol{z}_1$ to R_2.
7. R_2 computes $\boldsymbol{z}_3 := \boldsymbol{d} \cdot y_3 + \boldsymbol{c}_3 - \boldsymbol{d} \cdot e$, and $\boldsymbol{z} := \boldsymbol{z}_1 + \boldsymbol{z}_3$.
8. R_2 computes, for $j \in [1, \ell]$, $\boldsymbol{o}[j] := \frac{1}{2}\big(\big(\frac{\boldsymbol{z}[j]}{p}\big) + 1\big) \bmod p$.
9. R_2 outputs $\boldsymbol{o}$.

Fig. 22: Protocol for secure evaluation of PRF.

Theorem 7. *Protocol Π_{DOPRF} UC-securely instantiates functionality $\mathcal{F}_{\mathsf{DOPRF}}$ for $\mathsf{mode} = \mathsf{unmasked}$ in the presence of one passive corruption.*

Proof. Correctness and security for the simulator are argued as follows.

Correctness. Let x_1, x_2, x_3 be $\mathsf{K}, \mathsf{R}_1, \mathsf{R}_2$'s additive share of x, resp. From the protocol,

$$\begin{aligned} \boldsymbol{c}_1 &= b \cdot (\boldsymbol{s} \circ \boldsymbol{s}) - b \cdot (\boldsymbol{d}) - \boldsymbol{c}_3 \\ \boldsymbol{w} &= (\boldsymbol{s} \circ \boldsymbol{s}) \cdot (\boldsymbol{k} + x_1 + x_2 + m) \end{aligned}$$

Substituting these in $\boldsymbol{z}_1$,

$$\begin{aligned} \boldsymbol{z}_1 &= (x_3 - m - b) \cdot (\boldsymbol{s} \circ \boldsymbol{s}) + b \cdot (\boldsymbol{s} \circ \boldsymbol{s}) \\ &\quad - b \cdot (\boldsymbol{d}) - \boldsymbol{c}_3 + (\boldsymbol{s} \circ \boldsymbol{s}) \cdot (\boldsymbol{k} + x_1 + x_2 + m) \\ &= (x + \boldsymbol{k}) \cdot (\boldsymbol{s} \circ \boldsymbol{s}) - b \cdot (\boldsymbol{d}) - \boldsymbol{c}_3 \end{aligned}$$

Substituting value for $\boldsymbol{z}_3$,

$$\begin{aligned} \boldsymbol{z}_3 &= \boldsymbol{d} \cdot y_3 + \boldsymbol{c}_3 - \boldsymbol{d} \cdot e \\ &= \boldsymbol{d} \cdot (x_3 - m) + \boldsymbol{c}_3 - \boldsymbol{d} \cdot (x_3 - m - b) \\ &= \boldsymbol{c}_3 + \boldsymbol{d} \cdot b \\ \boldsymbol{z} = \boldsymbol{z}_1 + \boldsymbol{z}_3 &= (x + \boldsymbol{k}) \cdot (\boldsymbol{s} \circ \boldsymbol{s}) \end{aligned}$$

The correctness of the final output is guaranteed by the multiplicative property of Legendre symbol, and since $L_p(\boldsymbol{s} \circ \boldsymbol{s}) = 1^\ell$, i.e., $L_p((x + \boldsymbol{k}) \cdot (\boldsymbol{s} \circ \boldsymbol{s})) = L_p(x + \boldsymbol{k}) L_p(\boldsymbol{s} \circ \boldsymbol{s})$. Note that, if for any j, there is some $s_j \in \boldsymbol{s}$ such that $s_j = 0$, then the correctness guarantees fail. However, this happens with negligible probability since $\boldsymbol{s}$ is chosen uniformly at random. Similarly, if for some j, $(k_j + x) = 0$ then as well the correctness cannot be guaranteed. However, if there is an environment that can select inputs $\langle x \rangle$ such that this happens for a randomly chosen k_j, then it can be used as an argument against hardness of computing shifted Legendre symbol. This reduction is similar as shown in [GRR+16] for their construction based on Legendre PRF.

Security. First, assume that K is corrupt. The simulator generates view in KGen as: on receiving output $\boldsymbol{k}$ from the functionality, sample $\boldsymbol{k}_1, \boldsymbol{k}_2 \leftarrow \mathbb{F}_p^\ell$ such that $\boldsymbol{k}_1 + \boldsymbol{k}_2 = \boldsymbol{k}$. Send $\boldsymbol{k}_1, \boldsymbol{k}_2$ to K on behalf of $\mathsf{R}_1, \mathsf{R}_2$. The simulator, in the Init phase, samples common PRG seeds with K: it receives $k_{k,1}$ on behalf of R_1, and sends $k_{2,k}$ to K on behalf of R_2. Then it receives inputs of the corrupt party $\langle x \rangle$, and key $\boldsymbol{k}$, and calls $\mathcal{F}_{\mathsf{DOPRF}}$ on $\mathsf{mode} = \mathsf{unmasked}$. It receives no output in return. It simulates the transcript as follows. It samples $\boldsymbol{c}_1 \leftarrow \mathbb{F}_p^\ell$ and $y_2 \leftarrow \mathbb{F}_p$, and sends them to K on behalf of R_2. It samples $e \leftarrow \mathbb{F}_p^\ell$ and sends it to K on behalf of R_2. Receive $\boldsymbol{w}, \boldsymbol{z}_1$ from K on behalf of R_2. This concludes the simulation. The transcript is indistinguishable from the real world experiment because each of the vectors and elements sent to K are masked with a secret random value, and thus look uniformly random to it: $\boldsymbol{c}_1$ is masked with $\boldsymbol{c}_3$, y_2 is masked with m, e is masked with b.

Now, assume that R_1 is corrupt. The simulator, in the Init phase, samples common PRG seeds with R_2: it receives $k_{1,2}$ on behalf of R_2, and sends $k_{\mathsf{K},1}$ to R_1 on behalf of K. Then it receives the corrupt party's input $\langle x \rangle$ and calls $\mathcal{F}_{\mathsf{DOPRF}}$ with $\mathsf{mode} = \mathsf{unmasked}$. It receives no output in return. Since R_1 receives no message in the protocol, the simulator simulates the transcript by simply accepting all messages sent by R_1 on behalf of K and R_2.

Finally, assume that R_2 is corrupt. The simulator , in the Init phase, samples common PRG seeds with R_2: it receives $k_{2,\mathsf{K}}$ on behalf of K, and sends $k_{1,2}$ to R_2 on behalf of R_1. Then it receives the corrupt party's input $\langle x \rangle$ and calls $\mathcal{F}_{\mathsf{DOPRF}}$ with $\mathsf{mode} = \mathsf{unmasked}$ to receive output $\boldsymbol{o}$. It simulates the transcript as follows. Receive e from R_2 on behalf of K. Locally compute $\boldsymbol{z}_3$ just as R_2 would, i.e., since $\boldsymbol{s}, \boldsymbol{d}, m, \boldsymbol{c}_3, b$ are sampled by R_2 using shared PRG seeds, the simulator can also obtain these values and compute y_3, e, and $\boldsymbol{z}_3$ just as R_2 would. It then sets $\boldsymbol{z}_1 := \boldsymbol{o} - \boldsymbol{z}_3$ and sends $\boldsymbol{z}_1$ to R_2. This concludes the simulation. Since the only message that R_2 receives in the protocol execution is $\boldsymbol{z}_1$, which is set in the particular way such that $\boldsymbol{o} = \boldsymbol{z}_1 + \boldsymbol{z}_3$, it is distributed identically to the real world view. ☐

Theorem 8. *Protocol Π^m_{DOPRF} UC-securely instantiates functionality $\mathcal{F}_{\mathsf{DOPRF}}$ for $\mathsf{mode} = \mathsf{masked}$ in the presence of one passive corruption.*

Proof. Correctness and security for the simulator are argued as follows.

Correctness. Substituting all the values for $\boldsymbol{c}_3, \boldsymbol{w}$, we get

$$\begin{aligned} \boldsymbol{z}_3 &= (\boldsymbol{k} + x_1 - m - \boldsymbol{a}) \cdot (\boldsymbol{t}) + (\boldsymbol{a}) \circ (\boldsymbol{t} - \boldsymbol{e}) - \boldsymbol{c}_1 + \boldsymbol{t} \cdot (x_1 + x_3 + m) \\ &= (\boldsymbol{k} + x) \cdot \boldsymbol{t} - \boldsymbol{a} \circ \boldsymbol{e} - \boldsymbol{c}_1 \end{aligned}$$

Similarly, for $\boldsymbol{z_1}$, and $\boldsymbol{z}$ we get,

$$\boldsymbol{z} = \boldsymbol{z}_1 + \boldsymbol{z}_3 = (\boldsymbol{k} + x) \cdot \boldsymbol{t}$$

Consider a single bit $r_j \in \boldsymbol{r}$, and corresponding $t_j \in \boldsymbol{t}$, $k_j \in \boldsymbol{k}$, $s_j \in \boldsymbol{s}$, $o_j \in \boldsymbol{o}$. If $r_j = 0$, then $t_j = s_j^2$, and $L_p(t_j) = 1$. Thus, $o_j = L_p(k_j + x)$, and $o_j \oplus r_j = L_p(k_j + x)$ (since $r_j = 0$). Else if $r_j = 1$, $t_j = s_j^2 \cdot \alpha$. If $(k_j + x)$ is a quadratic residue modulo p then $t_j \cdot (k_j + x)$ is not. Which means $L_p(t_j \cdot (k_j + x)) = 0$. Thus, $o_j \oplus r_j = L_p(k_j + x) = 1$. On the other hand, if $(k_j + x)$ is not a quadratic residue modulo p then $t_j \cdot (k_j + x)$ is. Thus, $L_p(t_j \cdot (k_j + x)) = 1$, and $o_j \oplus r_j = L_p(k_j + x) = 0$. This logic can be repeated for each output bit. Here too, the correctness argument fails in the cases discussed in the unmasked version of the protocol, and once again, it can be argued that it either happens with negligible probability or the assumption that shifted Legendre symbol is hard to compute does not hold.

Security. The simulator generates view in KGen as: on receiving output $\boldsymbol{k}$ from the functionality, sample $\boldsymbol{k}_2, \boldsymbol{k}_3 \leftarrow \mathbb{F}_p^\ell$ such that $\boldsymbol{k}_2 + \boldsymbol{k}_3 = \boldsymbol{k}$. Send $\boldsymbol{k}_2, \boldsymbol{k}_3$ to K on behalf of $\mathsf{R}_2, \mathsf{R}_2$. In the Init phase, the simulator sets up common PRG seeds just as in the unmasked version. We skip that detail here.

Suppose that K is corrupt. The simulator receives corrupt party's input $\langle x \rangle, \boldsymbol{k}$ and calls $\mathcal{F}_{\mathsf{DOPRF}}$ with $\mathsf{mode} = \mathsf{masked}$ and obtains output $\boldsymbol{y} \oplus \boldsymbol{r}$. Receive $\boldsymbol{d}$ on behalf of R_2. Given common PRG seeds and inputs of the corrupt party, the simulator can locally compute $\boldsymbol{z}_1$ just as K would have. It then sets $\boldsymbol{z}_3 := \boldsymbol{y} \oplus \boldsymbol{r} - \boldsymbol{z}_1$ and sends it to K. This is indistinguishable form the real world because of the programming of $\boldsymbol{z}_3$ and because of correctness of the scheme.

Next, suppose that R_1 is corrupt. The simulator receives input $\langle x \rangle$ and calls $\mathcal{F}_{\mathsf{DOPRF}}$ to receive $\boldsymbol{r}$ as output. The simulator receives $\boldsymbol{r}_1$ from R_1 on the behalf of R_2, sets $\boldsymbol{r}_2 := \boldsymbol{r} - \boldsymbol{r}_1$ and sends it to R_1. This concludes simulation and is clearly indistinguishable form the real world.

Finally, suppose R_2 is corrupt. The simulator receives input $\langle x \rangle$ and calls $\mathcal{F}_{\mathsf{DOPRF}}$ to receive $\boldsymbol{r}$ as output, and, just as in the previous case, fixes $\boldsymbol{r}_1 := \boldsymbol{r} - \boldsymbol{r}_2$. It then samples $y_2 \leftarrow \mathbb{F}_p, \boldsymbol{c}_3, \boldsymbol{d} \leftarrow \mathbb{F}_P^\ell$ to R_2. This is indistinguishable as all three values are masked by $m, \boldsymbol{c}_1$ and $\boldsymbol{a}$, resp. □

C Other Functionalities

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

Select: On receiving $(\langle \mathsf{flag} \rangle, \langle x \rangle, \langle y \rangle)$ from all the parties,

1. Reconstruct x, y, and flag.
2. $d = \mathsf{flag} \cdot x + (1 - \mathsf{flag}) \cdot y$.
3. Output $\langle d \rangle$ to all parties.

Fig. 23: Functionality for parties P_1, P_2, P_3 for obliviously selecting between one out of two values.

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

Parameter: Function description f.
Run: On receiving $\langle x \rangle$ from all parties, do:

1. Reconstruct x.
2. Compute $y := f(x)$.
3. Output $\langle y \rangle$ to all parties.

Fig. 24: Functionality for parties P_1, P_2, P_3 for securely computing a function f on additive shares.

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

Zero(n)**:** On receiving command and input n from all parties, generate random additive shares of $\boldsymbol{\alpha} = 0^n$, and output $\langle \boldsymbol{\alpha} \rangle$ to all parties.

Fig. 25: Functionality for parties P_1, P_2, P_3 to obtain random additive shares of 0.

D Further Benchmark Results

Table 4: Amortized runtimes for our protocol and for Duoram [VHG23] in ms per access for memory sizes $n = 2^8$ to $n = 2^{26}$ in the LAN and WAN setting with 16 threads for the preprocessing and 1 thread for the online phase. For each memory size and phase of the protocol, we marked the better runtimes with bold font.

$\log_2 n$	Time per Access in ms					
	LAN			WAN		
	Online	Prep.	Total	Online	Prep.	Total
Ramen (This Work)						
8	2.51	**0.41**	2.92	328.84	**10.12**	338.96
9	3.80	**0.68**	4.48	336.30	**7.57**	343.87
10	2.09	**0.57**	2.66	332.16	**5.42**	337.57
11	1.54	**0.57**	2.11	331.37	**3.98**	335.35
12	2.77	**1.31**	4.08	334.10	**3.41**	337.51
13	1.86	**1.24**	3.11	336.81	**2.85**	339.65
14	3.57	**1.37**	4.95	335.77	**2.92**	338.69
15	5.20	**2.00**	7.20	336.89	**3.78**	340.67
16	7.87	**2.59**	10.47	338.32	**5.09**	343.40
17	11.69	**3.78**	15.47	340.20	**5.87**	346.07
18	12.31	**5.27**	**17.58**	342.78	**7.31**	350.09
19	17.88	**8.12**	**26.00**	346.56	**9.63**	**356.18**
20	22.51	**12.18**	**34.69**	351.22	**13.15**	**364.37**
21	**26.81**	**17.69**	**44.50**	357.92	**19.74**	**377.65**
22	**29.70**	**25.66**	**55.36**	367.61	**28.35**	**395.97**
23	**33.25**	**37.18**	**70.43**	381.83	**41.10**	**422.93**
24	**38.85**	**53.84**	**92.69**	403.44	**59.64**	**463.07**
25	**49.98**	**77.87**	**127.85**	**436.62**	**85.99**	**522.61**
26	**64.97**	**112.54**	**177.51**	**477.26**	**124.30**	**601.56**
Three-Party Duoram [VHG23]						
8	**0.28**	1.60	**1.89**	**62.76**	138.54	**201.30**
9	**0.16**	2.12	**2.29**	**62.74**	147.40	**210.13**
10	**0.16**	2.09	**2.24**	**62.82**	159.45	**222.27**
11	**0.19**	2.16	**2.35**	**62.84**	171.60	**234.44**
12	**0.21**	1.75	**1.95**	**62.85**	183.56	**246.41**
13	**0.23**	2.56	**2.79**	**62.88**	196.18	**259.06**
14	**0.39**	3.88	**4.27**	**62.94**	209.51	**272.44**
15	**1.07**	2.61	**3.68**	**63.09**	224.40	**287.48**
16	**1.41**	4.03	**5.44**	**63.96**	238.19	**302.15**
17	**3.64**	7.64	**11.28**	**65.41**	253.89	**319.30**
18	**6.89**	13.82	20.71	**68.44**	270.19	**338.63**
19	**11.67**	31.24	42.92	**74.36**	304.04	378.40
20	**17.01**	56.84	73.85	**85.92**	350.37	436.29
21	31.44	114.70	146.14	**111.58**	464.25	575.83
22	63.67	223.57	287.24	**156.06**	543.40	699.46
23	124.57	444.66	569.22	**216.42**	784.49	1 000.91
24	238.60	888.97	1 127.57	**303.40**	1 235.31	1 538.71
25	464.33	1 774.95	2 239.29	521.04	2 129.11	2 650.15
26	922.67	3 550.03	4 472.70	978.71	3 907.21	4 885.92

Table 5: Amortized runtimes in ms per access for 16 threads and memory size $n = 2^{22}$ in different network settings having either a bandwidth limit or a certain enforced latency.

(a) With varying bandwidth.

Bandwidth in Mbit/s	Online	Prep.	Total
10	59.50	56.25	115.75
50	25.54	30.83	56.36
100	21.33	28.54	49.87
1 000	18.35	25.90	44.26
9 420	18.13	25.68	43.81

(b) With varying latency.

RTT in ms	Online	Prep.	Total
1.0	17.86	25.67	43.54
5.0	75.82	25.63	101.45
10.0	129.15	25.74	154.89
20.0	236.93	25.95	262.88
30.0	342.35	26.11	368.46

Table 6: Amortized runtimes in ms per access as well as speedup and efficiency of the parallelization for 1 to 16 threads in the LAN setting with memory size $n = 2^{22}$.

Threads	Online			Prep.			Total		
	Time	Speedup	Efficiency	Time	Speedup	Efficiency	Time	Speedup	Efficiency
1	29.65	1.00	1.00	215.49	1.00	1.00	245.14	1.00	1.00
2	27.11	1.09	0.55	108.76	1.98	0.99	135.87	1.80	0.90
3	26.39	1.12	0.37	84.63	2.55	0.85	111.02	2.21	0.74
4	22.71	1.31	0.33	57.00	3.78	0.95	79.71	3.08	0.77
5	21.85	1.36	0.27	52.22	4.13	0.83	74.07	3.31	0.66
6	21.14	1.40	0.23	44.25	4.87	0.81	65.39	3.75	0.62
7	19.99	1.48	0.21	36.27	5.94	0.85	56.26	4.36	0.62
8	19.45	1.52	0.19	32.90	6.55	0.82	52.34	4.68	0.59
9	19.13	1.55	0.17	32.43	6.65	0.74	51.56	4.75	0.53
10	18.94	1.57	0.16	32.07	6.72	0.67	51.01	4.81	0.48
11	18.71	1.59	0.14	30.24	7.13	0.65	48.94	5.01	0.46
12	18.62	1.59	0.13	28.92	7.45	0.62	47.54	5.16	0.43
13	18.43	1.61	0.12	28.12	7.66	0.59	46.55	5.27	0.41
14	18.22	1.63	0.12	27.36	7.88	0.56	45.58	5.38	0.38
15	17.98	1.65	0.11	26.34	8.18	0.55	44.32	5.53	0.37
16	17.86	1.66	0.10	25.67	8.39	0.52	43.54	5.63	0.35