Ramen: Souper Fast Three-Party Computation for RAM Programs
Full Version*

Lennart Braun1, Mahak Pancholi1, Rahul Rachuri2**, and Mark Simkin3
{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 $25\times$
 faster in the LAN and $8\times$ 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]. 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 $25 \times$ faster in the LAN and $8 \times$ 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>
<thead>
<tr>
<th></th>
<th>Rounds</th>
<th>Communication</th>
<th>Computation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jarecki et al. [JW18]</td>
<td>$O(\text{polylog}(n))$</td>
<td>$O(\text{polylog}(n))$</td>
<td>$O(\text{polylog}(n))$</td>
</tr>
<tr>
<td>Bunn et al. [BKKO20]</td>
<td>$O(1)$</td>
<td>$O(\sqrt{n})$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>Vadapalli et al. [VHG23]</td>
<td>$O(\text{polylog}(n)) + O(1)$</td>
<td>$O(\text{polylog}(n)) + O(1)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td><strong>Ramen (This Work)</strong></td>
<td>$O(1)$</td>
<td>$O(\sqrt{n} \cdot \text{polylog}(n))$</td>
<td>$O(\sqrt{n} \cdot \text{polylog}(n))$</td>
</tr>
</tbody>
</table>

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 $O(\sqrt{n} \cdot \text{polylog}(n))$.

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 $\pi$ over the domain $\{1, \ldots, n + \sqrt{n}\}$. The encoded memory is defined as $\tilde{M}[\pi(i)] := M[i]$ for all $1 \leq i \leq n + \sqrt{n}$. The client stores $\pi$ and the server stores an encryption of $\tilde{M}$ along with an initially empty array stash of size $\sqrt{n}$.

Whenever the client wants to perform an operation on $M[i]$, it downloads the full stash and checks whether $M[i]$ 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 $\tilde{M}$ that has not yet been read. If the desired entry $M[i]$ is not in the stash, then the client directly accesses $\tilde{M}[\pi(i)]$, performs the desired operation, and moves it to the stash. To ensure that the stash does not become too large, 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 $\pi$. 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 $M$ 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 $\tilde{M}$, 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 $f : X \rightarrow Y$ 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 $\lambda$ 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 $O(\lambda \log X)$ 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 $v$ of length $n$ and client $C$ wanting to privately retrieve $v[i]$. The client generates functions $f_1$ and $f_2$ for a point function $f$ that evaluates to 1 at $i$ and sends $f_2$ to $S_b$ for $b \in \{1, 2\}$. The servers independently compute $\sum_{i=1}^{n} f_b(i) \cdot v[i]$ and send the result back to $C$. The client can sum up the two received values to retrieve $v[i]$ 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 $F_{\text{DOPRF}}$ 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 $\text{PRF}$ and that all parties jointly hold an additive secret sharing of a value $x$, i.e., each $P_i$ holds
Additionally, the parties can jointly compute an additive secret sharing of the value $\text{PRF}(k, x)$. Whenever $k, x$ are given in plain, the value $\text{PRF}(k, x)$ can be computed locally without using $\mathcal{F}_{\text{DOPRF}}$. 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 $\text{PRF}$ 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 $M_1, M_2, M_3$, such that $M = M_1 + M_2 + M_3$. For $i \in \{1, 2, 3\}$, parties $P_{i-1}$ and $P_{i+1}$ will hold a PRF key $k_i$. Additionally, $P_{i-1}$ will hold a random mask vector $r_i$ 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 $(\text{PRF}(k_i, j), M_{i-1}[j] + r_i[\text{PRF}(k_i, j)])$, for $j \in \{1, \ldots, m\}$, sorted by $\text{PRF}(k_i, j)$. In this overview, we will largely ignore the role of the mask vectors $r_i$, 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 $\text{stash}$ among all parties. Additionally, $P_1$ and $P_3$ hold an initially empty ordered list $L_{\text{stash}}$ in plain and $P_2$ holds a PRF key $k_{\text{stash}}$. The list $L_{\text{stash}}$ will hold values $\text{PRF}(k_{\text{stash}}, \text{adr})$ for all addresses $\text{adr}$ that have already been read in the current epoch. The secret-shared stash will hold tuples $(\text{val}, \text{val}_{\text{old}})$, where the $j$-th entry corresponds to the memory location accessed at time $j$ in the current epoch. The value $\text{val}$ is the current value at that memory location, and $\text{val}_{\text{old}}$ is the value that was at that address at the beginning of the epoch. Looking ahead, storing both $\text{val}$ and $\text{val}_{\text{old}}$ 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 $\text{adr}$ from the CPU, the parties need to determine, whether the address is already in the stash. For this, the parties first use $\mathcal{F}_{\text{DOPRF}}$ to compute a secret sharing of $\text{PRF}(k_{\text{stash}}, \text{adr})$. Parties $P_2$ and $P_3$ agree on a mask $R$ and let $P_1$ reconstruct the value $\text{PRF}(k_{\text{stash}}, \text{adr}) + R$. Party $P_1$ then generates DPF keys $k_2, k_3$ for a point function that evaluates to 1 at $\text{PRF}(k_{\text{stash}}, \text{adr}) + R$ and to 0 everywhere else. $P_2$ and $P_3$ receive $k_2$ and $k_3$, respectively. Using their list $L_{\text{stash}}$ of read addresses, they add the mask $R$ to each entry and evaluate their DPF key shares on each entry. If $\text{PRF}(k_{\text{stash}}, \text{adr})$ 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 $v$ of zero entries with a single 1 at location $j$. If the address was not yet read, then all entries in $v$ will be 0. $P_2$ and $P_3$ non-interactively compute flag $\in \{0, 1\}$, which is sum of all entries in $v$, and loc, which is the inner product of $v$ 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 $\text{adr}$ was actually found in the stash.

Using a small amount of generic secure computation, the parties can use loc to either reveal $\text{PRF}(k_{\text{stash}}, \text{adr})$ or $\text{PRF}(k_{\text{stash}}, \text{adr})$, where $\text{adr}$ is an unused dummy address, to $P_2$ and $P_3$, who can add it to their list $L_{\text{stash}}$. 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 $\text{adr}'$, which is either the desired address $\text{adr}$ or an unused dummy address $\text{adr}$ in the main memory. Recall that each party $P_i$ holds key $k_{i+1}$, mask $r_{i+1}$, and a secret share of the memory that is permuted using key $k_i$ and masked using mask $r_i$. Using three invocations of $\mathcal{F}_{\text{DOPRF}}$, 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 $P_{1-1} = P_3$. 

4
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 

$$\left(\text{PRF}(k_i, \text{adr}), \text{PRF}(k_i, \overline{\text{adr}})\right)$$

along with a secret share of stash 

$$\left(\text{val}, \overline{\text{val}_\text{old}}\right), \left(\text{val}, \overline{\text{val}_\text{old}}\right).$$

$P_i$ creates keys $f_{i-1}$ and $f_{i+1}$ for a multi-point function that evaluates to her share of $\text{val} - \text{val}_\text{old}$ and $\overline{\text{val}} - \overline{\text{val}_\text{old}}$ at locations $\text{PRF}(k_i, \text{adr})$ and $\text{PRF}(k_i, \overline{\text{adr}})$, respectively. She sends $f_{i-1}$ and $f_{i+1}$ to $P_{i-1}$ and $P_{i+1}$, who evaluate their received functions at $\text{PRF}(k_{i,1}), \ldots, \text{PRF}(k_{i,n+\sqrt{n}})$. These evaluations create an additively secret-shared vector between $P_{i-1}$ and $P_{i+1}$. At positions $\text{adr}$ and $\overline{\text{adr}}$, 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, VH32].

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 $O(\text{polylog}(n))$ bandwidth overheads per access, but are generally difficult to adapt to the distributed ORAM setting. Their resulting protocols are rather complex, require $O(\log n)$ 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 $O(\sqrt{n} \cdot \text{polylog}(n))$ per party per access and each access requires $O(\log n)$ 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 $O(\sqrt{n})$, a computational overhead of $O(n)$, and require $O(1)$ 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 corruption, 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 $F_{DOPPR}$ functionality per access. In contrast to their protocol, our reinitialization is very simple and we only require a small constant number of $F_{DOPPR}$ 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 $O(\text{polylog}(n))$ rounds and communication in the preprocessing phase, and only $O(1)$ rounds and bandwidth overhead of $O(1)$ in the online phase. However, it suffers from a computational overhead of $O(n)$ per access per party. In contrast to their work, we also require $O(1)$ rounds per access and we have both a bandwidth and computational overhead of $O(\sqrt{n} \cdot \text{polylog}(n))$ 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 $\lambda$, and $\text{stat}$ is the statistical security parameter. Elements in memory $M$ are elements from a field $\mathbb{F}_p$, where the field size is at least $2^\lambda$. We use $\leftarrow$ 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 $\text{NS}$. We let $\text{st}_{\text{NS}}$ denote the current internal state of the program execution. The function $\text{NS}$ takes $\text{st}_{\text{NS}}$ as well as the last read value as input and outputs an instruction $I$ and the updated state $\text{st}_{\text{NS}}$. The instruction $I$ is a tuple $(\text{op}, \text{adr}, \text{val})$, where $\text{op}$ is either read or write and $\text{val}$ is the value to be written into the memory $M$ at location $\text{adr}$. The state $\text{st}_{\text{NS}}$ contains start, stop, or 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 $(\text{indx}, \text{val})$, where $\text{indx}$ denotes the index ranging from $1, \ldots, n$, and $\text{val}$ is the value at that index. We write $M[j].\text{val}$ and $M[j].\text{indx}$ to denote the value, respectively the index at location $j$ in $M$.

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

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

Pseudorandom Function (PRF) Let $\text{PRF}: K_{\text{PRF}} \times \mathcal{X}_{\text{PRF}} \rightarrow \mathcal{Y}_{\text{PRF}}$ be a pseudorandom function, where $K_{\text{PRF}}$ is the key space, $\mathcal{X}_{\text{PRF}}$ is the input, and $\mathcal{Y}_{\text{PRF}}$ is the output domains.
Pseudorandom Permutation (PRP) Let \( \text{PRP} : \mathcal{K}_{\text{PRP}} \times \mathcal{X}_{\text{PRP}} \rightarrow \mathcal{X}_{\text{PRP}} \) be a pseudorandom permutation, where \( \mathcal{K}_{\text{PRP}} \) is the key space and \( \mathcal{X}_{\text{PRP}} \) 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 \( \mathcal{F} = \{ f_{S,y} : [m] \rightarrow \mathbb{F} \} \) 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 for all \( x \in [m] \),

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

Let \( \lambda \) be the computational, and \( \text{stat} \) be the statistical parameter. A DPF scheme \( \Phi \) consists of the following two algorithms:

\[
(k_0, k_1) \leftarrow \Phi.\text{Gen}(1^\lambda, \text{stat}, f): \text{Given description } f \in \mathcal{F}, \text{ the algorithm returns two keys } k_0 \text{ and } k_1.
\]

\[
f_b(x) \leftarrow \Phi.\text{Eval}(k_0, x): \text{Given key } k_0 \text{ for party } b \in \{0, 1\} \text{ and input } x \in [m], \text{ return share } f_b(x) \text{ of } f(x).
\]

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

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

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

\[
\{ k_b \mid (k_0, k_1) \leftarrow \Phi.\text{Gen}(1^\lambda, \text{stat}, f_\lambda) \} \approx \{ k_b \leftarrow \text{Sim}_b(1^\lambda, \text{stat}, \mathbb{F}) \}.
\]

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 \( \mathcal{F}_{3,\text{OT}} \) (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 \( r \) of length \( \ell \). Then, the Access phase allows the index party (or chooser) \( C \) to repeatedly reveal entries in \( r \) at positions of its choice to the receiver party \( R \). The receiver \( R \) learns nothing about the index or the other values in \( r \).

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

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 $\pi$. $S$ defines each entry $j$ of the vector $r$ as $\text{PRF}(k_C, \pi(j)) + \text{PRF}(k_R, \pi(j))$. Then $S$ sends the keys $(k_C, k_{PRP})$ to $C$ and $(k_R, \pi)$ to $R$. This completes the Init phase. During Access, when $C$ receives an index $\text{indx}$ as input, it can locally compute the permuted index $\pi(\text{indx})$ and reveal $\text{PRF}(k_C, \pi(\text{indx}))$ to $R$. To allow $R$ to compute $\text{PRF}(k_R, \pi(\text{indx}))$, party $C$ also sends $\pi(\text{indx})$. Finally, $R$ can locally compute $r[\text{indx}] = \text{PRF}(k_C, \pi(\text{indx})) + \text{PRF}(k_R, \pi(\text{indx}))$. Since, only one out of two PRF keys are revealed to $C$ and $R$, they never learn the vector $r$. 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.

**Functionality $F_{3,\text{OT}}$**

**Parameters:** Parties $S, C, R$, vector length $\ell$, and output range $Y$.

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

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

**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 $F_f$ (Fig. 3), which allows us to execute a RAM program for a function $f(x, M)$, where $x$ is the input, and $M$ is the memory of size $n$. Functionality $F_f$ executes the RAM program by executing NS (the next step function) iteratively until $\text{st}_{NS}$ outputs stop. At this point, the final output of the function evaluation is stored in $\text{st}_{NS}$ 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 $F_f$ is to execute NS using any efficient MPC protocol.
of choice ($F_{\text{MPC}}$, Fig. 24), to obtain additive shares of instruction $I$, and use a specialized distributed ORAM functionality ($F_{\text{DORAM}}$, Fig. 13) for executing $I$ on memory $M$. The formal protocol description ($\Pi$, 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 $F_{\text{DORAM}}$.

### Functionality $F_i$

**Parameters:** A next-step function $\text{NS}$.

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

The functionality $F_{\text{DORAM}}$ (Fig. 13, Appendix A.4) receives additive shares of $M$ in the $\text{Init}$ phase, reconstructs and stores $M$ locally. Upon receiving additive shares of instruction $I$ in the $\text{Access}$ phase, $F_{\text{DORAM}}$ reconstructs and executes $I$ locally on $M$, then outputs additive shares of the result. Finally, $F_{\text{DORAM}}$ 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 $\text{Refresh}$ 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 $\text{read}$ operations on $M$. In each iteration of the $\text{Access}$ phase, parties receive additive shares of the instruction $I = (\text{op}, \text{adr}, \text{val})$ to be executed. To hide the type of operation, for both $\text{read}$ and $\text{write}$ operations, parties execute the following steps in sequence: 1. Check stash for the most up-to-date value for address $\text{adr}$. 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 $\Pi_{\text{DORAM}}$ (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 $F_{\text{Stash}}$, $F_{r,M}$, and $F_{w,M}$ which handle the main complexity of our construction, and whose instantiation is explained in the next subsections.

We now discuss the four steps of $\Pi_{\text{DORAM}}$ in more detail: Throughout the protocol execution we maintain the invariant that, if $\text{adr}$ 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 $\text{adr}$ by calling $F_{\text{Stash}}$ (Fig. 16, Appendix A.5). Then in Step 2, using $F_{r,M}$ (Fig. 18, Appendix A.6), $M$ is searched for some address, which is determined as follows: If $\text{adr}$ was previously read during the epoch, then parties read a fresh dummy address from $M$ in Step 2, otherwise they read $\text{adr}$. The dummy address for iteration $c$ is deterministically set to be $n + c$. In both steps, depending on whether or not $\text{adr}$ 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 $F_{\text{Stash}}$. This update is always made at location $c$, for iteration $c$. A second update must be made, which corresponds to the case when $\text{op} = \text{write}$. If $\text{adr}$ is a repeat address, the stash is updated at $\text{loc}$, where the most recent value of $\text{adr}$ is stored. If $\text{adr}$ is new, it is updated at $c$. In order to hide the type of operation, this is done even for $\text{op} = \text{read}$ but the update value is set to 0. For privacy in Steps 2 and 3, choices dependent on whether or not $\text{adr}$ was read previously and
whether \( op = \text{read} \) or \( \text{write} \) need to be made obliviously. This is done using a generic MPC instantiation of \( F_{\text{Select}} \) (Fig. 23, Appendix C).

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

### 4.2 Instantiating \( F_{\text{Stash}} \)

The functionality \( F_{\text{Stash}} \) (Fig. 16, Appendix A.6) for accessing the stash is instantiated by \( \Pi_{\text{Stash}} \), which consists of two parts.

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

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

### 4.3 Instantiating \( F_{\text{R-M}} \)

The functionality \( F_{\text{R-M}} \) (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 \( \text{Init} \), for \( i \in [3], P_{t-1} \) defines a vector \( r_i \) using an \( \text{Init} \) call to \( F_{3,\text{OT}} \). It also picks a PRF key \( k_i \) along with \( P_{t+1} \). Then \( P_{t-1} \) sends the masked and permuted memory share \( M'_i \) consisting of tuples \( (\text{PRF}(k_i, j), M_{t-1}[j] + r_i[\text{PRF}(k_i, j)]) \) for \( j \in [m] \), sorted by their first component, to \( P_t \).

In \( \text{Access} \), the value \( \text{adr}' = \text{PRF}(k_i, \text{adr}) \) is revealed to \( P_t \) via one oblivious PRF evaluation. \( P_t \) locally retrieves the value at \( M[\text{adr}'] \) which is equal to \( M_{t-1}[\text{adr}].\text{val} + r_i[\text{adr}'] \). To form an additive sharing between \( P_t \) and \( P_{t+1} \), we must reveal the mask at the index \( \text{adr}' \) to \( P_{t+1} \). At this point, notice that, \( P_{t-1} \) holds the vector \( r_i \), \( P_t \) holds the index \( \text{adr}' \), and \( P_{t+1} \) should learn the mask \( r_i[\text{adr}'] \). A call to \( F_{3,\text{OT}}(\text{Access}) \) allows us to do exactly this. One execution of the above procedure results in additive shares of \( (M[\text{adr}].\text{val}) \) held by \( P_t, P_{t+1} \). Repeating this procedure thrice for each party’s share of \( M \), gives us additive shares of \( M[\text{adr}].\text{val} \).

Observe that since the shuffling of \( (M) \) mentioned above is performed by sorting PRF evaluations of the addresses, \( M'_i \) consists of tuples of the form \( (\text{index}', \text{val}') \), where \( \text{index}' \in \mathcal{Y}_{\text{PRF}} \). For correctness, we require \( Y_{\text{PRF}} \) to be big enough in order to avoid any collisions, which means that it can be much bigger than \( m \). Naively implementing \( r_i \) would then require making it as big as the size of \( Y_{\text{PRF}} \). However, this can be prevented by leveraging the fact that the sorted list \( \{\text{PRF}(k_i, \text{index})\}_1^{m} \) is public, as \( k_i \) is known to \( P_{t-1}, P_{t+1} \) and the list is revealed to \( P_t \), and that there are no collisions in \( Y_{\text{PRF}} \) evaluations. Let the sorted list be denoted by \( L_i \). Now the parties define an inverse map denoted by \( L_i^{-1} \) and defined...
We use the Legendre PRF to instantiate the Appendix A.

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

Read the stashed value $\text{stash[loc]:val}$:
8. Convert $(\text{stash.val})$ to $[\text{stash}.\text{val}]$: For $i \in \{1, 2, 3\}$, and for $j \in [[\text{stash}]]$, each $P_i$ sends $(\text{stash}[j].\text{val})_i$ to $P_{i+1}$.
9. Each party initializes $(\text{val}_i) := 0$.
10. For $i \in \{1, 2, 3\}$, do:
   (a) $P_{i-1}$ samples $r_{i-1,i}, r_{i+1,i} \leftarrow \mathcal{F}_p$ 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 $(\text{loc}) + r_{i-1,i}$, and $P_{i+1}$ sends $(\text{loc}) + r_{i+1,i}$, to $P_i$. $P_i$ reconstructs $(\text{loc}) := (\text{loc}) + r_i$.
   (c) $P_i$ defines $f_i : \mathcal{Y} \to \{0, 1\}$ such that $f_i(j) := 1$ if $j = \text{loc}_i$. It generates DPF keys $(k_{i-1,i}, k_{i+1,i}) \leftarrow \Phi_2.\text{Gen}(1^\lambda, \text{stat}, f_i)$, $P_i$ sends $k_{i-1,i}, k_{i+1,i}$ to $P_{i-1}$ and $P_{i+1}$, respectively.

5.4 Instantiating $\mathcal{F}_\text{DOPRF}$

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

$$f_{\text{PRF}}(k, x) = L_p(k + x) \quad \text{with} \quad L_p(a) = \frac{1}{2} \left( \left( \frac{a}{p} \right) + 1 \right) \mod p.$$
Write in Stash: Each party has input \((\text{val}_a), (\text{adr}_a), c, (\text{stash}), (\text{st}_\text{stash}), (\ell)\).

1. \(P_2, P_3\) call \(\text{PRF}_{\text{DOPRF}}\) with \((\text{Eval}, \text{unmasked}, (\text{adr}_a))\), and \(P_1\) calls it with \((\text{Eval}, \text{unmasked}, (\text{adr}), k_{\text{stash}})\) acting as \(K\), \(P_2\) acting as \(R_2\), receives \(\text{PRF}(k_{\text{stash}}, \text{adr})\).
2. \(P_2\) sends \(\text{PRF}(k_{\text{stash}}, \text{adr})\) to \(P_3\) and they both append list \(L_{\text{stash}} := L_{\text{stash}} || \text{PRF}(k_{\text{stash}}, \text{adr})\).
3. Each party \(P_i\) locally computes the \(p_i\) idea is to mask \(f\) protocol in \([\text{GRR}]\). Compared to the latter, we use it to obtain \(\text{PRF}\).

4. Parties parse \((\text{stash})\) as \(((\text{flag}), (\text{loc}), (\text{val}_a))\), and \((I)\) as \(((\text{op}), (\text{adr}), (\text{val}))\).
5. Parties call \((\text{val}_a) \leftarrow F_{\text{select}}(\text{flag}), (\text{val}_a), (\text{val}_a))\) and \((\text{val}_a) \leftarrow F_{\text{select}}(\text{op}), (\text{val}_a), (\text{val}_a))\).
6. Each \(P_i\) computes \(\langle \Delta \rangle := \langle \text{val}_\text{stash} \rangle - \langle \text{val}_\text{old} \rangle\) and initializes a vector \(\delta_i\) of length \(|\text{stash}|\) at \((0, \ldots, 0)\).
7. For \(i \in \{1, 2, 3\}\), do:
   - \(P_{i-1}\) samples \(\rho_{i-1, i}, \rho_{i, i+1} \leftarrow \mathbb{Z}_p^2\), and sends them to \(P_{i+1}\). Both \(P_{i-1}\) and \(P_{i+1}\) compute \(\rho_i := \rho_{i-1, i} + \rho_{i, i+1}\).
   - \(P_{i-1}\) sends \((\text{loc}) + \rho_{i-1, i}, P_{i+1}\) sends \((\text{loc}) + \rho_{i, i+1}\) to \(P_i\).
   - \(P_i\) reconstructs \((\text{loc}) := \text{loc} + \rho_i\).
   - \(P_i\) defines \(f_i : \mathbb{F}_p \rightarrow \mathbb{F}_p\), such that \(f_i(j) := \langle \Delta \rangle\) at \(j = \text{loc}\). It generates \(\text{DFP}\) keys \((k_{i, i-1}, k_{i, i+1}) \leftarrow \Phi_i \cdot \text{Gen}(1^\lambda, \text{stat}, f_i)\). \(P_i\) sends \(k_{i, i-1}, k_{i, i+1}\) to \(P_{i-1}\) and \(P_{i+1}\), respectively.
   - For \(j \in |\text{stash}|\), \(P_{i-1}\) updates \(\delta_{i-1}[j] := \delta_{i-1}[j] + \Phi_i \cdot \text{Eval}(k_{i, i-1}, j + \rho_i)\), and \(P_{i+1}\) updates \(\delta_{i+1}[j] := \delta_{i+1}[j] + \Phi_i \cdot \text{Eval}(k_{i, i+1}, j + \rho_i)\).
8. All parties call \((\text{O}_1) \leftarrow F_{\text{DOPRF}}(\text{stash})\).
9. For \(j \in |\text{stash}|\), each party \(P_i\) updates \(\langle \text{stash}[j], \text{val} \rangle := \langle \text{stash}[j], \text{val} \rangle + \delta_i[j] + \langle \text{O}_1[j] \rangle\).
10. All parties call \((\text{O}_2) \leftarrow F_{\text{DOPRF}}(\text{stash})\) and \((\text{O}_3) \leftarrow F_{\text{DOPRF}}(\text{stash})\).
11. For \(j \in |\text{stash}|\), each party \(P_i\) updates \(\langle \text{stash}[j], \text{adr} \rangle := \langle \text{stash}[j], \text{adr} \rangle + \langle \text{O}_2[j] \rangle\) and \(\langle \text{stash}[j], \text{val}_{old} \rangle := \langle \text{stash}[j], \text{val}_{old} \rangle + \langle \text{O}_3[j] \rangle\).
12. Parties output \((\text{stash})\). In addition, \(P_2, P_3\) output \(L_{\text{stash}}\).

*steps 10 and 11 can be avoided with a minor change in the \(F_{\text{Stash}}\) 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 \(\ell\) bit output, we select \(\ell\) independent keys and replicate the computation for a single bit output \(\ell\) times. More specifically, for an \(\ell\) bit output, we pick a vector of keys of size \(\ell\), denoted as \(k\), and define \(\text{PRF}(k, x)\) as:

\[
\text{PRF}(k, x) = \left(\sum_{i=0}^{\ell-1} 2^i \cdot L_p(k_i + x)\right) \mod{p}.
\]

We optimize the implementation of our protocol by using two different types of \(F_{\text{DOPRF}}\) evaluations, which we call the unmasked and the masked type, respectively. In both types, one party \(K\) holds the \(\text{PRF}\) key \(k\). In the unmasked type, some other receiver party \(R_2\) learns, in clear, the \(\text{PRF}\) 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 \((R_1, R_2)\) 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 \(\text{PRF}\) evaluations in the write phase of \(\Pi_{\text{stash}}\), and for all \(\text{PRF}\) evaluations in \(\Pi_{\text{stash}}\). The masked type is used to obtain \(\text{PRF}\) evaluations in the read phase of \(\Pi_{\text{stash}}\).

The functionality \(F_{\text{DOPRF}}\) 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 \(F_{\text{DOPRF}}\).

For completeness, we present the masked version \(\Pi_{\text{DOPRF}}\) here in Fig. 9, while the unmasked version \(\Pi_{\text{DOPRF}}\) is deferred to Appendix B in Fig. 22. Given \(\Pi_{\text{DOPRF}}\), 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 \(\text{PRF}\) evaluation. For the masked version, the idea is similar. We reveal either
Protocol $P_{\alpha,M}$

**Parameters:** PRF $PRF: K \times [m] \to \mathbb{Y}$.

**Init:** Each party $P_i$ inputs its share $(M)$ which is interpreted as a list of tuples $(\text{indx}, \langle \text{val}\rangle)$, where $\text{indx} \in [m]$. For $i \in \{1, 2, 3\}$.

1. All parties call $F_{\text{DPRF}}$ 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 $L_i[j] := PRF(k_i, j)$, for $j \in [m]$ and sort it.
4. $P_{i-1}$ calls $F_{\text{3-OT}}$ with (Init). $P_i, P_{i+1}$ call $F_{\text{3-OT}}$ with Init. $P_{i-1}$ receives mask vector as $r_i$.
5. $P_{i-1}$ computes:
   - (a) For $j \in [m]$, set $M'_i[j], \text{indx} := PRF(k_i, j)$, and $M'_i[j], \text{val} := (M[j].\text{val}) + r_i[L_i^{-1}(PRF(k_i, j))]$.
   - (b) Sort tuples in $M'_i$ lexicographically with respect to $M'_i, \text{ indx}$.
   - (c) Send $M'_i$ to $P_i$.
6. $P_i$ receives $M'_i$ from $P_{i-1}$.
7. $P_i$ locally computes list $L_i[j] := M'_i[j].\text{ indx}$, for $j \in [m]$.
8. $P_i$ initializes empty list $\text{adr}_i^{\text{rad}}$.
9. Output: $P_i$ outputs $k_{i-1}, k_{i+1}, L_i$.

**Access:** Each party $P_i$ inputs additive share $(\text{adr}_i)$, the set $\text{adr}_i^{\text{rad}}$ and keys $k_{i-1}, k_{i+1}$.

1. Each party $P_i$ initializes $(\nu) := 0$.
2. For $i \in \{1, 2, 3\}$, do:
   - (a) $P_{i-1}$ calls $F_{\text{DPRF}}$ with inputs $(\text{Eval}, \text{unmasked}, (\text{adr}_i), k_i)$, and $P_i, P_{i+1}$ call with input $(\text{Eval}, \text{unmasked}, (\text{adr}_i))$ with $P_i$ acting as $R_2$. $P_i$ receives $\text{adr}_i := PRF(k_i, \text{adr}_i)$.
   - (b) $P_i$ updates $\text{adr}_i^{\text{rad}} := \text{adr}_i^{\text{rad}}[\text{adr}_i]$.
   - (c) $P_i$ computes $j' := L_i^{-1}(\text{adr}_i)$.
   - (d) $P_i$ sets $(x) := M'_i[j'], \text{val}$ and $P_{i-1}$ sets $(x) := 0$.
   - (e) $P_i$ calls $F_{\text{3-OT}}$ with $(\text{Access}, j')$, and $P_{i-1}, P_{i+1}$ call with Access. $P_{i+1}$ receives $r_i[j']$, and sets $(x) := -r_i[j']$.
   - (f) Each party computes: $(\nu) := (\nu) + (x)$.
3. Output: Each party $P_i$ outputs $(\nu, \text{adr}_i^{\text{rad}})$.

---

**Protocol $P_{\alpha,M}$**

**Parameters:** $\Phi = (\text{Gen}, \text{Eval})$ is a multi-point DP scheme for functions $[m] \to \mathbb{F}_p$, 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 $(M), L_i, \text{ and (stash)}$. $P_i$ also inputs auxiliary information $\text{adr}_i^{\text{rad}}$, and two PRF keys $(k_{i-1}, k_{i+1})$. $M$ is interpreted as set of tuples $(\text{indx}, \text{val})$.

1. For $i \in \{1, 2, 3\}$, do:
   - (a) $P_{i-1}, P_{i+1}$ locally compute a list $L_i[j] := PRF(k_i, j)$, 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 $f_i: [m] \to \mathbb{F}_p$ such that, for $j \in [\sqrt{n}]$, $f_i((L_i^{-1}(\text{adr}_i^{\text{rad}})[j])) := (\text{stash}[j].\text{val}) - (\text{stash}[j].\text{val}_\text{rad})$. It generates multi-point DP keys $(mk_{i-1}, mk_{i+1}) \leftarrow \Phi.\text{Gen}(1^3, \text{stat}, f_i)$ and sends them to $P_{i-1}$ and $P_{i+1}$ respectively.
   - (d) For $t \in \{1, 2, 3\} \setminus \{i\}$: $P_i$ locally updates $(M)_i$ as: For $j \in [n]$, $(M[j].\text{val}) := (M[j].\text{val}) + \Phi.\text{Eval}(mk_{i,t}, L_i^{-1}(PRF(k_{i,t}))$.
2. All parties call $F_{\text{Zero}}(n)$ to receive $(0)$.
3. Parties locally compute $(M) := (M) + (0)$.

---

$s^2(k + x)$ or $\alpha \cdot s^2(k + x)$ depending on whether $r = 0$ or 1, where $\alpha$ 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 $\Pi^{\text{m}}_{\text{DORAM}}$, and Steps 1–3 in $\Pi^{\text{m}}_{\text{DOPRF}}$ (Fig. 22). Moreover, the PRG seeds and the value $\alpha$ can be set up once and for all. $\alpha$ can be reused in all $\Pi^{\text{m}}_{\text{DOPRF}}$ instantiations as it is always masked by $s$ which is sampled freshly in each instantiation.

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

**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 $2^{-\text{stat}}$, where $\text{stat}$ is the statistical security parameter, we set the parameter $\ell := 2 \log_2(m) + \text{stat}$. When we use the DOPRF in the stash protocol, where the domain size is $\sqrt{n}$, we set $\ell := 2 \log_2(\sqrt{n}) + \text{stat}$, accordingly.

\[\begin{array}{|c|}
\hline
\text{Parameters: Output length } \ell, \text{ a PRF: } K \times \mathcal{X} \rightarrow \{0,1\}^\ell. \\
\text{KeyGen: On receiving (KeyGen, } P_t) \text{ from all parties, treat } P_t \text{ as } K. \text{ Randomly sample } k \leftarrow K. \text{ Output } k \text{ to } K. \\
\text{Eval: On receiving (Eval, mode, } (x)) \text{ from all parties and receive } k \text{ from } K. \text{ Do:} \\
1. \text{ Reconstruct } x. \text{ Compute } y := \text{PRF}(k, x). \\
2. \text{ Sample } r \leftarrow \{0,1\}^\ell. \\
3. \text{ If mode = masked, output } r \text{ to } R_1, R_2, \text{ and } y \oplus r \text{ to } K. \\
4. \text{ Else, mode = unmasked output } y \text{ to } R_2. \\
\hline
\end{array}\]

Fig. 8: Functionality for a distributed PRF evaluation between parties $R_1, R_2, K$ where $K$ holds the key. In $\text{mode = unmasked}$ it allows $R_2$ to obtain PRF evaluation in clear, and in $\text{mode = masked}$ 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 $\Pi_t$. For the sake of analysis, assume that the number of instructions required to execute $\Pi_t$ is $t \cdot \sqrt{n}$, i.e., we have $t$ epochs. $\Pi_t$ makes one call to $\mathcal{F}_{\text{DORAM}}(\text{Init})$ and $\mathcal{F}_{\text{DORAM}}(\text{GetM})$ once per execution. Per instruction, $\Pi_t$ makes one call to $\mathcal{F}_{\text{MPC}}$ and $\mathcal{F}_{\text{DORAM}}(\text{Access})$. When instantiated, the only prohibitive component here are the calls to $\mathcal{F}_{\text{DORAM}}$. We give the per instruction, per party cost (in terms of field elements) for $\mathcal{F}_{\text{DORAM}}(\text{Access})$ and sub-protocols required for instantiating it. We give per execution cost for instantiating $\mathcal{F}_{\text{DORAM}}(\text{Init})$ and $\mathcal{F}_{\text{DORAM}}(\text{GetM})$. The costs are summarized in Table 2.

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

14
To benchmark the performance of our protocols, we implemented them using the Rust programming language. Our implementation is available as open source software at [source link].

Init: Parties sample pairwise PRG seeds; K samples $k_{a,1} \leftarrow \{0, 1\}^l$, $R_1$ samples $k_{1,2} \leftarrow \{0, 1\}^l$, and $R_2$ samples $k_{2,2} \leftarrow \{0, 1\}^l$, and sends it to $R_1$, $R_2$, and K, respectively. Let $a$ be a fixed quadratic non-residue modulo $p$ known to $R_1$ and $R_2$.

Eval: Each party has input $x$, and pairwise PRG seeds. K in addition has key $k$.

1. $K$ and $R_1$ compute $m, a, c_1, e \leftarrow PRG(k_{a,1})$, where $m \in \mathbb{F}_p$, $a, a_1, e \in \mathbb{F}_p^*$.
2. $R_1, R_2$ compute $s \leftarrow PRG(k_{1,2})$, where $s \in \mathbb{F}_p^*$.
3. $R_1$ samples $r_1 \leftarrow \{0, 1\}^l$, and sends it to $R_2$. Similarly, $R_2$ samples $r_2 \leftarrow \{0, 1\}^l$, and sends it to $R_1$.

   Both $R_1, R_2$ set $r := r_1 \oplus r_2$.
5. $R_1$ defines $b := t - e$, and $c_3 := a \circ b - c_1$. Send $c_3$ 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 $d := k + y_1 - a$ to $R_2$.
8. $R_2$ computes $w := t \cdot (y_2 + y_3)$, and $z_3 := d \circ t + c_3 + w$. It sends $z_3$ to $K$.
9. $K$ computes $z_3 := e \circ (k + y_1)c_1 - d \circ e$, and $z := z_1 + z_3$.
10. $K$ computes, for $j \in [1, l]$, $o[j] := \frac{1}{2}((z_3^2) + 1) \mod p$.
11. $K$ outputs $o, R_1, R_2$ output $r$.

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

Cost for one $\Pi_t$ execution. For $t \cdot \sqrt{n}$ instructions, $\Pi_t$ executes $\Pi_{DORAM}$(Init) once, $\Pi_{DORAM}$(Access) $t \cdot \sqrt{n}$ times, and $\Pi_{DORAM}$(GetM) once. This amounts to $O(\sqrt{n}\log(m))$ 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 [source link].

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 \( \Theta(1) \) rounds of communication.

<table>
<thead>
<tr>
<th>Primitive</th>
<th>Computation</th>
<th>Communication</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \Pi_{DOPRF} )</td>
<td>( \Theta(\log(m)) )</td>
<td>( \Theta(\log(m)) )</td>
</tr>
<tr>
<td>( \Pi_{3,OT}(\text{Init}) )</td>
<td>( \Theta(m) )</td>
<td>( \Theta(1) )</td>
</tr>
<tr>
<td>( \Pi_{3,OT}(\text{Access}) )</td>
<td>( \Theta(1) )</td>
<td>( \Theta(1) )</td>
</tr>
<tr>
<td>( \Pi_{\text{RAM}}(\text{Read}) )</td>
<td>( \Theta(\sqrt{n}) )</td>
<td>( \Theta(\sqrt{n}) )</td>
</tr>
<tr>
<td>( \Pi_{\text{RAM}}(\text{Write}) )</td>
<td>( \Theta(\sqrt{n}) )</td>
<td>( \Theta(\sqrt{n} \log(p)) )</td>
</tr>
<tr>
<td>( \Pi_{\text{RAM}}(\text{Init}) )</td>
<td>( \Theta(m \log(m)) )</td>
<td>( \Theta(m \log(m)) )</td>
</tr>
<tr>
<td>( \Pi_{\text{RAM}}(\text{Access}) )</td>
<td>( \Theta(\log(m)) )</td>
<td>( \Theta(\log(m)) )</td>
</tr>
<tr>
<td>( \Pi_{\text{M}} )</td>
<td>( \Theta(\sqrt{n}) )</td>
<td>( \Theta(\sqrt{n}) )</td>
</tr>
<tr>
<td>( \Pi_{\text{DORAM}}(\text{Init}) )</td>
<td>( \Theta(\log(m)/t) )</td>
<td>( \Theta(\log(m)/t) )</td>
</tr>
<tr>
<td>( \Pi_{\text{DORAM}}(\text{Access}) )</td>
<td>( \Theta(\sqrt{n} \log(m)) )</td>
<td>( \Theta(\sqrt{n} \log(m)) )</td>
</tr>
<tr>
<td>( \Pi_{\text{DORAM}}(\text{GetM}) )</td>
<td>( \Theta(\log(m)/t) )</td>
<td>( \Theta(\log(m)/t) )</td>
</tr>
</tbody>
</table>

key-AES-based hash function \cite{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 \texttt{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 \cite{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 \( n = 2^{26} \) 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](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.

<table>
<thead>
<tr>
<th>log$_2 n$</th>
<th>Time per Access in ms</th>
<th>Communication in KiB</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>LAN Online</td>
<td>Prep.</td>
</tr>
<tr>
<td>8</td>
<td>2.83</td>
<td>0.43</td>
</tr>
<tr>
<td>9</td>
<td>5.70</td>
<td>0.77</td>
</tr>
<tr>
<td>10</td>
<td>3.21</td>
<td>0.82</td>
</tr>
<tr>
<td>11</td>
<td>5.80</td>
<td>1.13</td>
</tr>
<tr>
<td>12</td>
<td>4.00</td>
<td>1.10</td>
</tr>
<tr>
<td>13</td>
<td>2.28</td>
<td>1.32</td>
</tr>
<tr>
<td>14</td>
<td>5.70</td>
<td>1.76</td>
</tr>
<tr>
<td>15</td>
<td>4.30</td>
<td>1.98</td>
</tr>
<tr>
<td>16</td>
<td>4.48</td>
<td>2.59</td>
</tr>
<tr>
<td>18</td>
<td>11.11</td>
<td>5.29</td>
</tr>
<tr>
<td>19</td>
<td>8.75</td>
<td>7.67</td>
</tr>
<tr>
<td>20</td>
<td>8.86</td>
<td>12.15</td>
</tr>
<tr>
<td>21</td>
<td>15.74</td>
<td>17.73</td>
</tr>
<tr>
<td>22</td>
<td>17.86</td>
<td>25.67</td>
</tr>
<tr>
<td>23</td>
<td>21.21</td>
<td>37.16</td>
</tr>
<tr>
<td>24</td>
<td>25.65</td>
<td>53.82</td>
</tr>
<tr>
<td>25</td>
<td>31.28</td>
<td>77.86</td>
</tr>
<tr>
<td>26</td>
<td>37.56</td>
<td>112.53</td>
</tr>
</tbody>
</table>

The 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 $O(\sqrt{n})$ vs. Duoram’s $O(n)$ – cost kicks in: While our runtimes increase relatively slowly, those of Duoram deteriorate rapidly, so that our protocol is $14 \times$ 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 $2 \times$ 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 $25 \times$ in the LAN setting, and $8 \times$ 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
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.\footnote{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.}

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.
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 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 $A$ and an environment $Z$. The environment $Z$ can write inputs to all parties, read outputs of all parties, and additionally interact with $A$ throughout execution. After $A$ 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 $Z$ wants to write input $x$ for a corrupted party, it first interacts with $A$ who might change it to $x'$, after which $Z$ writes $x'$ on the input tape of the corrupt party. Let $\text{Real}_{Π, A, Z}$ denote the output of the distinguisher $Z$ when interacting in the real-world experiment. The ideal-world experiment is defined with respect to an ideal functionality $F$, ideal adversary $S$, the environment $Z$ and a set of dummy parties $P_1, P_2, P_3$. Similar to the previous case, $Z$ writes to the input tapes, and read from the output tapes of all dummy parties. As in the real-world experiment, here too $Z$ interacts with the adversary $S$ throughout the execution. When $S$ is activated for the first time, it activates an instance of $A$ internally, and relays communication between $A$ and $Z$. This is useful when $Z$ wants to interact with $A$ to decide corrupt party’s inputs; it does so via $S$. This means that $S$ in the passive case is aware of the inputs of the corrupted party even if they might be modified by $A$. Finally, when each dummy party receives its input, it sends it to $Z$ to receive outputs. The corrupt party’s output is received by $S$ who writes it on its output tape to be read by $Z$. Let $\text{Ideal}_{F, S, Z}$ denote the output of the distinguisher $Z$ when interacting in the ideal-world experiment.

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

$$\{\text{Ideal}_{F, S, Z}(x, \lambda, z)\}_{x, \lambda, z} \equiv \{\text{Real}_{Π, A, Z}(x, \lambda, z)\}_{x, \lambda, z}$$

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

Each proof of security will follow the same pattern. First the simulator $S$ starts by invoking an instance of $A$ and runs a simulated interaction of $A$ with $Z$ (by simply relaying messages between them). $S$ receives the adversary’s inputs from $Z$, and it writes them on $A$’s input tape. Similarly, every output value written by $A$ is copied by $S$ on its output tape to be read by $Z$. The simulator $S$ calls $F$ on the adversary’s input to receive the function’s output. It then continues to simulate $A$’s view using this output. Since this is a common strategy for all simulators, we skip the explicit invocation of $A$ 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 $Π_{3\cdot OT}$

**Theorem 1.** Protocol $Π_{3\cdot OT}$ UC-securely implements functionality $F_{3\cdot OT}$ in presence of one passive corruption.

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

**$S$ corrupt:** From $Z$, $S$ receives command $\text{Init}$. It receives keys $k_C, k_R, k_{PRP}$ from $S$ on behalf of $C, R$, computes vector $r$ just as $S$ would in the protocol $Π_{3\cdot OT}$, and calls $F_{3\cdot OT}$ on $r$. It receives no output in return.

**$C$ corrupt:** On receiving command $(\text{Access}, \text{indx})$ from $Z$, call $F_{3\cdot OT}$. $S$ receives no output in return. Simulate the view as follows. It simulates the transcript in the $\text{Init}$ 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 $\text{Access}$ phase, $S$ simulates the transcript by accepting messages sent by $C.
R corrupt: On receiving command (Init) from Z, S calls \( F_{3\cdot OT} \). TO simulate the view, it samples and returns a random key \( k_R \leftarrow \mathcal{K}_{PRF} \). This is exactly as in the real world and hence is indistinguishable. For each iteration of Access, on receiving command (Access) from Z, the simulator works as follows:

1. Since R has no input, S simply calls \( F_{3\cdot OT} \) on input Access and receives \( x \) which is the output of the current iteration.
2. Pick a random index \( indx' \in [\ell] \) that has not been picked in any of the previous iterations.
3. Set \( a := x - \text{PRF}(k_R, indx') \).
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.

- Hyb0: The real world view obtained by executing steps in \( II_{3\cdot OT} \).
- Hyb1: Same as Hyb0 except that S receives \( x \) from \( F_{3\cdot OT} \) and sets \( a := x - \text{PRF}(k_R, indx) \).
- Hyb2: Same as Hyb1 except that \( indx \) is replaced by a uniformly random index \( indx' \in [\ell] \) 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 Hyb2 is exactly how S behaves for a corrupt R. \( \square \)

### A.3 Security Proof of \( II_t \)

#### Protocol \( II_t \)

<table>
<thead>
<tr>
<th>Parameters: A next-step function NS. Each party inputs shares ( \langle x \rangle, \langle M \rangle ), and gets output ( \langle \text{st}_{NS} \rangle, \langle M \rangle, \text{rnd} ).</th>
</tr>
</thead>
<tbody>
<tr>
<td>Run: 1. All parties initialize ( \langle \text{st}<em>{NS} \rangle := \langle \text{start}, \langle x \rangle \rangle, \langle d \rangle := 0 ) and ( \text{rnd} := 0 ). Call ( F</em>{\text{DORAM}} ) with (Init, ( \langle M \rangle )).</td>
</tr>
<tr>
<td>2. Do:</td>
</tr>
<tr>
<td>(a) The parties call ( F_{\text{MPC}} ) with inputs ( \langle \text{st}<em>{NS} \rangle, \langle d \rangle ) to securely compute ( \langle \text{st}</em>{NS} \rangle, \langle l \rangle \leftarrow NS(\langle \text{st}_{NS} \rangle, \langle d \rangle) ).</td>
</tr>
<tr>
<td>(b) Parse ( \langle \text{st}_{NS} \rangle ) as ( \langle z \rangle ).</td>
</tr>
<tr>
<td>(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 = \text{stop} ), then break. Else if ( z = \text{continue} ), continue.</td>
</tr>
<tr>
<td>(d) Call ( F_{\text{DORAM}} ) with input (Access, ( \langle l \rangle )) to receive output ( \langle d \rangle ).</td>
</tr>
<tr>
<td>(e) Update ( \text{rnd} := \text{rnd} + 1 ).</td>
</tr>
<tr>
<td>3. Call ( F_{\text{DORAM}} ) with input (GetM) to receive output ( \langle M \rangle ).</td>
</tr>
<tr>
<td>4. Each party locally outputs ( \langle \text{st}_{NS} \rangle, \langle M \rangle, \text{rnd} ).</td>
</tr>
</tbody>
</table>

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

#### Theorem 2.
Protocol \( II_t \) UC-securely instantiates functionality \( F_t \) in presence of one passive corruption in the \( (F_{\text{DORAM}}, F_{\text{MPC}}, F_{\text{Check}}) \)-hybrid model.

**Proof.** The correctness of \( II_t \) is clear from inspection. We argue security against a corrupt \( P_1 \). The simulation strategy for other corruption cases is analogous.

- Hyb0: Same as the real world execution except that, when \( F_{\text{DORAM}} \) is called with input GetM, reply with \( \langle M_{out} \rangle \). This is indistinguishable because correctness.

- Hyb1: When \( \text{crnd} = \text{rnd} \), \( S_f \) returns random shares for \( \langle l \rangle \), and \( \langle \text{st}_{II_{out}} \rangle \), where \( \langle \text{st}_{II_{out}} \rangle \) is received from the functionality. This is indistinguishable because security of additive sharing, and correctness of \( II_t \).

- Hyb2: When \( \text{crnd} \neq \text{rnd} \), \( S_f \) returns random shares for \( \langle st_{II} \rangle \) and \( \langle l \rangle \) rather than the correctly computed shares. Moreover, it returns calls to \( F_{\text{DORAM}}(\text{Access}) \) with a random \( \langle d \rangle \). This is indistinguishable from the previous hybrid because of the security of the additive shares.

Notice that Hyb2 is the view generated by \( S_f \). \( \square \)
Simulator $S_1$

1. On receiving inputs of the corrupt party $(M), (x)$, call $F_1$ to receive outputs outputs $(⟨\text{st}_{\text{NS, out}}⟩, ⟨M_{\text{out}}⟩, \text{rnd})$. Simulate the view as follows.
2. Receive call to $F_{\text{DORAM}}$(init) from $P_1$. Return nothing in response. Initialize current counter $\text{crnd} := 0$.
3. For $\text{crnd} < \text{rnd}$, do:
   (a) Receive call to $F_{\text{MPC}}$ from $P_1$. Sample and return random $(⟨\text{st}_{\text{NS}}⟩, (l))$ (of appropriate length).
   (b) Receive $(z)$ from $P_1$. Sample $(z)_2, (z)_3$ such that $z = \text{continue}$. Send $(z)_2, (z)_3$ to $P_1$ on the behalf of $P_2, P_3$.
   (c) Receive call to $F_{\text{DORAM}}$(Access) from $P_1$. Return $(d) ← F_p$.
   (d) Update $\text{crnd} := \text{crnd} + 1$.
4. If $\text{crnd} = \text{rnd}$, do:
   (a) Receive call to $F_{\text{MPC}}$ from $P_1$. Sample $(l)$ uniformly at random of appropriate length and return $(⟨\text{st}_{\text{NS, out}}⟩, (l))$.
   (b) Receive call to $F_{\text{Check}}$ from $P_1$. Return stop to $P_1$.
5. Receive call to $F_{\text{DORAM}}$(GetM) from $P_1$ and return $(M_{\text{out}})$.

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

A.4 Security Proof of $\Pi_{\text{DORAM}}$

$F_{\text{DORAM}}$

Init: On receiving (Init, $(M)$) from all parties, reconstruct and store $M$ locally.
Access: On receiving (Access, $(l)$) from all parties, reconstruct instruction $l := (\text{op}, \text{adr}, \text{val})$, and do:
1. If $l = (\text{read}, \text{adr}, 0)$, set $d := M[\text{adr}]\text{val}$.
2. Else if, $l = (\text{write}, \text{adr}, \text{val})$, set $d := M[\text{adr}]\text{val}$ and $M[\text{adr}] := l\text{val}$.
3. Output additive share $(d)$ to party $P_i$.
Get Memory : On receiving (GetM) from all parties, output additive shares $(M)$ to party $P_i$.

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

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

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

Correctness. Note that the very first execution of Refresh in $\Pi_{\text{DORAM}}$ acts as the instantiation of Init command in $F_{\text{DORAM}}$. 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 $\text{adr}^{\text{read}}_{i}$ such that it is distinct from all previous addresses in the set. All other values are picked
Parameters: A PRF : $K \times [1, m] \rightarrow \mathcal{Y}$, where $m = n + \sqrt{n}$.

Init/Refresh: Each party has input $(M)$ which is parsed as set of tuples $(\text{indx}, \langle \text{val} \rangle)$, where $\text{indx} \in [1, n]$.

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

Access: Each party inputs $(l)$, and gets output $(d)$ at the end. In iteration $c$,

1. Call $F_{\text{st}}$ with input $(\langle \text{Read}, c, (\text{stash}), (l) \rangle)$ and receive $(\text{stash})$.
2. Set $(c + n)$ as: $P_i$ sets its share as $c + n$ and $P_2, P_3$ set their shares to be 0. Parties call $F_{\text{Select}}$ with $(\langle \text{flag} \rangle, \langle c + n \rangle, \langle L_i \rangle)$ to receive $(\text{adr}_{i}^{\text{read}})$.
3. Each party $P_i$ sets $(F_{\text{st}})$ with input $(\langle \text{Access}, (\text{adr}_{i}^{\text{read}}), k_{i-1}, k_{i+1} \rangle)$ and receives $(\langle \text{val} \rangle, \text{adr}_{i}^{\text{read}})$.
4. Call $F_{\text{st}}$ with $(\langle \text{Write}, (\text{val}_{i}), (\text{adr}_{i}^{\text{read}}), c, (\text{stash}) \rangle, \langle (\text{stash}), (l) \rangle)$. All parties receive an updated $(\text{stash})$.
5. Call $F_{\text{Select}}$ with $(\langle \text{flag} \rangle, \langle \text{val} \rangle, \langle (\text{val} \rangle))$ to receive $(d)$.
6. If $c = \sqrt{n}$:
   - (a) Each party $P_i$ calls $F_{\text{st}}$ with $(\langle (M) \rangle, (\text{stash}), L_i, \text{adr}_{i}^{\text{read}}, k_{i-1}, k_{i+1})$ to receive updated shares of $(M)$. Here, $(M)_{n}$ represents the first $n$ tuples in $(M)$.
   - (b) Parties run $\text{Refresh}((M))$.

7. Increment $c := c + 1$ and locally output $(d)$.

Get Memory: Each party receives output the updated share $(M)$ at the end.

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

Fig. 14: Protocol for distributed ORAM, realizing $F_{\text{DORAM}}$.

Independently of the previous iterations. Additionally, in case of multiple iterations, the steps for $F_{\text{GetM}}$ are executed before $F_{\text{Access}}$, and the $i$-th iteration $F_{\text{Access}}$ is simulated if the previous iteration $F_{\text{Access}}$ is executed. The hybrids are as follows.

- $\text{Hyb}_0$: Same as the real world execution except that the outputs are set differently by $F_{\text{DORAM}}$. It replies to the call to $F_{\text{w,m}}$ with $(M_{\text{out}})$. Additionally, it replies to the call for $F_{\text{Select}}$ (Step 5) with $d_{\text{out}}$, where $d_{\text{out}}$ is the current iteration output received from the functionality. This is indistinguishable because of correctness.
- $\text{Hyb}_1$: $F_{\text{DORAM}}$ answers call to $F_{\text{r,m}}$ with randomly sampled $(\text{val}_{i})$ and a randomly sampled address from $L_2 \setminus \text{adr}_{i}^{\text{read}}$. This is indistinguishable because of the security of PRF function.
- $\text{Hyb}_2$: $F_{\text{DORAM}}$ replies with random additive shares in $F_{\text{Access}}$ phase, i.e. a random $\text{stash}_{\text{read}}$, $\text{adr}_{i}^{\text{read}}$, $(\text{stash})$, and $(M)$ (when $c = \sqrt{n} - 1$). This is indistinguishable because of security of additive sharing.
- $\text{Hyb}_3$: During $\text{Init}$, 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.
- $\text{Hyb}_4$: $F_{\text{DORAM}}$ replies calls to $F_{\text{r,m}}(\text{Init})$ 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 $F_{\text{DORAM}}$. 

A.5 Security Proof of $\Pi_{\text{stash}}$

The functionality for reading and writing to stash $(F_{\text{stash}})$ appears in Fig. 16.

**Theorem 4.** Protocol $\Pi_{\text{stash}}$ UC-securely instantiates functionality $F_{\text{stash}}$ in presence of one passive corruption in the $(F_{\text{DORAM}}, F_{\text{Select}}, F_{\text{Zero}})$-hybrid model.

**Proof.** Correctness and security for the simulator are argued as follows.
**Init/Refresh:** On receiving corrupt party’s inputs \((M)\), call \(\mathcal{F}_{\text{DORAM}}\) on \((\text{Init}, (M))\). Simulate transcript as follows:

1. On behalf of \(P_1, P_3\), set access counter \(c := 0\), \((\text{stash}) := [(0, 0, 0)]^\sqrt{n}\), and set \((\text{st}_{\text{stash}}) := \bot\), where \(\text{st}_{\text{stash}} := (\text{flag}, \text{loc}, \text{val})\).
2. From \(P_2\), receive call to \(\mathcal{F}_{\text{init}}\) with \((\text{Init}, (M))\). Sample two \(\text{PRF}\) 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 \(\text{adr}^{\text{read}}_1, \text{adr}^{\text{read}}_3 := \bot\).

**Access:** On receiving inputs of the corrupt party for the current iteration, \((l)\), call \(\mathcal{F}_{\text{DORAM}}\) on Access to receive output \(d_{\text{out}}\). Simulate transcript messages as follows:

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

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

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

**Functionality \(\mathcal{F}_{\text{Stash}}\)**

**Parameters:** Stash size \(\sqrt{n}\), a memory size \(n\).

**Read Stash:** On receiving \((\text{Read}, c, (\text{stash}), (l))\) from all the parties,

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

**Write in Stash:** On receiving \((\text{Write}, (\text{val}), (\text{adr}_M), c, (\text{stash}), (\text{st}_{\text{stash}}), (l))\) from all parties,

1. Reconstruct \(\text{val}_l, \text{adr}_M, \text{stash}, \text{st}_{\text{stash}}\), and \(l\). Parse \(\text{st}_{\text{stash}}\) as \((\text{flag}, \text{loc}, \text{val}_l)\).
2. Update \(\text{stash}[c].\text{adr} := \text{adr}_M\), \(\text{stash}[c].\text{val} := \text{val}_l\), and \(\text{stash}[c].\text{val}_{\text{old}} := \text{val}_l\).
3. If \(\text{lop} = \text{write}\), update \(\text{stash}[\text{loc}].\text{val} := 1.\text{val}\).
4. Output shares \((\text{stash})\) 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 \(L_{\text{stash}}\) maintained by \(P_2, P_3\) is a list of \(\text{PRF}\) evaluations of addresses that were read previously from the memory, i.e., \(\text{adr}_M\), under key \(k_{\text{stash}}\) that is known to \(P_1\). The correctness of \(L_{\text{stash}}\) is guaranteed by correctness of write operation. In each iteration a new address is added to \(L_{\text{stash}}\). In the start of read operation, suppose address \(\text{adr}\) needs to be read. In case it was previously read, \(\text{PRF}(k_{\text{stash}}, \text{adr})\) is present in \(L_{\text{stash}}\), \(\text{adr}' := \text{PRF}(k_{\text{stash}}, \text{adr}) + R\) is revealed to \(P_1\), where \(P_2, P_3\) know \(R\). \(P_1\) generates \(\text{DPF}\) keys which \(P_2, P_3\) evaluate on \(L_{\text{stash}}\) after shifting each entry in \(L_{\text{stash}}\) by \(R\). Thus if \(\text{PRF}(k_{\text{stash}}, \text{adr})\) is present in \(L_{\text{stash}}\), so is \(\text{PRF}(k_{\text{stash}}, \text{adr}) + R\), 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 \( L_{\text{stash}} \) is multiplied by the corresponding index, if the address is present (say at index \( j \)) in \( L_{\text{stash}} \), \( P_2, P_3 \) get secret shares of \( \text{loc} := j \), otherwise they receive shares of 0. Since, they select between this and \( c \) based on flag, this gives them the correct shares of \( \text{loc} \).

Before computing shares of \( \text{val} \), parties convert additive shares of \( \text{stash}.\text{val} \) 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 \( \langle \text{stash}.\text{val} \rangle_j \). Suppose address to be read is present in stash. Then from previous argument, the parties hold secret shares of \( \text{loc} := j \), otherwise, they hold shares of \( \text{loc} := c \). A shifted \( \text{loc}_2 := \text{loc}_2 + r_2 \) is revealed to \( P_2 \), where \( P_1, P_3 \) know \( r_1 \). \( P_2 \) generates DPF keys for \( \text{loc} \), and \( P_1, P_3 \) execute it on their common share \( \langle \text{stash}.\text{val} \rangle_j \) after shifting each index by \( r_1 \). If \( \text{loc} := j \), then there hold additive shares of just one out of three shares of \( \langle \text{val} \rangle \). Otherwise, it is a share of 0. This is done thrice, to obtain the final share \( \text{val} \).

For write operation, first \( L_{\text{stash}} \) is updated with the newest read address \( \text{adr}_M \) by calling \( F_{\text{DOPRF}} \). Since a new address \( \text{adr}_M \) is always made, parties add this in stash at the fixed location \( c \). If \( \text{adr} \) was read in a previous iteration, from correctness of read (as argued above), \( \text{flag} := 1, \text{loc} := j, \text{val}_j := \text{stash}[j].\text{val} \), and, \( \text{flag} := 0, \text{loc} := c, \text{val}_c := 0 \), otherwise. If \( \text{op} = \text{write} \), then \( \text{stash}[j].\text{val} \) should be updated to \( \text{lval} \) (if \( \text{flag} = 1 \)), and otherwise, \( \text{stash}[c].\text{val} \) should be update to \( \text{lval} \) (if \( \text{flag} = 0 \)). In the first case \( \Delta := \text{lval} - \text{stash}[j].\text{val} \), while in the second case, \( \Delta := \text{lval} - \text{val}_c \). Moreover, if \( \text{op} = \text{read} \), then \( \text{stash}.\text{val} \) should remain unchanged, i.e., \( \Delta := 0 \). To compute \( \Delta \), parties obtain the old value \( \langle \text{val}_j := \text{stash}[j].\text{val} \rangle \) or \( \langle \text{val}_c := \text{stash}[c].\text{val} \rangle \) using the first select operation with flag. Using the second \( F_{\text{Select}} \), they obtain the new value \( \langle \text{lval} \rangle \) depending on \( \text{op} \). Observe that, if \( \text{op} = \text{write} \), we have \( \text{val}_\text{old} := \text{stash}[j].\text{val} \) (for \( \text{flag} = 1 \)), and \( \text{val}_\text{old} := \text{stash}[c].\text{val} \) (for \( \text{flag} = 0 \)), and \( \text{val}_\text{new} := \text{lval} \). This gives the right \( \Delta \) for \( \text{op} = \text{write} \). On the other hand, if \( \text{op} = \text{read} \), we have \( \text{val}_\text{new} := \text{val}_\text{old} \) 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 iteration. 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, \( S_{\text{stash}} \) will internally maintain the set \( L_{\text{stash}} \), and in each iteration, it will sample the response for \( F_{\text{DOPRF}} \) (call in Step 1 of write) by excluding entries in \( L_{\text{stash}} \). Other than this, \( S_{\text{stash}} \) 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.

- **Hyb0:** Same as the real world execution except that \( S_{\text{stash}} \) computes outputs differently (steps 10 and 11). It answers calls to \( F_{\text{Zero}} \) with \( \langle \text{stash}.\text{out} \rangle - \delta_i - \langle \text{stash}.\text{val} \rangle \), where \( \delta_i \) and \( \langle \text{stash}.\text{val} \rangle \) are exactly as in the real world execution.
- **Hyb1:** \( S_{\text{stash}} \) answers calls to \( F_{\text{DOPRF}} \) 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.
- **Hyb2:** \( S_{\text{stash}} \) replies to the calls to \( F_{\text{Select}} \) 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.
- **Hyb3:** \( S_{\text{stash}} \) sends random \( w_0, w_2 \) (step 5) instead of correctly masked values. This is indistinguishable because the masks are random and unknown to \( P_1 \).
- **Hyb4:** \( S_{\text{stash}} \) generates DPF keys by running the simulator \( S \) (Step 6). This is indistinguishable from real keys because of the security of the DPF scheme.
- **Hyb5:** \( S_{\text{stash}} \) computes outputs differently. \( S_{\text{stash}} \) answers the call to \( F_{\text{Select}} \) and replies with \( \langle \text{loc} \rangle \) (Step 5). It then computes \( y \) as \( P_1 \) would have (Step 5 in \( H_{\text{stash}} \)), and then answers call to \( F_{\text{Zero}} \) with (flag) \( \rightarrow y \). It continues executing steps in \( H_{\text{stash}} \) and computes \( v \) as \( P_1 \) would have (Step 10d in \( H_{\text{stash}} \)). It answers call \( F_{\text{Zero}} \) with \( \langle \text{val} \rangle - v \) (Step 14). These changes are indistinguishable because of correctness.
- **Hyb6:** \( S_{\text{stash}} \) answers the call to \( F_{\text{DOPRF}} \) by sampling a random \( R \) (Step 2). This is indistinguishable because of the security of additive shares.
- **Hyb7:** \( S_{\text{stash}} \) sends random \( w_0, w_2 \) (Step 10) instead of correctly masked values. This is indistinguishable because the masks are random and unknown to \( P_1 \).
- **Hyb₂**: $S_{stash}$ sends a random vector $z$ to $P_1$ (Step 8). This is indistinguishable, again, because of the additive secret sharing.
- **Hyb₂**: $S_{stash}$ generates DPF keys by running the simulator $S$ (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 $S_{stash}$. □

---

**(Parameters)**: Stash size $\sqrt{n}$, memory size $n$. A PRF: $\mathcal{K} \times [m] \to \mathcal{Y}$. Three DPF schemes: $\Phi_2$ for functions $\mathcal{Y} \to \{0,1\}$, $\Phi_2$ for $\mathcal{Y} \to \{0,1\}$, and $\Phi_3$ for $\mathcal{Y} \to \{0,1\}$. $S$ is the simulator for the DPF schemes.

**Read in Stash**: On receiving corrupt party’s inputs $c, (stash), (\ell)$, call $F_{Stash}$ with Read and the inputs to receive outputs $(s_{stash}) = ((\text{flag}), (\text{loc}), (\text{val}))$. Parse $\ell$ as $(\text{(op)}, (\text{adr}), (\text{val}))$. Simulate transcript as follows:

1. If $c = 1$: $S_{stash}$ initializes an empty list $L_{stash}$ on behalf of $P_3$. Receives call to $F_{DOPRF}$ from $P_2$ and replies nothing. If $P_1$ is corrupt, sample a PRF key $k_{stash} \leftarrow \mathcal{Y}$ and reply to $P_1$ with $k_{stash}$.
2. Receive call to $F_{DOPRF}$ with input $(\text{Eval}, \text{masked}, (\text{adr}))$ from $P_2$, sample $R \leftarrow \mathcal{F}_p$ and send it. If $P_1$ was corrupt, receive $k_{stash}$ in addition and send a random $x \leftarrow \mathcal{F}_p$.
3. Receive $x + R_1$ from $P_2$ on behalf of $P_1$. If $P_1$ is corrupt, sample $w_1, w_2 \leftarrow \mathcal{F}_p$ and send it to $P_1$ on behalf of $P_2$, $P_3$.
4. Run simulator $k_{\text{DFP}}^1 \leftarrow S(1^\lambda, \mathcal{Y}, \ell)$, and send $k_{\text{DFP}}^1$ to $P_2$.
5. Receive call to $F_{Select}$ and reply with $(\text{loc})$ received from $F_{Stash}$.
6. For $j \in \{\text{stash}\}$ compute $y := x + \text{Eval}(k_j, L_{stash}[j], R)$.
7. Receive call to $F_{Zero}$ and reply with $(0) := (\text{flag}) - y$, where $(\text{flag})$ is the value received from $F_{Stash}$.
8. Receive $(\text{stash}.\text{val})_1$ from $P_2$ on behalf of $P_3$. Sample $z \leftarrow \mathcal{F}_p$ and send it to $P_2$.
9. Sample $r_{3,1}, r_{2,1} \leftarrow \mathcal{F}_p$ and send it to $P_2$ on behalf of $P_3$. Receive $r_{2,1}, r_{1,1}$ from $P_2$ on behalf of $P_1$.
10. Send $w_1, w_2 \leftarrow \mathcal{F}_p$ to $P_2$ on behalf of $P_1, P_3$, respectively. Receive $(\text{loc})_1 + r_{2,1}, (\text{loc})_1 + r_{2,3}$ from $P_2$ on behalf of $P_1, P_3$, respectively.
11. Run simulator $k_{\text{DFP}}^1 \leftarrow S(1^\lambda, \mathcal{F}_p, \{0,1\})$, and $k_{\text{DFP}}^2 \leftarrow S(1^\lambda, \mathcal{F}_p, \{0,1\})$. Send $k_{\text{DFP}}^1, k_{\text{DFP}}^2$ to $P_2$ on behalf of $P_1, P_3$. Receive $k_{\text{DFP}}^1, k_{\text{DFP}}^2$ 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 $j \in \{\text{stash}\}$, compute $v := v + (\text{Eval}(1, k_{\text{DFP}}^1, j) \times (\text{stash}., j, \text{val}))_1 + (\text{Eval}(1, k_{\text{DFP}}^2, j) \times z[j])$.
14. Receive call to $F_{Zero}$ from $P_1$. Reply with $(\text{value}) - v$ where $(\text{value})$ is received from $F_{Stash}$.

**Write in Stash**: On receiving corrupt party’s inputs $(\text{value}), (\text{adr}), (\text{ stap}), (\text{stash}), (\text{loc}), (\ell), (\text{value})$, call $F_{Stash}$ with the inputs to receive output $(\text{stash}.\text{val})$. Simulate transcript as follows:

1. Receive call to $F_{DOPRF}$ from $P_2$, and reply with $z' \leftarrow \mathcal{F}_p$.
2. Receive call to $F_{Select}$ from $P_2$ and reply with $y' \leftarrow \mathcal{F}_p$.
3. Receive call to $F_{Select}$ from $P_2$ and reply with $z' \leftarrow \mathcal{F}_p$.
4. Sample $r_{3,1}, r_{2,1} \leftarrow \mathcal{F}_p$ and send it to $P_2$ on behalf of $P_3$. Receive $r_{2,1}, r_{1,1}$ from $P_2$ on behalf of $P_1$.
5. Send $w_1, w_2 \leftarrow \mathcal{F}_p$ to $P_2$ on behalf of $P_1, P_3$, respectively. Receive $(\text{loc})_1 + r_{2,1}, (\text{loc})_1 + r_{2,3}$ from $P_2$ on behalf of $P_1, P_3$, respectively.
6. Run simulator $k_{\text{DFP}}^1 \leftarrow S(1^\lambda, \mathcal{F}_p, \{0,1\})$, and $k_{\text{DFP}}^2 \leftarrow S(1^\lambda, \mathcal{F}_p, \{0,1\})$. Send $k_{\text{DFP}}^1, k_{\text{DFP}}^2$ to $P_2$ on behalf of $P_1, P_3$. Receive $k_{\text{DFP}}^1, k_{\text{DFP}}^2$ from $P_2$ on behalf of $P_1, P_3$, respectively.
7. Initialize a vector $\delta$ of length $|\text{stash}|$ as $(0, \ldots, 0)$.
8. Define $r_1 := r_{2,1} + r_{3,1}$, and $r_3 := r_{2,3} + r_{1,3}$.
9. For $j \in \{\text{stash}\}$, compute $\delta[j] := \delta[j] + \text{Eval}(k_{\text{DFP}}^1, j + r_1) + \text{Eval}(k_{\text{DFP}}^2, j + r_3)$.
10. Receive call to $F_{Zero}$ from $P_2$. Reply with $(\text{stash}.\text{val}) - \delta - (\text{stash}.\text{val})$.
11. Receive two calls to $F_{Zero}$ from $P_2$. Reply with $(\text{stash}.\text{val}) - (\text{stash}.\text{adr})$ and $(\text{stash}.\text{val}) - (\text{stash}.\text{val})$.

![Fig. 17: Simulator for Theorem 4 for reading and writing to stash.](image)

---

### A.6 Security Proof of $\Pi_{r,M}$

**Theorem 5.** Protocol $\Pi_{r,M}$ UC-securely instantiates functionality $F_{r,M}$ in the presence of one passive corruption in the $(F_{DOPRF}, F_{3-OT})$-hybrid model.
Proof. Correctness and Security are argued below.

**Correctness.** It follows from the correctness of $F_{3\cdot OT}$, $F_{DOPRF}$, 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 straightforward way. The simulator for the case of corrupt $P_2$ appears in Fig. 6. $S_{r,M}$ generates an indistinguishable transcript because of the following hybrids.

- **Hyb0:** Same as the real world execution except that the simulator sets $(x)$ as $(d) - y$, where $(d)$ is obtained from $F_{r,M}$. This is indistinguishable as in the real execution $(d) := x + y$ as well.
- **Hyb1:** Reply to $F_{DOPRF}$ is set as $adr' = adr^{read} \setminus adr^{read}_2$, where $adr^{read}$ is obtained form $F_{r,M}$. Again, this is indistinguishable from the previous hybrid because there too this set is updated similarly.
- **Hyb2:** Same as Hyb1 except $M^*$ is picked at random. This is indistinguishable from Hyb1 because the output of $F_{3\cdot OT}$ 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 $S_{r,M}$. □

---

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

**Functionality $F_{r,M}$**

**Parameters:** A PRF $: K \times [m] \rightarrow Y$.

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

---

**Fig. 19:** Simulator for Theorem 5 for reading from memory.
A.7 Security Proof of $\Pi_{w-M}$

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

**Functionality $F_{w-M}$**

**Update:** On receiving $(\langle M \rangle, \langle \text{stash} \rangle, L_i, \text{adr}_{i}^{\text{rad}}, k_{i-1}, k_{i+1})$ from each $P_i$,
1. Reconstruct $M$ and stash.
2. For $j \in [\sqrt{n}]$, set $\text{adr} := \text{stash}[j].\text{adr}$ and $\text{val} := \text{stash}[j].\text{val}$. Update $M[\text{adr}].\text{val} := \text{val}$.
3. Output $(M)$ to all parties.

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

**Parameter:** An MPDPF scheme $\Phi = (\text{Gen}, \text{Eval})$ for input domain $[m]$ and output domain $F_p$. Let $S$ be the simulator for this scheme.

**Update:** On receiving corrupt party’s inputs $(\langle M \rangle, \langle \text{stash} \rangle, \text{adr}_2^{\text{rad}}, L_2, k_1, k_3)$, call $F_{w-M}$ on inputs to receive output $(M)$. 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 $mk_{2,i} \leftarrow S(1^\lambda, m, F_p)$.
   (b) Send key $mk_{2,i}$ to $P_2$ on behalf of $P_i$.
3. Receive call to $F_{\text{Zero}}(n)$.
   (a) For $j \in [n]$, evaluate $x'_j := \Phi.\text{Eval}(mk_{2,2}, L_1^{-1}(\text{PRF}(k_1,j)))$
   and $y'_j := \Phi.\text{Eval}(mk_{3,2}, L_3^{-1}(\text{PRF}(k_3,j)))$.
   (b) For $j \in [n]$, set $\alpha_2[j] := \langle M \rangle_{2,j} - \langle M \rangle_{2,j} - x'_j - y'_j$.
   (c) Return $\alpha_2$.

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 $\text{adr}$ is stored at $(a$ unique $)$ index $L_i^{-1}(\text{PRF}(k_i, \text{adr}))$ in the vector $\{\Phi.\text{Eval}(mk_{i-1,i}, L_i^{-1}(\text{PRF}(k_i,0))) + \Phi.\text{Eval}(mk_{i-1,i}, L_i^{-1}(\text{PRF}(k_i,m-1)))\}$. 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 $L_i^{-1}$ all updates are recorded in this vector. Update to position $j$ in $M$ with one share, is then $M[j].\text{val} + \{\Phi.\text{Eval}(mk_{i-1,i}, L_i^{-1}(\text{PRF}(k_i,j)))\}$, 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 $S_{w-M}$ appears in Fig. 21, and we argue indistinguishability with the following hybrids.

- $\text{Hyb}_0$: Same as the real world execution except that the output of the call to the functionality $F_{\text{Zero}}$ is answered by setting $\alpha_2$ as $\langle M' \rangle_2 - \langle M \rangle_2 - x - y$, where $\langle M' \rangle_2$ is the value received from $F_{w-M}$. This is indistinguishable since in the real protocol as well, $\langle M' \rangle_2 := \alpha_2 + \langle M \rangle_2 + x + y$. 

29
the presence of one passive corruption.

Let Correctness.

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

Theorem 7. Protocol II\textsubscript{DOPRF} UC-securely instantiates functionality $F_{DOPRF}$ for mode = unmasked in the presence of one passive corruption.

Proof. Correctness. Let $x_1, x_2, x_3$ be $K, R_1, R_2$’s additive share of $x$, resp. From the protocol,

$$c_1 = b \cdot (s \circ s) - b \cdot (d) - c_3$$

$$w = (s \circ s) \cdot (k + x_1 + x_2 + m)$$

Substituting these in $z_1$,

$$z_1 = (x_3 - m - b) \cdot (s \circ s) + b \cdot (s \circ s)$$

$$- b \cdot (d) - c_3 + (s \circ s) \cdot (k + x_1 + x_2 + m)$$

$$= (x + k) \cdot (s \circ s) - b \cdot (d) - c_3$$

Substituting value for $z_3$,

$$z_3 = d \cdot y_3 + c_3 - d \cdot e$$

$$= d \cdot (x_3 - m) + c_3 - d \cdot (x_3 - m - b)$$

$$= c_3 + d \cdot b$$

$$z = z_1 + z_3 = (x + k) \cdot (s \circ s)$$

The correctness of the final output is guaranteed by the multiplicative property of Legendre symbol, and since $L_p(s \circ s) = 1^l$, i.e., $L_p((x + k) \cdot (s \circ s)) = L_p(x + k)L_p(s \circ s)$. Note that, if for any $j$, there is
some $s_j \in s$ such that $s_j = 0$, then the correctness guarantees fail. However, this happens with negligible probability since $s$ 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 $(x)$ 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 $\text{KGen}$ as: on receiving output $k$ from the functionality, sample $k_1, k_2 \leftarrow \mathbb{F}_p^2$ such that $k_1 + k_2 = k$. Send $k_1, k_2$ to $K$ on behalf of $R_1, R_2$. 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 $(x)$, and key $k$, and calls $\mathcal{F}_{\text{DOPRF}}$ on mode = unmasked. It receives no output in return. It simulates the transcript as follows. It samples $c_1 \leftarrow \mathbb{F}_p^d$ and $y_2 \leftarrow \mathbb{F}_p$, and sends them to $K$ on behalf of $R_2$. It samples $c \leftarrow \mathbb{F}_p$ and sends it to $K$ on behalf of $R_2$. Receive $w, z_1$ 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: $c_1$ is masked with $c_3$, $y_2$ 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 $k_{k,1}$ to $R_1$ on behalf of $K$. Then it receives the corrupt party’s input $(x)$ and calls $\mathcal{F}_{\text{DOPRF}}$ with mode = 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 $k_{2,K}$ 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 $(x)$ and calls $\mathcal{F}_{\text{DOPRF}}$ with mode = unmasked to receive output $o$. It simulates the transcript as follows. Receive $e$ from $R_2$ on behalf of $K$. Locally compute $z_3$ just as $R_2$ would, i.e., since $s, d, m, c_3, b$ are sampled by $R_2$ using shared PRG seeds, the simulator can also obtain these values and compute $y_3, e$, and $z_3$ just as $R_2$ would. It then sets $z_1 := o - z_3$ and sends $z_1$ to $R_2$. This concludes the simulation. Since the only message that $R_2$ receives in the protocol execution is $z_1$, which is set in the particular way such that $o = z_1 + z_3$, it is distributed identically to the real world view. 

**Theorem 8.** Protocol $\Pi_{\text{DOPRF}}$ UC-securely instantiates functionality $\mathcal{F}_{\text{DOPRF}}$ for mode = 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 $c_3, w$, we get

$$z_3 = (k + x_1 - m - a) \cdot (t) + (a) \circ (t - e) - c_1 + t \cdot (x_1 + x_3 + m)$$

$$= (k + x) \cdot t - a \circ e - c_3$$

Similarly, for $z_1$, and $z$ we get,

$$z = z_1 + z_3 = (k + x) \cdot t$$

Consider a single bit $r_j \in r$, and corresponding $t_j \in t$, $k_j \in k$, $s_j \in s$, $o_j \in o$. 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 $r_j = 1$, $t_j = s_j^2 \cdot a$. 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 $\text{KGen}$ as: on receiving output $k$ from the functionality, sample $k_3, k_3 \leftarrow \mathbb{F}_p^2$ such that $k_2 + k_3 = k$. Send $k_3, k_3$ to $K$ on behalf of $R_3, R_2$. 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 $\langle \ldots \rangle$, $k$ and calls $F_{\text{DORF}}$ with $\text{mode} = \text{masked}$ and obtains output $y \oplus r$. Receive $d$ on behalf of $R_2$. Given common PRG seeds and inputs of the corrupt party, the simulator can locally compute $z_1$ just as $K$ would have. It then sets $z_3 := y \oplus r - z_1$ and sends it to $K$. This is indistinguishable from the real world because of the programming of $z_3$ and because of correctness of the scheme.

Next, suppose that $R_1$ is corrupt. The simulator receives input $\langle x \rangle$ and calls $F_{\text{DORF}}$ to receive $r$ as output. The simulator receives $r_1$ from $R_1$ on the behalf of $R_2$, sets $r_2 := r - r_1$ and sends it to $R_1$. This concludes simulation and is clearly indistinguishable from the real world.

Finally, suppose $R_2$ is corrupt. The simulator receives input $\langle x \rangle$ and calls $F_{\text{DORF}}$ to receive $r$ as output, and, just as in the previous case, fixes $r_1 := r - r_2$. It then samples $y_2 \leftarrow F_p, c_3, d \leftarrow F_p$ to $R_2$. This is indistinguishable as all three values are masked by $m, c_1$ and $a$, resp.

\section{Other Functionalities}

\begin{table}[h]
\centering
\begin{tabular}{|l|}
\hline
\textbf{Functionality $F_{\text{Select}}$} \\
\hline
\textbf{Select:} On receiving $\langle \langle \text{flag} \rangle, \langle x \rangle, \langle y \rangle \rangle$ from all the parties, \\
1. Reconstruct $x, y, \text{ and flag}$. \\
2. $d = \text{flag} \cdot x + (1 - \text{flag}) \cdot y$. \\
3. Output $\langle d \rangle$ to all parties. \\
\hline
\end{tabular}
\caption{Functionality for parties $P_1, P_2, P_3$ for obliviously selecting between one out of two values.}
\end{table}

\begin{table}[h]
\centering
\begin{tabular}{|l|}
\hline
\textbf{Functionality $F_{\text{MPC}}$} \\
\hline
\textbf{Parameter:} Function description $f$. \\
\textbf{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. \\
\hline
\end{tabular}
\caption{Functionality for parties $P_1, P_2, P_3$ for securely computing a function $f$ on additive shares.}
\end{table}

\begin{table}[h]
\centering
\begin{tabular}{|l|}
\hline
\textbf{Functionality $F_{\text{Zero}}$} \\
\hline
\textbf{Zero$(n)$:} On receiving command and input $n$ from all parties, generate random additive shares of $\alpha = 0^n$, and output $\langle \alpha \rangle$ to all parties. \\
\hline
\end{tabular}
\caption{Functionality for parties $P_1, P_2, P_3$ to obtain random additive shares of 0.}
\end{table}

\section{Further Benchmark Results}

32
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.

<table>
<thead>
<tr>
<th>$\log_2 n$</th>
<th>Time per Access in ms</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>LAN</td>
</tr>
<tr>
<td></td>
<td>Online</td>
</tr>
<tr>
<td><strong>Ramen (This Work)</strong></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>2.51</td>
</tr>
<tr>
<td>9</td>
<td>3.80</td>
</tr>
<tr>
<td>10</td>
<td>2.09</td>
</tr>
<tr>
<td>11</td>
<td>1.54</td>
</tr>
<tr>
<td>12</td>
<td>2.77</td>
</tr>
<tr>
<td>13</td>
<td>1.86</td>
</tr>
<tr>
<td>14</td>
<td>3.57</td>
</tr>
<tr>
<td>15</td>
<td>5.20</td>
</tr>
<tr>
<td>16</td>
<td>7.87</td>
</tr>
<tr>
<td>17</td>
<td>11.69</td>
</tr>
<tr>
<td>18</td>
<td>12.31</td>
</tr>
<tr>
<td>19</td>
<td>17.88</td>
</tr>
<tr>
<td>20</td>
<td>22.51</td>
</tr>
<tr>
<td>21</td>
<td>26.81</td>
</tr>
<tr>
<td>22</td>
<td>29.70</td>
</tr>
<tr>
<td>23</td>
<td>33.25</td>
</tr>
<tr>
<td>24</td>
<td>38.85</td>
</tr>
<tr>
<td>25</td>
<td>49.98</td>
</tr>
<tr>
<td>26</td>
<td>64.97</td>
</tr>
<tr>
<td><strong>Three-Party Duoram [VHG23]</strong></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>0.28</td>
</tr>
<tr>
<td>9</td>
<td>0.16</td>
</tr>
<tr>
<td>10</td>
<td>0.16</td>
</tr>
<tr>
<td>11</td>
<td>0.19</td>
</tr>
<tr>
<td>12</td>
<td>0.21</td>
</tr>
<tr>
<td>13</td>
<td>0.23</td>
</tr>
<tr>
<td>14</td>
<td>0.39</td>
</tr>
<tr>
<td>15</td>
<td>1.07</td>
</tr>
<tr>
<td>16</td>
<td>1.41</td>
</tr>
<tr>
<td>17</td>
<td>3.64</td>
</tr>
<tr>
<td>18</td>
<td>6.89</td>
</tr>
<tr>
<td>19</td>
<td>11.67</td>
</tr>
<tr>
<td>20</td>
<td>17.01</td>
</tr>
<tr>
<td>21</td>
<td>31.44</td>
</tr>
<tr>
<td>22</td>
<td>63.67</td>
</tr>
<tr>
<td>23</td>
<td>124.57</td>
</tr>
<tr>
<td>24</td>
<td>238.60</td>
</tr>
<tr>
<td>25</td>
<td>464.33</td>
</tr>
<tr>
<td>26</td>
<td>922.67</td>
</tr>
</tbody>
</table>
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.

<table>
<thead>
<tr>
<th>Bandwidth in Mbit/s</th>
<th>Online</th>
<th>Prep.</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>59.50</td>
<td>56.25</td>
<td>115.75</td>
</tr>
<tr>
<td>50</td>
<td>25.54</td>
<td>30.83</td>
<td>56.36</td>
</tr>
<tr>
<td>100</td>
<td>21.33</td>
<td>28.54</td>
<td>49.87</td>
</tr>
<tr>
<td>1000</td>
<td>18.35</td>
<td>25.90</td>
<td>44.26</td>
</tr>
<tr>
<td>9420</td>
<td>18.13</td>
<td>25.68</td>
<td>43.81</td>
</tr>
</tbody>
</table>

(b) With varying latency.

<table>
<thead>
<tr>
<th>RTT in ms</th>
<th>Online</th>
<th>Prep.</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0</td>
<td>17.86</td>
<td>25.67</td>
<td>43.54</td>
</tr>
<tr>
<td>5.0</td>
<td>75.82</td>
<td>25.63</td>
<td>101.45</td>
</tr>
<tr>
<td>10.0</td>
<td>129.15</td>
<td>25.74</td>
<td>154.89</td>
</tr>
<tr>
<td>20.0</td>
<td>236.93</td>
<td>25.95</td>
<td>262.88</td>
</tr>
<tr>
<td>30.0</td>
<td>342.35</td>
<td>26.11</td>
<td>368.46</td>
</tr>
</tbody>
</table>

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 \( n = 2^{22} \).

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