2PC Memory-Manipulating Programs with Constant Overhead*

David Heath
daheath@illinois.edu
University of Illinois Urbana-Champaign

Abstract

General-purpose secure multiparty computation (MPC) remains bottlenecked in large part by a lack of efficient techniques for handling memory access. We demonstrate a remarkably simple and efficient 2PC instantiation of random access memory (RAM), based on distributed point functions (DPFs, Gilboa and Ishai, Eurocrypt'14). Our semi-honest 2PC protocol can be achieved from oblivious transfer (OT) and a black-box pseudorandom generator (PRG).

For a memory storing large enough data words, our 2PC RAM incurs *constant* communication overhead per access. Like prior works using DPFs to achieve memory access, our work incurs linear *computation* per access, but per-access *communication* is lean.

Our 2PC RAM is built on top of an obliviousness-friendly model of computation called the single access machine model (SAM, Appan et al., CCS'24). In the SAM model, each memory slot can be read at most once. We present a simple 2PC SAM protocol, where each single-access memory operation incurs at most $O(w + \lambda \lg n)$ bits of communication, where w is the word size, n is the number of memory words, and λ is a security parameter. Of this cost, only $2w + 2 \lg n$ bits are incurred in the online phase.

There are now many oblivious algorithms that compile directly to SAM more efficiently than via a compilation to RAM, and our 2PC SAM can instantiate these algorithms. As one example, we can use our 2PC SAM to implement privacy-preserving graph traversal (DFS or BFS) over a secret-shared size-n graph while revealing nothing beyond the runtime of the SAM program. Our construction achieves online communication $O(n \lg n)$ *bits*, asymptotically matching the number of bits touched in a corresponding cleartext graph traversal.

Keywords: MPC, Oblivious RAM, Distributed Point Functions

*Abridged version appears in proceedings of ACM CCS 2026.

Contents

1 Introduction **3**
1.1 Contribution 5
1.2 Technical Overview 6

2 Preliminaries **7**
2.1 Notation 7
2.2 Security Model and Functionalities 7
2.3 Distributed Point Functions 8

3 Review of the Single Access Machine Model **10**
3.1 How to Implement RAM with SAM 11
3.2 How to Implement Pointers with SAM 13

4 Technical Approach **14**

5 Experimental Evaluation **15**
5.1 Implementation Details 16
5.2 Experimental Results 17

6 Related Work **20**

A Details for Periodic RAM Resets **27**

B Details of DPF Shifting **29**

C Security Proof **29**

D Noisiness in round complexity of SAM-based RAM **31**

1 Introduction

Secure multiparty computation (MPC) allows mutually untrusting parties to securely run programs on their joint, private inputs. This work focuses on two-party computation (2PC) protocols secure against a *semi-honest* adversary, and our goal is to improve *bandwidth consumption*. We in particular wish to improve 2PC cost for arbitrary and potentially complex programs.

We distinguish a protocol's *offline* and *online* phases. The offline phase can be executed before party inputs are known, allowing parties to prepare "overnight", while computational resources are not in contention. The goal in an offline-online protocol is to move cost to the offline phase, allowing for a lightweight online phase.

The problem of memory manipulation In theory, 2PC/MPC protocols allow parties to execute *arbitrary* programs $\mathcal{P}$, and the community's foundational techniques work when $\mathcal{P}$ is expressed as a poly-size circuit [Yao86, GMW87]. Expressing $\mathcal{P}$ as a circuit can be problematic, particularly when $\mathcal{P}$ manipulates a large main memory storing n data words, each of size w bits. In general, compiling a memory-manipulating program to a circuit incurs $\Theta(nw)$ gates *per memory access*, due to the need to compile each access to a subcircuit that linearly scans all of memory. While the resulting circuit remains poly-size, it is concretely large, and MPC's communication cost typically scales with circuit size.

ORAM and RAM-MPC Fortunately, we can achieve cheaper MPC for memory-manipulating programs by incorporating *Oblivious RAM* (ORAM). ORAM is a cryptographic technique allowing a memory-constrained client to securely outsource its data storage to an untrusted server [GO96]. In short, ORAM allows the client to obfuscate its logical access pattern, hiding the logical address accessed at each step. Roughly, MPC and ORAM are combined by letting the MPC circuit play the role of the ORAM client, and letting the MPC parties jointly implement the ORAM server, where the parties hold *secret shares* of the ORAM server's main memory; see [OS97, GKK$^+$12, KM19, BKKO20, HV21, FNO22, VHG23, FNO25] and others. This arrangement is sometimes called *distributed ORAM* (DORAM) or *RAM-MPC*.

Remark 1 (Multi-server ORAM versus RAM-MPC)**.** *In some cases the term DORAM refers to a client/multi-server interaction, and in others it refers to a full MPC RAM, where the ORAM client is intended to be an MPC circuit. We use DORAM to refer to the former setting and RAM-MPC to refer to the latter. The DORAM setting is easier than full RAM-MPC, since in DORAM it is efficient to, e.g., have the client use a pseudorandom function to decide where to store data items on the servers; see e.g. [GKW18]. While this remains possible in the full MPC setting, it becomes prohibitively expensive, due to the implied non-black-box use of cryptographic primitives. We focus on the latter more challenging RAM-MPC setting where all memory inputs/outputs are simple additive secret shares, and all cryptography is black-box, even when we instantiate a client with an MPC protocol.*

The three main costs to consider in RAM-MPC are the per-access amortized bandwidth consumption (communication complexity), latency (round complexity), and computation. We can consider how these three costs scale as functions of the size of memory n and the bit-width of memory elements w. Of these three costs, computation is traditionally viewed as less important than the others, especially when the necessary computation can be parallelized and/or hardware-accelerated. Thus, we are often willing to increase computation cost if it means improving communication.

Trading Computation for Communication with Distributed Point Functions By using an appropriate off-the-shelf ORAM, e.g. Circuit ORAM [WCS15], it is possible to obtain a RAM-MPC protocol where all three costs scale linearly in w and at most polylogarithmically in n. While this is *far* preferable to compiling to linear scans, the polylogarithmic factors and moderate hidden constants of such an approach mean that memory accesses remain relatively expensive.

Of course, it is not necessary to use an *off-the-shelf* ORAM. The MPC setting is richer in its capabilities than the bare client/server ORAM setting, and we can leverage this fact to design a custom approach with lower cost. The literature has given many RAM-MPC constructions of this kind. Here, we discuss the two most relevant prior works; see Section 6 for discussion of other techniques.

	Model	Setting	Assumes	Offline		Online		
				Comm. (bits)	Comp.	Rounds	Comm. (bits)	Comp.
Ours	SAM	2PC	OT	$O(w + \lambda \lg n)$	$\tilde{O}(wn)$	1	$2w + 2\lg n$	$\tilde{O}(wn)$
	RAM			$O(w + \lambda(\lg^2 n / \lg B))$	$\tilde{O}(wn)$	$O(\lg n / \lg B)$	$2w + O(B(\lg^2 n / \lg B))$	$\tilde{O}(wn)$
[AHR24]	SAM	2PC	OT	—	—	$\tilde{O}(1)$	$O(w\lambda)$	$\tilde{O}(w)$
via [WCS15]	RAM			—	—	$\tilde{O}(1)$	$O((w + \lg^2 n)\lambda)$	$\tilde{O}(w)$
[ZWR+16]	RAM	2PC	OT	—	—	$O(\lg n)$	$\tilde{O}(w\sqrt{n})$	$\tilde{O}(w\sqrt{n})$
[Ds17]	RAM	2PC	OT	—	—	$O(\lg n)$	$\tilde{O}(w\sqrt{n})$	$\tilde{O}(wn)$
[VHG23]	RAM	2PC	FHE	—	—	$O(1)$	$\tilde{O}(w)$	$\tilde{O}(wn)$
		3PC	OWF	$O(w + \lambda \lg n)$	$\tilde{O}(wn)$	$O(1)$	$O(w)$	$\tilde{O}(wn)$
[JW18]	RAM	3PC	OWF	—	—	$O(\lg n)$	$O(\lambda \lg^3 n + w \lg n)$	$\tilde{O}(w)$
[BKKO20]	RAM	3PC	OWF	—	—	$O(1)$	$\tilde{O}(w\sqrt{n})$	$\tilde{O}(wn)$
[HV21]	RAM	2PC	OPRF	—	—	$O(1)$	$\tilde{O}(w\sqrt{n})$	$\tilde{O}(w\sqrt{n})$
[HKO23]	SAM	2PC	OT	$O(w \lg n \lg\lg n\lambda)$	$\tilde{O}(w)$	0	0	$\tilde{O}(w)$
	RAM			$O((w + \lg^2 n) \lg n \lg\lg n\lambda)$	$\tilde{O}(w)$			
[FNO25]	RAM	3PC	None	—	—	$O(\lg n)$	$O((w + \lg^2 n)\frac{\lg n}{\lg\lg n})$	$\tilde{O}(w)$

[1] We list results for both SAM and RAM; see comparison of the models in Section 3.
[2] SAM operations `alloc`/`read`/`write` may incur different costs; we list the maximum of those costs.
[3] w denotes word size (bits), n denotes memory capacity (words), and λ is a security parameter.
[4] To implement RAM by SAM, one builds RAM recursively, instantiating a search tree; Parameter $2 \le B \le n$ sets the tree's degree. As B increases, the tree depth (and hence round complexity) decreases; see also Section 3.
[5] $\tilde{O}(\cdot)$ hides arbitrary $\mathrm{poly}(\lg n, \lambda)$ factors. Therefore, for example, $\tilde{O}(1)$ should be considered inferior to $O(\lg n)$, but not inferior to $O(\sqrt{n})$.
[6] When we do not list offline cost it is because the construction does not naturally asymptotically benefit from moving cost to an offline phase.
[7] All protocols that list an offline phase can preprocess arbitrary numbers of accesses within $O(\lg n)$ rounds.

Table 1: Cost comparison of our approach with prior work.

One elegant example of a custom RAM-MPC is *Floram* [Ds17]. In Floram, the two parties locally touch their shares of *each memory slot on each memory access*; thus, computation cost is linear. In exchange, Floram enjoys decent asymptotic communication cost and superb constant factors. Indeed, while Floram is asymptotically inferior to other off-the-shelf ORAMs, e.g. Circuit ORAM [WCS15], the authors found that their simple protocol achieved lower wall-clock runtime performance for a wide range of parameters.

In short, Floram leverages *distributed point functions* (DPFs, [GI14, BGI16]) to read/write to main memory in a secret-shared fashion; see Section 2 for a review of DPFs. As stated above, their Floram construction requires $\tilde{O}(wn)$ computation per access, but only $\tilde{O}(w\sqrt{n})$ bits of communication, where $\tilde{O}(\cdot)$ hides $\text{poly}(\lg n, \lambda)$ factors and where λ is a security parameter.

More recently, *Duoram* [VHG23] built on [Ds17]'s insight to achieve a RAM-3PC protocol with linear computation cost and—remarkably—only $O(w + \lambda \lg n)$ communication per access. Thus, for large enough words of size $w = \Omega(\lambda \lg n)$ bits, Duoram achieves *constant* communication overhead. Like Floram, Duoram also enjoys low constant factors and practically good performance. The authors also extend Duoram to the RAM-2PC setting, but to do so they leverage a particularly communication-efficient single-server private information retrieval (PIR) protocol called Spiral [MW22]. This PIR protocol builds on fully homomorphic encryption (FHE), and it is far less practical than the simple techniques [VHG23] use in the 3PC setting.

Single Access Machines; SAM-MPC Our goal is to further improve the asymptotic and concrete cost of semi-honest MPC handling of memory-manipulating programs. To do so, rather than directly targeting RAM programs, we build on the work of [AHR24, PAR+26] which proposed a weakening of RAM called the single access machine (SAM) model. The SAM model is essentially identical to the RAM model, except that each main memory slot may only be written[1] at most once, and read at most once.

Remark 2 (Compiling RAM to SAM)**.** *While we work with the SAM model, our approach implies results for the more standard RAM model. This is because RAM programs can be compiled to SAM programs. SAM can implement programs that manipulate tree-based data structures, so one can implement a random access memory as a search tree. This approach is reminiscent of the* position map *technique used in the ORAM literature; see Section 3 and [AHR24] for details. We choose to work with SAM rather than RAM both because our SAM results are more general, due to the above compilation strategy, and because the semantics of SAM are a natural match for our protocol.*

1.1 Contribution

2PC for SAM programs Following [Ds17, VHG23], we construct MPC handling of memory-manipulating programs on top of distributed point functions (DPFs). Our two-party protocol supports secure evaluation of arbitrary functions expressed as SAM programs. Our semi-honest 2PC protocol is built on only oblivious transfer (OT) and a black-box pseudorandom generator (PRG). The amortized cost of each memory access is as follows:

- **Communication:** The parties in an offline phase perform $O(\lg n)$ OTs and exchange $O(w + \lambda \lg n)$ bits, where λ is the length of a PRG seed. In the online phase, the parties exchange only $2 \lg n + 2w$ bits!
- **Rounds:** The offline phase, which generates correlated randomness for any polynomial number of accesses, takes $O(\lg n)$ total rounds. In the online phase, each access takes 1 round.
- **Computation:** In both the offline and online phase, the parties handle each access in $\tilde{O}(wn)$ time. Offline and online computation is highly parallelizable.

Our protocol also supports parallel accesses. Any number of non-dependent memory accesses (i.e., the input to one access does not depend on the output of another) can be handled in one round.

Table 1 compares our approach to relevant prior work in terms of setting, cryptographic assumption, online round complexity, and online/offline communication/computation. Section 6 provides additional discussion of prior work.

[1] In fact, our approach achieves a slightly stronger notion, where each memory cell may be written repeatedly; the cell accumulates an XOR of all values written to it.

We highlight that our 2PC SAM protocol is remarkably simple, easy to implement, and has excellent online communication cost.

We implemented our end-to-end SAM protocol and evaluated it on two benchmarks: an end-to-end 2PC RAM protocol, and a SAM-based 2PC graph depth-first search. Our protocol demonstrates considerable concrete improvement over prior work. The code is available[2] online. Section 5 describes our implementation/benchmarks and examines performance.

2PC for pointer-machine programs and immutable memory programs [AHR24] showed that any *pointer-machine program* can be compiled to a SAM program [Sch80, KU58]. The pointer machine model is one where memory is structured as a graph, roughly akin to objects in an object-oriented programming language. A pointer-machine program may not use random access arrays, but it may use data structures based on explicit management of *pointers to objects*, e.g. stacks, queues, trees, graphs, etc. Recent work [PAR$^+$26] improved the SAM-based handling of pointer machine programs by a significant constant factor; we use their techniques in our implementation.

[AH25] further showed that any program manipulating an *immutable memory*—i.e. where memory slots can be written to once, then arbitrarily read—can also be compiled to a SAM program with constant overhead. This brings further programs into scope, including those written in purely functional programming languages (i.e. those based on Church's foundational Lambda Calculus), as well as rich data structures like suffix trees and multidimensional range trees.

Combining our 2PC SAM protocol with the [AHR24, AH25, PAR$^+$26] compilers yields a 2PC protocol for pointer-machine programs and immutable memory programs. Such results were not known from prior techniques. In the following, we refer to the *in-degree* of an object, which is the number of existing copies of pointers to that object:

Corollary 1 (2PC for Constant-Degree/Immutable Pointer Machine Programs)**.** *Assuming OT, there exists a semi-honest 2PC protocol* Π *for any computation expressed as a pointer-machine program. If the pointer machine program runs in time* n *and if all objects are either (1) immutable or (2) have constant*[3] *in-degree, then* Π *consumes* $O(n \lg n)$ *bits of online communication and* $O(n \lg n)$ *OTs.*

This result is remarkable in the sense that a *cleartext* constant in-degree pointer machine program will necessarily touch $O(n \lg n)$ bits (pointers are necessarily of size $\Theta(\lg n)$ bits), so the online phase of the above protocol incurs constant communication overhead.

1.2 Technical Overview

Using DPFs to achieve PIR write The main building block of our protocol is that of a "secret-shared PIR write". Namely, the parties hold a secret-shared index α and a secret-shared value β, and they wish to construct a secret-shared length-n array that holds zero everywhere, except at index α, where the array holds β. This building block can be efficiently instantiated from a random, preprocessed DPF and $2w + 2\lg n$ bits of online communication, where w is the bit-length of array elements; see Section 2 for details.

Protocol Sketch With this PIR write mechanism in hand, our basic 2PC SAM protocol is informally as follows: To write a secret-shared value to memory, the parties use the above PIR write to efficiently and obliviously save secret shares of that value to a randomly chosen (without replacement) memory slot. To read secret shares of the value back later, the parties reveal to themselves the randomly chosen memory slot in the clear, then fetch their secret shares. This revelation is secure because (1) the location is random and (2) the SAM model ensures that we will read each location at most once. At least at a high level, this is the entire protocol.

The simplicity of this protocol leads to its low cost: Each memory write consumes a single DPF, and each memory read is almost free. DPFs can be prepared in the offline phase, and in the online phase, each memory operation consumes only a small number of bits.

[2] `https://github.com/DAHeath/sam-2pc`

[3] If the pointer-machine program manipulates mutable objects with higher in-degree, then each time it reads a pointer to an object with in-degree d, it incurs $O(\lg n \lg d)$ bits of online communication [AHR24].

Despite the simplicity of this protocol, to our knowledge it has not been observed by prior work, possibly because it naturally instantiates SAM—not RAM—and SAM was formalized only relatively recently [AHR24]. It is worth pointing out that when we use the above simple protocol to instantiate RAM (by compiling the RAM program to a SAM program; see Section 3), this basic approach improves over prior work Floram [Ds17] by reducing asymptotic overhead and over prior work Duoram [VHG23] by removing the expensive use of single-server PIR.

2 Preliminaries

2.1 Notation

- lg denotes the base-2 logarithm. Namely, $\lg x \triangleq \lceil \log_2 x \rceil$.
- n denotes the size of memory in words; w denotes word size in bits. For simplicity, we assume n is a power of two.
- In general, we use $\alpha \in \{0,1\}^{\lg n}$ to denote a memory address and $\beta \in \{0,1\}^w$ to denote a memory value.
- λ denotes a security parameter (e.g., the length of PRG seeds).
- $x \triangleq y$ denotes that x is equal to y by definition.
- $x \approx y$ denotes that ensembles x and y are computationally indistinguishable (in λ).
- Let $x \in \{0,1\}^*$ denote a string. $[\![x]\!]$ denotes a two-party XOR sharing of x. We denote P_0's share by $[\![x]\!]_0$ and P_1's share by $[\![x]\!]_1$. Namely, $[\![x]\!]_0 \oplus [\![x]\!]_1 = x$.
- We sometimes use $\tilde{O}(\cdot)$ notation, especially for computation costs. This notation hides $\mathrm{poly}(\lg n, \lambda)$ factors.

We consistently use oblivious operations on XOR secret shares. Recall that XOR operations are communication free, whereas non-linear gates are not. Many of our non-linear operations are vectorized; for instance, we consistently multiply an entire secret-shared vector by a secret-shared bit. Note that in the two-party setting, such a vector-scalar multiplication can be achieved from two oblivious transfers.

2.2 Security Model and Functionalities

Security model. We prove semi-honest simulation-based security in the $\mathcal{F}_{\mathsf{pre}}$-hybrid model.

Our SAM functionality uses the standard convention, common in ideal functionalities for secret-shared reactive state, that the simulator may choose the corrupted party's output share on Alloc and Read, with the functionality setting the honest party's share accordingly. This matches the real protocol: read outputs are shares already stored in the parties' local state, rather than freshly rerandomized shares. Re-randomizing these shares would require extra communication.

Functionalities We build our SAM protocol assuming access to a preprocessing functionality $\mathcal{F}_{\mathsf{pre}}$; see Figure 1. Our preprocessing functionality $\mathcal{F}_{\mathsf{pre}}$ allows parties to sample DPF keys (see review of DPFs in Section 2.3) and secret-shared uniform permutations. $\mathcal{F}_{\mathsf{pre}}$ can be instantiated in the 2PC setting in the OT-hybrid model given black-box access to a PRG [Ds17]. Preprocessing a permutation of size n can be achieved by a generic OT-based 2PC protocol for Boolean circuits. This protocol consumes $O(n \lg n)$ OTs; we explain in more detail below. Thus, the entire preprocessing protocol only requires OT and a black-box PRG, and OT can be made efficient with the help of efficient OT extension, e.g. [IKNP03, BCG+19, YWL+20, Roy22].

Our goal is to securely instantiate the ideal functionality $\mathcal{F}_{\mathsf{sam}}$; see Figure 2. In short, $\mathcal{F}_{\mathsf{sam}}$ allows parties to read/write a secret-shared memory, with restrictions placed on reads. Formally, the parties may (1) initialize a fresh SAM, (2) allocate memory addresses in that SAM, (3) read an allocated address, and (4) write to an allocated address. In our functionality, writing the same address accumulates an XOR of all

Functionality $\mathcal{F}_{\texttt{pre}}$:

- **Preprocess distributed point function:** Upon receiving $(\texttt{dpf}, \lg n, m)$ from both parties:
  - Uniformly sample $\alpha \in_{\$} \{0,1\}^{\lg n}, \beta \in_{\$} \{0,1\}^m$.
  - Call $(k_0, k_1) \leftarrow \texttt{DPF.Gen}(1^\lambda, \alpha, \beta)$. Send k_0 to P_0, and send k_1 to P_1.
  - Distribute additive shares $[\![\alpha]\!], [\![\beta]\!]$ to the two parties. Shares of α, β are chosen arbitrarily; if a party is corrupted, then the adversary chooses output shares for the corrupted party, and $\mathcal{F}_{\texttt{pre}}$ sets honest party shares accordingly.
- **Preprocess permutation:** Upon receiving $(\texttt{permute}, n)$, uniformly sample a permutation π on n elements, where π is encoded as a bitstring of length $n \lg n$. Sample a uniform secret sharing $[\![\pi]\!]$, and send one share to each party. Shares are chosen arbitrarily; if a party is corrupted, then the adversary chooses output shares for the corrupted party, and $\mathcal{F}_{\texttt{pre}}$ sets honest party shares accordingly.

Figure 1: We build secure two-party execution of SAM programs on the above preprocessing functionality $\mathcal{F}_{\texttt{pre}}$. In the 2PC setting, DPFs can be instantiated from OT and a black-box PRG [Ds17], and permutations can be instantiated via a generic Boolean 2PC protocol and an invocation of a permutation network [Wak68].

written values at the specified address. Reading the same address twice is considered an error, and we only consider SAM programs that do not cause errors. We review the SAM model further in Section 3.

In our handling of allocate/read, we allow the semi-honest adversary to choose its output secret shares, and the functionality sets honest party shares accordingly. Our protocol can securely achieve this notion because these shares are already stored locally at the parties; see Appendix C for the full proof of security.

Preprocessing a shared permutation π In our handling of SAM, the parties allocate memory addresses according to a secret permutation π; it is crucial that neither party knows π. This permutation is data-independent, so it can be computed in the offline phase. Our preprocessing functionality $\mathcal{F}_{\texttt{pre}}$ allows the parties to sample such a secret-shared permutation.

It is well known that in the 2PC setting, a secret-shared random permutation π can be computed via an invocation of a Waksman or Beneš *permutation network* [Wak68, Ben64]. Namely, the parties compute a permutation $\pi = \pi_0 \circ \pi_1$ where party P_i knows π_i, but not π_{1-i}. When subpermutations π_i are uniform, so is π, and neither party knows π. To compute π, P_1 locally samples a string that encodes π_1, then the parties use an OT-based 2PC protocol to implement a permutation network for π_0 programmed by P_0; the output of this 2PC protocol is a secret sharing of the string that encodes $\pi_0 \circ \pi_1$. A Waksman network on n inputs can be computed by a circuit with $n \lg n - n + 1$ *swap* gates on $(\lg n)$-bit words. Thus, this step can be handled by $O(n \lg n)$ 1-out-of-2 OTs on strings of length $\lg n$.

2.3 Distributed Point Functions

We review the concept of a distributed point function (DPF) [GI14, BGI16], which is central to our approach.

A DPF is a function secret sharing scheme (FSS [BGI15]) for the class of point functions:

Definition 1 (Point Function)**.** *Let $\alpha \in \{0,1\}^{\lg n}$ denote an index and let $\beta \in \{0,1\}^m$ denote a value. The **point function that stores β at position α** is a function $\texttt{Point}(\alpha, \beta, \cdot) : \{0,1\}^{\lg n} \to \{0,1\}^m$ defined as follows:*

$$\texttt{Point}(\alpha, \beta, x) \triangleq \begin{cases} \beta & \text{if } x = \alpha \\ 0 & \text{otherwise} \end{cases}$$

For simplicity, we consider a restriction of DPFs with Boolean domain/range. More generally, DPFs can range over arbitrary finite groups, but we do not need this flexibility for our approach.

Definition 2 (Distributed Point Function (DPF))**.** *A function secret sharing scheme for two-party **distributed point functions** is a pair of algorithms $(\texttt{Gen}, \texttt{Eval})$, defined as follows:*

Functionality $\mathcal{F}_{\texttt{sam}}$:

- **Parameters:** $\mathcal{F}_{\texttt{sam}}$ is parameterized by word-size w and array-size n.
- **Initialization:** At initialization, $\mathcal{F}_{\texttt{sam}}$ creates a fresh size-n array A storing w-bit words. Each array slot $A_\alpha \leftarrow 0^w$ is initialized as the all-zeros string.
- **Commands:** $\mathcal{F}_{\texttt{sam}}$ responds to the following commands:
  - **Allocate:** Upon receiving `Alloc` from both parties, uniformly sample a previously-unallocated address $\alpha \in [n]$, and respond with shares $[\![\alpha]\!]$. Shares are chosen arbitrarily; if a party is corrupted, then the adversary chooses the output share for the corrupted party, and $\mathcal{F}_{\texttt{sam}}$ sets the honest party's share such that shares sum to α. If $\mathcal{F}_{\texttt{sam}}$ already received n commands `Alloc`, then this command is considered **invalid**.
  - **Write:** Upon receiving from the parties $(\texttt{Write}, [\![\alpha]\!], [\![\beta]\!])$ for $\beta \in \{0,1\}^w$, set $A_\alpha \leftarrow A_\alpha \oplus \beta$.
  - **Read:** Upon receiving from the parties $(\texttt{Read}, [\![\alpha]\!])$, look up $\beta \leftarrow A_\alpha$, send α to the adversary, and respond with shares $[\![\beta]\!]$. Shares of β are chosen arbitrarily; if a party is corrupted, then the adversary chooses the output share for the corrupted party, and $\mathcal{F}_{\texttt{sam}}$ sets the honest party's share such that shares sum to β. If $\mathcal{F}_{\texttt{sam}}$ already received a command $(\texttt{Read}, [\![\alpha]\!])$ for the same address α, then this command is considered **invalid**.

A sequence of $\mathcal{F}_{\texttt{sam}}$ commands is considered **valid** if no command in the sequence is **invalid**.

Figure 2: Our target two-party SAM functionality $\mathcal{F}_{\texttt{sam}}$ implements a secret-shared array where each array slot may be read at most once. Repeatedly writing to an array slot accumulates the writes via XOR.

- $\texttt{Gen}(1^\lambda, \alpha, \beta)$ *takes as input* $\alpha \in \{0,1\}^{\lg n}, \beta \in \{0,1\}^m$*;* `Gen` *outputs a pair of keys* k_0, k_1*.*
- $\texttt{Eval}(k, x)$ *takes as input a key generated by* `Gen` *and an index* $x \in \{0,1\}^{\lg n}$*. It outputs* m *bits.*

The scheme is ***correct*** *if evaluations* $\texttt{Eval}(k_0, x), \texttt{Eval}(k_1, x)$ *yield XOR shares of the specified point function. Let* $(k_0, k_1) \leftarrow \texttt{Gen}(1^\lambda, \alpha, \beta)$*. Then the following must hold:*

$$\texttt{Eval}(k_0, x) \oplus \texttt{Eval}(k_1, x) = \texttt{Point}(\alpha, \beta, x)$$

The scheme is ***private*** *if there exists a simulator* $\mathcal{S}_{\texttt{dpf}}$ *such that for every* $b \in \{0,1\}$ *and for every polynomial-size point function description* $\alpha \in \{0,1\}^{\lg n}, \beta \in \{0,1\}^m$*, the following computational indistinguishability (in* λ*) holds:*

$$\{\ k_b \quad \textit{where} \quad (k_0, k_1) \leftarrow \texttt{Gen}(1^\lambda, \alpha, \beta)\ \} \approx \{\ \mathcal{S}_{\texttt{dpf}}(1^\lambda, b)\ \}$$

While efficient constructions for generating DPF keys exist, such constructions require a dealer [BGI16]. In the 2PC setting, such a dealer is not available, so we need to instantiate such a dealer by means of a protocol; [Ds17] gave a suitable construction:

Fact 1 (2PC DPF Protocol [Ds17])**.** *Assuming PRGs exist and in the OT-hybrid model, there exists a 2PC protocol for generating DPF keys from* $\{0,1\}^{\lg n}$ *to* $\{0,1\}^m$ *that consumes* $O(\lg n)$ *1-out-of-2 OTs over* λ*-bit strings, and* $O(m)$ *additional bits of communication. The parties expend* $\tilde{O}(nm)$ *local computation.*

Derandomizing Distributed Point Functions Our protocol generates DPF keys in the offline phase. These preprocessed keys are for a randomly chosen point function, and in the online phase we derandomize them. Here, we explain this derandomization.

Fact 2 (Shifting DPFs)**.** *There exists a local procedure* `Shift` *that satisfies the following correctness condition. Let* $\alpha, \alpha' \in \{0,1\}^{\lg n}$*, let* $\beta, \beta' \in \{0,1\}^m$*, and let* $(k_0, k_1) \leftarrow \texttt{Gen}(1^\lambda, \alpha, (\beta, 1))$*. Then:*

$$\texttt{Shift}(k_0, \alpha', \beta', x) \oplus \texttt{Shift}(k_1, \alpha', \beta', x) = \texttt{Point}(\alpha \oplus \alpha', \beta \oplus \beta', x)$$

Here, the pair $(\beta, 1)$ *denotes the* $(m+1)$*-bit concatenation of* β *with a one at the end.*

This follows by evaluating the preprocessed DPF at $x \oplus \alpha'$ and using the extra tag bit to add β' only at the selected point. For completeness, we explain `shift` in detail in Appendix B.

The crucial takeaway from Fact 2 is that if we wish to construct a DPF with specified evaluation point α and value β, it suffices to preprocess a *random* DPF with uniform evaluation point α' and uniform value β'. Given secret shares of $\alpha, \alpha', \beta, \beta'$, the parties can compute secret shares of $\alpha \oplus \alpha'$ and $\beta \oplus \beta'$, reveal these differences to themselves (which is secure because α' and β' are uniform), then locally call `Shift` on their respective keys.

Improving DPF computation overhead As discussed so far, the method by which two parties evaluate a DPF on its full size-n domain is by calling `Eval` (or `Shift`) n times. Existing PRG-based DPFs offer a better procedure for this "full domain" evaluation, which reduces the total number of PRG calls by factor $O(\lg n)$. In short, PRG-based DPFs involve using the PRG to induce a GGM-style binary tree [GGM84], and evaluation involves walking a path through that tree. A full tree walk on a tree of size n is $O(\lg n)$ times more efficient than n times traversing a path through the same tree; see [BGI16] for details. Our implementation applies this optimization.

3 Review of the Single Access Machine Model

General-purpose ORAM often incurs high asymptotic and/or concrete overhead. Thus, it is interesting to consider designing *special-purpose* oblivious handling of particular algorithms and data structures [GGH$^+$13, ZE13, WNL$^+$14, KS14, LWN$^+$15, NWI$^+$15, Shi20, JLS21]. Seeking to unify and improve oblivious algorithms and data structures, [AHR24] introduced SAM as a weakening of the RAM model of computation. Because SAM is more restrictive on the memory-manipulating program than RAM, SAM-based memory can be implemented more efficiently. In particular, [AHR24] observed that certain ORAM techniques naturally achieve SAM. The authors took off-the-shelf ORAM constructions Path ORAM [SvDS$^+$13] and Circuit ORAM [WCS15] and transformed them to achieve a notion of Oblivious SAM (OSAM). This transformation involved *removing* a large component—the so-called position map—of those constructions, reducing their asymptotic complexity.

Our work broadens the set of techniques that naturally implement SAM. Note that our approach to implementing SAM is entirely different than that of [AHR24], as it leverages DPFs to implement SAM.

Usefulness of SAM Limitations placed on SAM programs may at first glance seem overly restrictive, but [AHR24] showed that the SAM model is surprisingly expressive. In short, any pointer machine program (i.e., where the program can read/write pointers, but cannot, e.g., add a constant to a pointer) can be compiled to a SAM program. The cost of this compilation is somewhat nuanced: each time the pointer program dereferences a memory cell with d incoming pointers, amortized $O(\lg d)$ SAM operations are consumed.

In many cases, the authors show it is possible to keep the number of pointers to any memory cell to a constant, thus achieving *no* asymptotic blow-up. For example, stacks, queues, doubly-linked lists, trees, and even low-degree graphs are efficiently instantiable in the SAM model. Indeed, even graph traversals over arbitrary-degree graphs can be achieved without asymptotic blow-up [AHR24]. [PAR$^+$26] improved constant factors incurred in the SAM-based handling of pointer machine programs, and [AH25] further extended the set of programs achievable from SAM, showing that many interesting algorithms on strings and even programs written in purely functional programming languages efficiently compile to a SAM program. Thus, our new SAM protocol implies new results for the cost of running a variety of algorithms inside 2PC.

As one example, consider performing a graph traversal with depth-first or breadth-first search (DFS or BFS). DFS/BFS on a graph of size n (with word size $w = \Theta(\lg n)$) can be achieved by $O(n)$ SAM operations [AHR24], so our protocol implies a privacy-preserving algorithm for graph traversal that incurs only $O(n \lg n)$ bits of online communication. Thus, the online communication complexity of the 2PC protocol matches the total number of bits touched in a cleartext execution of the same algorithm. Contrast this with standard ORAM-based execution of the same algorithm using Circuit ORAM [WCS15], where online cost would be $O(n \lg^2 n\lambda)$ bits.

Of course, while we implement SAM, one might also be interested in how our protocol compares with prior work on RAM. Fortunately, any RAM program can be compiled to a SAM program with small blow-up;

we discuss shortly.

Managing SAM addresses A SAM program may allocate a memory address, then read/write[4] that address later on. But in the SAM model, the client program should run in constant space and offload memory storage to the single access memory, so how can the client program remember addresses it allocated earlier? The answer is that the client program may allocate an address, then save that address into SAM memory such that it can be read back and used later. It is in this way that the client program can build up tree-like and graph-like structures: a node in a binary tree is encoded as a pair of addresses to its children, and this pair can be saved as a memory word; see next for more details.

3.1 How to Implement RAM with SAM

As already discussed, the SAM model can be used to implement programs that manipulate trees. Because of this, we can use SAM operations to implement random access memory [AHR24]. We for completeness revisit the technique.

The main idea is to store the content of RAM in a balanced trie where the i-th trie leaf stores the content of the i-th RAM slot. Reading/writing the RAM involves walking to the appropriate leaf node. In many ORAMs, the client maintains a *position map*, which maps array indices to positions in the ORAM structure. Typically, this map is implemented recursively. The above technique of implementing RAM as a tree essentially instantiates the recursive position map as a single tree structure. This arrangement places data words and position-map metadata into a single SAM memory.

More precisely, each RAM access is implemented via a SAM algorithm as follows. To begin, the SAM program (aka, the 2PC circuit) holds a SAM address `root`, which is the root of the trie implementing RAM, as well as a RAM address α that it wishes to access. The algorithm's goal is simply to walk through to the α-th leaf of the trie, and then to read and/or overwrite its content. The challenge is that because memory is single-access, the algorithm must rebuild the RAM trie as it proceeds. Roughly, this involves iteratively walking from parent node to child node, at each step preallocating child write-back addresses, and writing back a fresh parent node before proceeding to the next step of the iteration. Algorithm 1 provides the basic RAM-to-SAM reduction. Note that this algorithm involves two helper procedures `MuxRead` and `MuxWrite`. In the 2PC setting, these are essentially oblivious swap operations and can be achieved by a small constant number of OTs.

Using large words to reduce RAM overhead Here, we discuss and work out the details of an alternate arrangement of SAM-based RAM, which was not discussed in [AHR24]. Our implementation of SAM-based RAM uses this alternate arrangement for increased efficiency. We leave this discussion in preliminaries because the techniques are either known (in the setting of position maps) or straightforward.

As an alternative to storing the position map and data words in a single SAM, one can store each metadata level of the RAM trie in its own SAM, and store the data leaves in a final data SAM. Namely, if the RAM trie has h levels, then there are h SAMs, one per level. To support this, we modify Algorithm 1 such that each iteration of the tree walk interacts with a distinct SAM. Note that interleaving operations on multiple SAMs does not pose a security problem because the RAM trie is balanced; for each RAM access, the parties observe each SAM queried in turn. The advantage of storing distinct levels of the tree in distinct SAMs is that we can choose different word sizes for each SAM. This affords two main advantages: First, rather than maintaining a binary tree, we may instead use a tree with variable fan-out at each level, packing large numbers of SAM addresses into each metadata word, especially towards the root where addresses are small. Second, if the RAM itself stores large words, e.g. if $w = \Omega(\lg^2 n)$, then the online cost of accessing the last SAM will asymptotically erase the online cost of all other SAM accesses.

For instance, if the RAM trie has depth $O(\lg n)$, and if each non-leaf SAM stores words of size $w' = O(\lg n)$, then the total number of bits read from SAM to traverse a path is only $O(\lg^2 n)$, asymptotically matched or exceeded by the size of a data word. In the literature and in the context of standard ORAM position maps, this is sometimes called the big block/little block trick; see e.g. [SvDS+13, WCS15].

[4] A SAM program may read an address that has not been written, in which case the SAM must return some default value. This capability is important for using SAM to handle DAG-like structures [AHR24]. Our 2PC SAM returns the all zeros string when reading an unwritten value.

Algorithm 1 Implementing RAM access via SAM operations

Input: A root of the RAM trie $\texttt{root}$, a logical address α, an indicator bit rw that indicates if this is a read or a write (0 denotes read, 1 denotes write), and a value $\beta_{\texttt{new}}$ to write back if this is indeed a write.

Output: A new root of the RAM trie $\texttt{root}'$ and the value $\beta_{\texttt{old}}$ read from RAM.

$\texttt{root}' \leftarrow \texttt{SAM.Alloc}$
$\texttt{parent} \leftarrow \texttt{root}$
$\texttt{parent}' \leftarrow \texttt{root}'$
for $i \in \{0, ..., h-1\}$ **do** ▷ $h = \lg n$ is the RAM trie's height
  $\texttt{content} \leftarrow \texttt{SAM.Read}(\texttt{parent})$
  $(\texttt{child}_0, \texttt{child}_1) \leftarrow \texttt{content}$
  $\texttt{child}_{\alpha_i} \leftarrow \texttt{MuxRead}(\alpha_i, \texttt{child}_0, \texttt{child}_1)$
  $\texttt{child}' \leftarrow \texttt{SAM.Alloc}$
  $\texttt{content}' \leftarrow \texttt{MuxWrite}(\alpha_i, \texttt{child}', \texttt{content})$
  $\texttt{SAM.Write}(\texttt{parent}', \texttt{content}')$
  $\texttt{parent} \leftarrow \texttt{child}_{\alpha_i}$
  $\texttt{parent}' \leftarrow \texttt{child}'$
end for
$\beta_{\texttt{old}} \leftarrow \texttt{SAM.Read}(\texttt{parent})$
$\texttt{SAM.Write}(\texttt{parent}', \texttt{MuxRead}(rw, \beta_{\texttt{old}}, \beta_{\texttt{new}}))$
return $(\texttt{root}', \beta_{\texttt{old}})$

In our approach, reading/writing SAM incurs constant rounds of communication, so when implementing RAM we incur a number of rounds that scales with the length of paths through the RAM trie. So if we choose to implement RAM as a binary trie, then we use $O(\lg n)$ rounds of communication per access.

If RAM words are large, we can also improve round complexity, without changing the asymptotic communication complexity. Namely, we can choose a parameter $2 \leq B \leq n$, and set this as the degree of our RAM trie. This reduces the height of the RAM trie to only $O(\lg n / \lg B)$. If $B = O(w / \lg^2 n)$, then this incurs no increase to the asymptotic communication complexity. This reduction in height in turn decreases the number of needed SAM operations by factor $\lg B$. If RAM words are *very* large, this can even reduce round complexity to a constant. In particular, suppose that $w = \Omega(\lambda \lg^2 n)$, and suppose that $n = O(\lambda^c)$ is bounded by some fixed polynomial in the security parameter, as is relatively standard. In this case, we can set $B = \lambda$ without blowing up communication complexity, thus achieving a tree of depth $O(\lg n / \lg \lambda) = O(1)$. Thus, for very large word size, our approach achieves 2PC-RAM where each RAM access incurs constant communication overhead and constant rounds.

Supporting long-running RAM programs If our two parties wish to securely run a RAM program where the number of RAM accesses is upper-bounded by some known quantity T and where T is not too large, such as $T = O(n)$, the SAM-based RAM we have discussed so far is satisfactory. Namely, the parties simply instantiate working SAM structures with size sufficient to support the T queries.

However, if T is not known, or if it is large, the above arrangement is problematic. For large T, the SAM-based RAM runs into efficiency problems, as its costs will scale with T, not the number of RAM slots n. Worse, if T is not known, it is unclear how to proceed.

The strategy for addressing these problems is relatively straightforward: periodically reset the SAM on each trie level i. We fix some constant factor c (in our implementation, we choose $c = 3$); for each level i with n_i nodes, we *reset* that level every $N_i = c \cdot n_i$ RAM accesses. Resetting involves reading the contents of all nodes on level i into an array and then initializing a fresh level i. This ensures that the cost of each SAM operation on level i scales with N_i, not with T, and we can choose N_i without any knowledge of T and sensibly proceed. While the main idea is straightforward, some care is needed to avoid asymptotic blow-up in cost. Appendix A describes this in detail. We note that this epochal reset strategy is analogous to the rebuild strategy described in [HKO22], though their underlying oblivious data structure mechanism is different.

3.2 How to Implement Pointers with SAM

Although our emphasis is RAM, our support for SAM allows us to take advantage of other SAM-based capabilities. The main such capability is support for *pointer machine programs* [Sch80, KU58]. Namely, SAM can indirectly—but asymptotically-efficiently—support programs consisting of objects storing pointers to other objects.

Indeed, Section 3.1 already explained a major part of this capability: we can represent objects with pointers by storing those pointers as SAM addresses. Thus, when we dereference an object from SAM memory, we can dereference the pointers obtained, to move from object to object.

However, as we have described so far, this would only support *tree-like* structures, not the graph-like structures enabled in the pointer machine model. The key ingredient that must be added is a way to arrange for two objects a and b to each hold pointers to some third object c. This is a challenge. Indeed, if we simply give a and b the SAM address for c, then when a dereferences c, the single access restriction means that b's pointer to c is *invalidated*, which is problematic for building up useful graph-like structures.

Both [AHR24] and [PAR$^+$26] demonstrate that SAM can be used to solve this *pointer sharing* problem. We recall [PAR$^+$26]'s solution because it is simpler and more efficient. (Note that the [PAR$^+$26] solution requires a SAM with support for repeated writes to the same location; [AHR24] did not assume this capability. Our SAM achieves repeated writes.) The idea is to have a and b hold the address not only of c, but also of one another. Namely a holds the address of b and vice versa. These cross addresses allow the SAM program to repair destroyed links in the pointer graph as it proceeds. Namely, a SAM program holding a pointer to a in local memory can use a strategy reminiscent of Algorithm 1 to both (1) read the content of c and (2) rebuild the SAM structure such that the pointer from b to c is valid.

More technically, the SAM program holding the address of a can proceed as follows:

- Read the address of a. Note that this invalidates b's copy of the address of a. The content of a—namely the addresses of both b and c—is now in local memory.
- Read the address of c. Again, note that this invalidates b's copy of the address of c.
- The desired data stored in c is now in local memory, and the SAM program can use it as appropriate. The remaining work is to restore the pointer structure that was destroyed.
- Allocate two new addresses for new objects a' and c', as replacements for a and c.
- Write the (possibly-updated by the SAM program) content of c to the address of c'.
- Overwrite[5] the content of b with the addresses of a' and c'.
- Overwrite the updated object a' to contain the addresses of b and c'.

With this done, the SAM program successfully retrieved the content of c *and* the pointer structure is restored to its original state: a' and b each have pointers to the updated c' (and to each other). By generalizing this basic idea, it is possible to not only handle this simple fan-in two case, but *arbitrary* graph-like structures which can be non-destructively traversed; see [AHR24, PAR$^+$26] for details.

We use SAM's support for pointer machine programs to implement graphs and, in particular, a graph depth-first search.

4 Technical Approach

For a brief informal explanation of our approach, see Section 1.2. Figure 3 lists our formal protocol. Our protocol handles each SAM operation as follows:

- $\alpha \leftarrow$ `Alloc`: Parties allocate memory addresses in a secret, random order. Namely, the sequence of allocations is described by a uniform permutation π, and each i-th memory address α_i is π_i. π is sampled during preprocessing.

[5] Recall that our SAM's repeated write aggregates an XOR operation. We can achieve the task of overwriting (a, c) by (a', c') by writing the difference $(a \oplus a', c \oplus c')$.

Protocol $\Pi_{\texttt{sam}}$:

- **Initialize:** Parties use $\mathcal{F}_{\texttt{pre}}$ to preprocess a size-n uniform, secret-shared permutation $[\![\pi]\!]$.

  Parties also use $\mathcal{F}_{\texttt{pre}}$ to preprocess n uniform DPFs from $\{0,1\}^{\lg n}$ to $\{0,1\}^{w+1}$. w bits of the range are for storing a data value, 1 bit is for the purpose of derandomization (see Section 2.3). In particular, for each $i \in [n]$ the preprocessing functionality (1) samples uniform $\alpha'_i \in \{0,1\}^{\lg n}$, (2) samples uniform $\beta'_i \in \{0,1\}^w$, and (3) generates DPF keys as follows:

  $$k_{0,i}, k_{1,i} \leftarrow \texttt{Gen}(1^\lambda, \alpha'_i, (\beta'_i, 1))$$

  $\mathcal{F}_{\texttt{pre}}$ distributes $[\![\alpha'_i]\!]$ and $[\![\beta'_i]\!]$ to the parties, sends $k_{0,i}$ to P_0, and sends $k_{1,i}$ to P_1.

  Parties initialize a secret-shared memory state $[\![\sigma]\!]$. Namely, σ is a length-n all-zeros array, where each σ_i is w bits long.

- **Allocate:** Let this be the i-th allocation. Parties output secret shares of the i-th entry of the preprocessed permutation: $[\![\alpha]\!] \triangleq [\![\pi_i]\!]$.

- **Write:** Let this be the i-th write. Parties input secret-shared address $[\![\alpha]\!]$ and value $[\![\beta]\!]$. Parties fetch their i-th preprocessed DPF corresponding to random index α'_i and value β'_i:

  $$k_{0,i}, k_{1,i}, [\![\alpha'_i]\!], [\![\beta'_i]\!]$$

  Parties locally compute $[\![\alpha \oplus \alpha'_i]\!]$ and $[\![\beta \oplus \beta'_i]\!]$, then exchange shares to reveal $\alpha \oplus \alpha'_i$ and $\beta \oplus \beta'_i$. This exchange consumes 1 round and $2\lg n + 2w$ bits. Recall from Section 2.3 that parties can derandomize their DPF shares at each index $j \in \{0,1\}^{\lg n}$ as follows:

  $$(\texttt{Shift}(k_{0,i}, \alpha \oplus \alpha'_i, \beta \oplus \beta'_i, j), \texttt{Shift}(k_{1,i}, \alpha \oplus \alpha'_i, \beta \oplus \beta'_i, j)) = [\![\texttt{Point}(\alpha, \beta, j)]\!]$$

  Parties use their derandomized DPF to update each index j of the state:

  $$[\![\sigma_j]\!] \leftarrow [\![\sigma_j]\!] \oplus [\![\texttt{Point}(\alpha, \beta, j)]\!]$$

  This correctly updates index α to hold value β, and it has no effect on other indices.

- **Read:** Parties input $[\![\alpha]\!]$. They exchange shares, revealing α in the clear. This exchange consumes 1 round and $2\lg n$ bits. Parties look up index α of the secret-shared array and output $[\![\sigma_\alpha]\!]$.

Figure 3: $\Pi_{\texttt{sam}}$ realizes the $\mathcal{F}_{\texttt{sam}}$ functionality (Figure 2) in the $\mathcal{F}_{\texttt{pre}}$-hybrid model (Figure 1).

- $\texttt{Write}(\alpha, \beta)$: Parties use a DPF to construct a secret-shared vector where index α holds β and all other indexes hold 0; they element-wise XOR this vector into the memory, writing β to address α.
- $\beta \leftarrow \texttt{Read}(\alpha)$: Parties exchange shares to reveal α. They locally read respective shares of memory address α.

The approach is correct by construction: each write saves element β to address α by the correctness of DPFs, so reading address α in the clear correctly fetches the element written there. The only challenge in terms of security is that on each read the parties reveal an address to themselves. But when SAM addresses are used correctly by a calling protocol, these revelations can be simulated because (1) addresses are uniformly permuted and (2) the SAM model guarantees each address is read at most once. Thus, each revealed address appears uniformly random (without replacement). See Appendix C for our security proof.

We note that our SAM functionality places the burden of managing SAM addresses on the calling protocol. Formally, this manifests in the fact that $\mathcal{F}_{\texttt{sam}}$ reveals read addresses α to the adversary. Roughly, as long as the calling protocol uses SAM addresses as intended, it is straightforward to prove security.

In more detail, our SAM functionality cannot enforce that a calling protocol does not use SAM addresses in a way that breaks security. For instance, a calling protocol might ignore output of the alloc procedure and

make up its own addresses, or it might mangle SAM addresses by running procedures on their bits, and so on. As long as a calling procedure (1) allocates all SAM addresses by calling alloc and (2) does not change the content of an address or conditionally branch its behavior depending on the bits of an address, then the outer protocol's usage of SAM is straightforwardly secure: Every allocated SAM address is secret-shared and uniform, and it is revealed at most once when it is eventually read. Thus, while the adversary in the outer protocol is given addresses α on each call to read, these are trivially simulated by uniformly sampling without replacement.

Cost Accounting Consider a sequence of $O(n)$ SAM operations on a memory of size n and with w-bit words.

- **Offline communication.** The bottleneck in offline communication is from preprocessing DPFs, which incurs $O(w + \lambda \lg n)$ bits per `Write`.
- **Offline rounds.** All DPFs can be preprocessed in parallel, and Waksman's network has depth $O(\lg n)$, so the total round complexity of the offline phase is only $O(\lg n)$ rounds.
- **Offline computation.** [Ds17]'s DPF protocol runs in $\tilde{O}(wn)$ time, which is the offline computation bottleneck.
- **Online communication.** The most expensive operation is `Write`, which requires derandomization of a DPF, incurring $2w + 2 \lg n$ bits of communication.
- **Online rounds.** Each operation type finishes within 1 round.
- **Online computation.** Derandomizing and evaluating a DPF at all indices takes $\tilde{O}(wn)$ time.

In Appendix C, we prove the following theorem. In short, that proof is a relatively straightforward delegation to the DPF simulator.

Theorem 1. *Assuming the parties issue a sequence of **valid** SAM commands (Figure 2),* $\Pi_{\texttt{sam}}$ *realizes* $\mathcal{F}_{\texttt{sam}}$ *with semi-honest security in the* $\mathcal{F}_{\texttt{pre}}$*-hybrid model.*

5 Experimental Evaluation

This section presents evaluation results for a concrete implementation of our protocol. Our emphasis is on using our SAM to construct RAM, but we also explore how to use SAM to implement a representative SAM-friendly graph algorithm—namely depth-first search—directly. We start by explaining our implementation and details of our experimental setup and benchmarks. Then, we discuss results and how they compare to prior 2PC RAM protocols. In our comparison to prior work, we focus on Duoram [VHG23] and Floram [Ds17], as these are the most closely related works in terms of setting and techniques.

5.1 Implementation Details

Prototype Description We prototyped our end-to-end protocol in C++. To implement our preprocessing, we use the EMP Toolkit's support for the Ferret OT extension protocol [WMK16, YWL+20]. To preprocess DPF keys, we instantiated Floram's distributed key generation protocol [Ds17] via the above OT backend, using hardware-accelerated AES to instantiate the necessary PRG. We leverage parallelism in our implementation of DPF generation/evaluation.

Our DPF generation uses a pebbling strategy to reduce space. The distributed key-generation protocol of Floram processes the DPF tree level by level. Rather than materializing an entire level of the DPF tree, our implementation recomputes the needed nodes by pebbling the tree, aggregating as it goes the two OT inputs needed for the current level. This keeps the working space logarithmic in the DPF domain size.

Our implementation provides the interface needed to implement 2PC for arbitrary SAM programs. It prepares necessary correlated randomness (including DPF keys, secret-shared permutations, and multiplication triples needed for low-level operations) in a low-round offline phase and then expends those in the online phase.

Our prototype includes instrumentation to measure (1) random OT consumption, (2) communication consumption, (3) round complexity, (4) AES evaluations, and (5) wall-clock runtime. We measure cost separately for the offline phase and online phase (except in the case of OTs, all OTs are consumed offline). Note that the generation of random OTs is not counted towards our communication consumption; instead, we note this separately. This allows us to separate the cost of our protocol on top of random OT generation. The communication cost of generating random OTs will depend on the underlying protocol; in the case of Ferret OT, each random OT can cost amortized less than one bit [YWL+20].

We run our protocol in two modes. In mode (1) we run the protocol end to end, where two distinct servers communicate over a network and complete a secure 2PC protocol. In mode (2), the implementation executes the same control flow as the real protocol and records exact counts of protocol events: OTs, messages, rounds, and AES calls. Thus, these measurements are precise (and validated against mode (1) for small instances), not asymptotic. The second mode is used due to limitations of our hardware setup, where we could not run large versions of memory for large numbers of accesses in a reasonable amount of time (we never observed memory being the bottleneck). Our wall-clock time benchmarks are made in mode (1). We observe that this limitation is consistent with prior work, e.g. [VHG23]. For mode (1), we ran two parties on an Amazon EC2 `c7i.2xlarge` instance with 16GiB RAM and supporting 8 virtual threads. We allocate 4 threads to each party, and run the protocol over the local network.

We exercise our SAM implementation on two main benchmarks:

RAM Benchmark Our main considered benchmark uses SAM to implement RAM, as described in Section 3.1. This benchmark showcases the performance of our approach, especially its low communication complexity.

In implementing this benchmark, we use optimizations described in Section 3.1, including the long-running RAM program optimization; for this, we set the reset constant to $c = 3$. That is, for a RAM of size n, we perform $3n$ RAM accesses before paying for a reset step, and the largest underlying SAM is of size $4n$. We run for $T = 6n$ RAM accesses to ensure reset machinery is fully exercised.

The RAM program microbenchmark itself is very simple: the parties write n RAM addresses $0, ..., n-1$, then read them back in reverse order. This is repeated until T accesses are consumed.

Graph DFS Benchmark Here, we more fully exercise our protocol's support for SAM by implementing graph depth-first search. This benchmark allows us to explore benefits enabled by our protocol's support for low-level SAM. We compare two implementations: a RAM-based DFS and a SAM-based one. To support the SAM-based DFS, we implement the SAM-based pointer machine logic as described in [PAR+26].

In both cases, the directed graph is represented sparsely via pointers. Namely, each vertex maintains pointers to its successors. To implement the DFS algorithm, we maintain a stack of successors. The algorithm itself is standard: at each step it (1) pops a vertex pointer from the stack, (2) checks if the vertex has already been visited, (3) if the vertex has not been visited, pushes successor pointers onto the stack, and (4) marks the vertex as visited.

The main difference between the two implementations is in how internal data structures are represented. In the case of RAM, pointers are raw addresses, and the stack is a contiguous block of memory cells; in the case of SAM, pointers are "smart pointers" [AHR24, PAR+26], and the stack is a singly-linked list. Thus, the SAM-based program is substantially more complex, but recall that the corresponding advantage is that SAM itself can be cheaper than RAM.

For the microbenchmark, we use a bounded-degree synthetic graph: each vertex v has outgoing edges to $v + 1 \bmod n$ and $v \oplus (n/2)$. This graph is not intended to model a particular application workload; rather, it gives a compact reproducible instance that exercises the generic pointer-traversal machinery. The graph is stored and traversed using the same pointer representation that would be used for arbitrary sparse graphs. The point of this benchmark is not the particular synthetic graph, but the fact that it leverages the pointer-machine operations needed by generic SAM graph traversals: stack operations, pointer traversal, and pointer repair.

A main challenge of achieving 2PC of DFS via the SAM model is that the SAM program that simulates pointer machine operations is non-trivial; recall discussion of handling two pointers to the same location from Section 3. The challenge here is that to achieve a secure 2PC protocol, the protocol must not leak which step of the pointer machine simulation it is in.

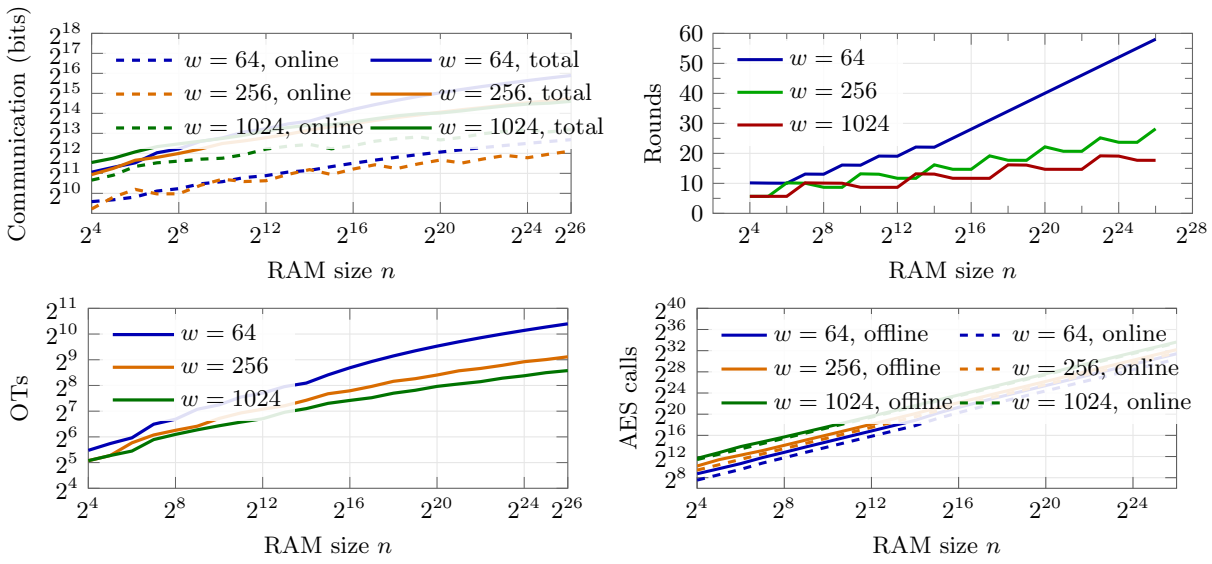

Figure 4: Our SAM-based RAM's resource consumption, amortized over n RAM accesses.

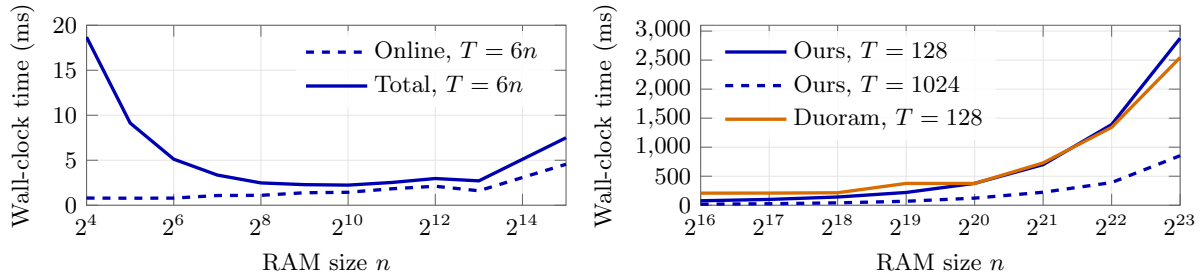

Figure 5: Amortized wall-clock runtime per access of our SAM-based RAM. Word size w is 64 bits. On the left we give amortization over $T = 6n$ accesses. On the right we give amortization over specified T accesses; Duoram's implementation hung for $T = 1024$.

To handle this, we designed an oblivious state machine with 11 distinct states. The machine maintains a modest number of (secret-shared) local registers, and at each step it performs one SAM read and one SAM write. The arguments passed to each operation are computed via operations on secret shares. When an operation is not needed, a dummy one is performed. A dummy read burns a SAM address, which means dummy SAM reads are relatively expensive, since we must allocate a SAM large enough to accommodate all dummies. This *control logic* for the SAM-based DFS is considerably more complex than the corresponding RAM-based DFS.

5.2 Experimental Results

RAM Cost Figure 4 plots the resources consumed by our SAM-based RAM. Communication costs of our protocol are lean, especially in the online phase. The plots also confirm the main bottleneck of our approach: the number of per-access calls to AES, which increases linearly in n.

Following discussion in Section 3.1, we vary word size w between 64 and 1024, observing how cost varies as w increases. This relationship is nuanced. As w increases, the SAM-based trie implementing RAM can support higher fan-out. This makes the trie shallower, and hence the number of rounds and OTs decrease. This *can* also decrease communication consumption, but note there is a trade-off, as each low-level operation in the RAM also has communication cost scaling with w. So while the number of operations decrease, the communication cost of each operation increases. For computation, there is a clear trend: increasing word

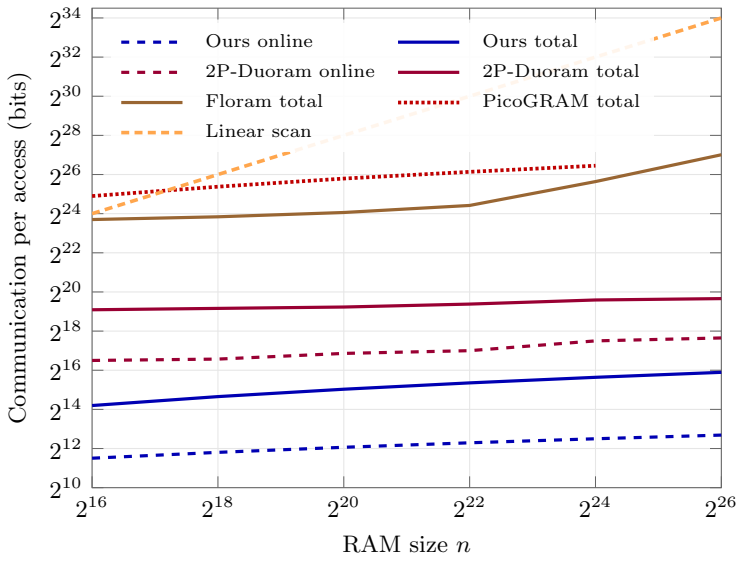

Figure 6: Communication cost of our SAM-based RAM versus 2PC RAMs from prior work. All RAMs use $w = 64$ bits.

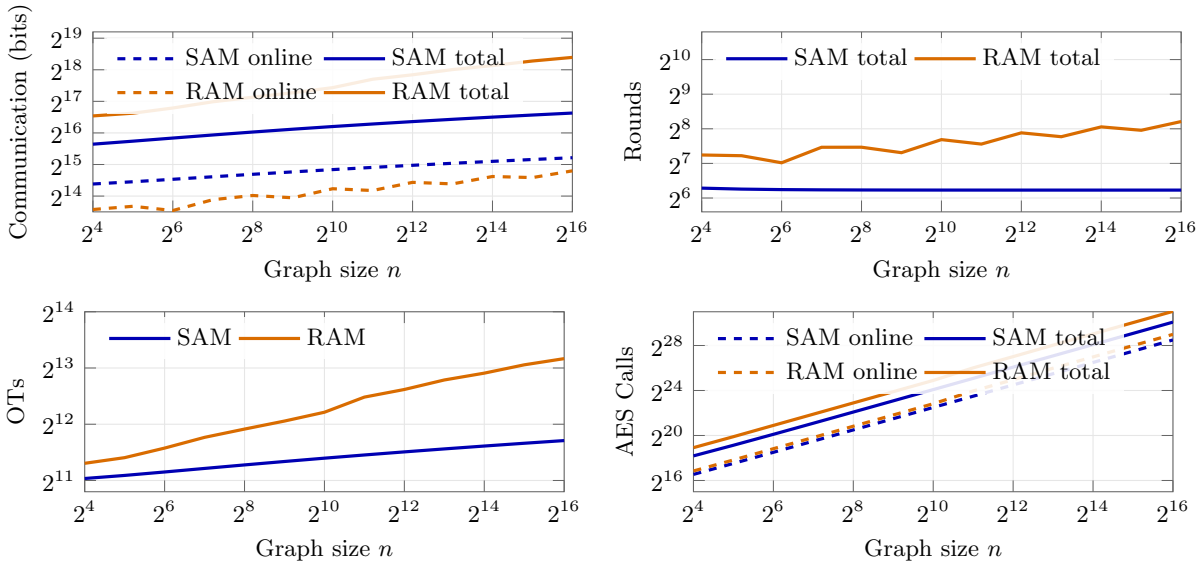

Figure 7: Amortized cost of our DFS experiment, implemented with both SAM-based RAM and SAM-based pointer machine operations. Amortization is over the number of traversed graph vertices n. Both machines use $w = 128$.

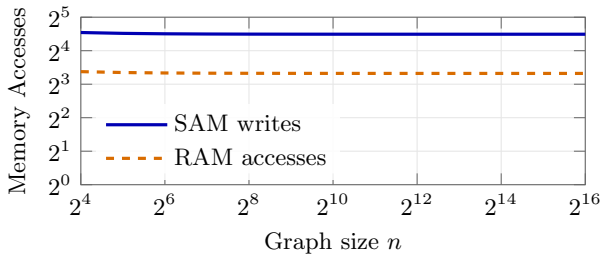

Figure 8: Amortized (over the number of vertices) number of memory accesses needed to complete graph DFS in the RAM model and the SAM model. We record SAM writes as these are the expensive operations in our protocol; the number of SAM reads is slightly lower than the number of SAM writes.

size increases the number of AES calls. This is simply because the leaves of each DPF tree must scale linearly with w.

We observe that there is some noisiness in the round complexity plot. We provide an explanation in Appendix D.

Wall-clock time. Figure 5 plots the wall-clock runtime of our approach. We give two plots; one gives the amortized time to complete 1024 accesses, the other to complete $6n$ accesses, sufficient to fully exercise the RAM's reset schedule. In the second plot, we compare to Duoram's runtime performance. To obtain this data, we used Duoram's publicly available repository[6] and ran it (and for this comparison ours) on our EC2 machine with the same setup.

The first plot shows that for small n, the bottleneck is preprocessing, largely dominated by the cost of preprocessing OTs. As n increases, however, cost starts to shift to the online phase, as AES evaluations become the principal cost. The second plot demonstrates that for $T = 1024$, our per-access cost is 2–3× less than that of Duoram at $T = 128$. Note that for $T = 128$ our performance is similar to that of Duoram; this is because our approach incurs a high one-time preprocessing cost for permutations that is not amortized over the relatively small number of accesses. We tried to evaluate Duoram at $T = 1024$, but the implementation hung.

Communication Cost Comparison We examine the communication consumption of our SAM-based RAM as compared to prior works Floram [Ds17], to Duoram's 2PC approach [VHG23], and to the recent concrete Garbled RAM implementation PicoGRAM [GTRS25]; see Figure 6. The Floram and Duoram numbers are taken from [VHG23], and the Garbled RAM numbers are taken from [GTRS25]. We plot Garbled RAM up to $n = 2^{24}$, as this is the largest available data point they provide. We show our total communication cost when $w = 64$, matching the experiments of prior work.

We include an optimistic linear-scan baseline that reads and writes each memory slot on each access. We charge this baseline $4nw$ bits per access. This is optimistic: it models only the communication needed to exchange shares. Even under this accounting, cost grows linearly with n, and all considered approaches outperform it. Our SAM-based RAM, in particular, outperforms linear scan's communication for all considered RAM sizes $n \geq 16$.

Figure 6 shows how lean our protocol is in terms of communication consumption. Our SAM-based RAM accesses incur significantly less online (resp. total) communication than that of Duoram. In addition, our approach avoids the single-server PIR/FHE machinery used by Duoram's 2PC construction.

Depth First Search Figure 7 plots the resource consumption of our two DFS implementations, one building on a pointer-machine-based memory, the other building on RAM. The main takeaway is that, despite more complex pointer-maintenance logic, direct SAM execution substantially reduces total communication, OTs, rounds, and AES calls. This is because while the program itself is more complex, the memory it builds on is simpler. One place where SAM currently *loses* to RAM is in terms of online communication. This is

[6] `https://git-crysp.uwaterloo.ca/avadapal/duoram`

because the SAM-based solution pays a considerable number of operations on secret-shared data to manage its state machine semantics. Finding techniques by which to more efficiently implement these semantics and to avoid this high cost is a potentially interesting direction.

Figure 8 plots the number of RAM accesses needed for the RAM version, and the number of SAM writes for the pointer-machine version. In both cases, the number of accesses is constant in n.

6 Related Work

Oblivious RAM ORAM is traditionally defined in the client/server model, where the server is a simple read/write memory and performs no other computations [GO96]. In this setting, $\Theta(\lg n)$ communication blow-up is necessary [GO96, LN18] and sufficient [AKL+20].

In other settings, the $\Omega(\lg n)$ lower bound on communication blow-up *does not apply*. For instance, Onion ORAM [DvDF+16] achieved constant blow-up by leveraging *server-side computation* (namely, the server performs homomorphic encryption operations). The MPC setting also has no such bandwidth lower bound, which is how we are able to achieve constant factor blow-up.

RAM-MPC via Off-the-Shelf ORAM When using off-the-shelf ORAM to achieve an MPC protocol, client/server blow-up is not the relevant cost metric. Instead, one considers the *circuit complexity* of the client program. Under this metric, Circuit ORAM [WCS15] remains best, achieving $O((w + \lg^2 n)\lambda)$ amortized Boolean gates per access, where λ is a statistical security parameter controlling failure probability. [AHR24] showed that Circuit ORAM can be naturally simplified to achieve OSAM rather than ORAM, and the resulting client uses $O(w\lambda)$ amortized Boolean gates per access.

Client/Multi-Server Protocols As mentioned in Section 1, rather than using off-the-shelf ORAM, we can construct custom protocols that use techniques unavailable in the client/server setting.

[OS97] pioneered this direction by considering a client/multi-server ORAM. The authors achieved feasibility results for different numbers of servers. For instance, with three servers, they achieved an information-theoretic protocol with communication complexity $O(n^{1/3} \lg^3 n)$. While seminal in kickstarting the study of multi-server ORAM, this result was superseded by later single-server techniques, such as Path ORAM and Circuit ORAM.

[LO13a] later demonstrated an efficient two-server ORAM. By leveraging both servers, it is possible to achieve a relatively simple and efficient protocol. While the [LO13a] construction is good for their considered client/multi-server setting, it becomes prohibitively expensive when one emulates client behavior with an MPC protocol to achieve full RAM-MPC. Namely, their protocol requires that on each access the client invoke a cryptographic primitive $O(\lg n)$ times. In the 2PC setting, these become non-black-box invocations, which is expensive; see also Remark 1.

Custom RAM-MPC Protocols [ZWR+16] was the first work to investigate how to achieve full RAM-MPC with better performance than via off-the-shelf ORAM. The authors modified the classic square-root complexity ORAM [GO96], replacing sorting networks by asymptotically-better *permutation networks*. The use of permutation networks does not make sense in the classic client/server setting, because programming the network uses $O(n)$ space, but they *can* be used in the MPC setting, because the MPC parties can afford to run linear-space algorithms. This RAM-MPC achieved relatively poor $\tilde{O}(\sqrt{n})$ asymptotic complexity, but the authors showed that their customized protocol concretely outperforms asymptotically-better Circuit ORAM for a variety of parameters.

DPF-based Protocols As noted in Section 1, Floram [Ds17] was pivotal in RAM-MPC. The authors used DPFs to trade computation for improved communication. Their construction still consumes relatively high communication overhead, and it also involves some invocations of non-black-box cryptography, which we avoid.

Also as mentioned in Section 1, Duoram [VHG23] adopted Floram's key ideas and used DPFs to achieve constant communication overhead for large word size in the 3PC. Our main result—constant-communication overhead—is thus similar to Duoram's. However, to get their approach to work in the 2PC setting, Duoram

increases communication overhead and resorts to using FHE-based private information retrieval, an expensive technique our approach avoids. Our approach achieves constant overhead in the 2PC setting, albeit for the weaker SAM model (or for the RAM model when working with very large words), and it relies only on OT.

Round-Efficient Protocols [JW18] studied RAM-MPC with a focus on *round complexity*. The authors observed that using Circuit ORAM to implement RAM-MPC incurs either (1) high round complexity, due to sequential computations inherent to the Circuit ORAM algorithm or (2) high communication complexity, due to the use of constant-round garbled-circuit-based techniques [Yao86]. The authors generated a 3PC instantiation of Circuit ORAM that leverages a number of custom improvements to achieve $O(\lg n)$ round complexity. For instance, the authors employ 3PC information-theoretic PIR to accelerate certain steps of lookup, and they use garbled-circuit-based techniques for the recursive position map. Their approach improves Circuit ORAM's round complexity, but it inherits its relatively high communication complexity.

[BKKO20] constructed a 3PC DORAM with *constant* round complexity and sublinear communication. They show a three-party DPF, which can be combined with two-party PIR protocols to achieve their stated asymptotic. The protocol requires linear computation.

Seeking to resolve the above tension between rounds and computation cost, [HV21] proposed a 2PC-RAM that achieves constant round complexity and *sublinear computation*. Their approach requires evaluation of an oblivious pseudorandom function (OPRF, [FIPR05]). The authors also showed how to improve communication/computation further by employing a particular powerful primitive—doubly efficient private information retrieval (DEPIR, [BIM04]).

Garbled RAM [LO13b] solves the problem of round complexity in RAM-MPC by allowing the MPC circuit to *non-interactively* perform memory lookups. In the semi-honest model, the communication cost of all GRAM memory lookups can be moved to the offline phase. The most efficient GRAM [HKO23] is notable in that it achieves its cost by implementing the Circuit ORAM algorithm inside a Garbled Circuit [Yao86]. Because [AHR24] showed that Circuit ORAM can be simplified (by removing the position map) to construct "Circuit OSAM", it is straightforward that [HKO23]'s GRAM can be simplified to construct "GSAM" at corresponding lower cost. Known approaches to GRAM/GSAM incur offline communication cost $\omega(w\lambda)$ where λ is a computational security parameter; notice that our communication *does not* include scaling in the product $w\lambda$. In Figure 6, we plot the communication consumption of the recent concretely efficient Garbled RAM implementation PicoGRAM [GTRS25]. In short, the advantage of constant round protocols comes at significant cost in terms of communication consumption.

Our approach has close to optimal round complexity – for each memory operation, parties exchange secret shares. Moreover, it is straightforward that our technique allows any number of non-dependent memory accesses to be performed in parallel, allowing further reduction in round consumption.

Other RAM Techniques [FNO25] constructed a 3PC-RAM that achieves sub-logarithmic per-access communication overhead for large enough words w. The protocol is remarkable in that it does this without cryptographic assumptions. At the highest level, the authors achieve their cost by leveraging 3PC information-theoretic PIR.

[FOSZ23] achieved a 3-server DORAM protocol with polylogarithmic overhead. The authors remarkably showed their RAM can scale to the billion-address regime and showed improved performance over 3-party FSS solutions. This targets a different party/security model from ours: their protocol is a 3-server DORAM, whereas our focus is semi-honest 2PC from OT and black-box PRGs.

Graph Techniques While our focus is on random access memory, one important application of the SAM model is its ability to efficiently handle pointer machine programs with improved asymptotics, and hence to implement, e.g., graph algorithms. The literature has numerous techniques that consider MPC handling of graph algorithms. In short, while SAM offers a generic graph traversal mechanism, custom MPC graph algorithms tailored to specific settings are often preferable in their target regimes. Roughly, prior MPC graph techniques fall into several categories:

Generic-ORAM-based graph algorithms. This was first explored by [KS14] and later by [LWN+15]. In general, such techniques incur cost commensurate with the number of cleartext steps multiplied into the cost of the ORAM. For graph algorithms that naturally compile to pointer-machine/SAM programs, SAM can

remove the logarithmic overhead of ORAM simulation. For instance, Figure 8 shows that for our considered graph DFS problem, the number of consumed low-level SAM operations is within a small constant factor of the requisite number of RAM accesses. We refer the reader to [PAR+26] for further discussion of the advantage of SAM as compared to RAM for graph problems, including SSSP, BFS/DFS, contact discovery, and more.

Algorithms for dense graphs. Several works have considered oblivious algorithms for *dense* graphs, i.e. where the number of edges is roughly quadratic in the number of vertices. Such algorithms can leverage the near-completeness of such graphs to attain higher performance, e.g. by using an adjacency matrix representation. The literature has explored such algorithms for BFS, SSSP (with and without negative weights), minimum spanning trees (MSTs), and various network flow algorithms [BSA13, ACM+13, AV15, AC22]. These works include algorithms with communication cost within either a constant factor (in the case of BFS, unweighted SSSP, MST) or a logarithmic factor blow-up (in the case of max flow, weighted SSSP) as compared to the corresponding cleartext runtime.

Comparing such techniques with the SAM-based approach is nuanced. For dense graphs, the above techniques have excellent asymptotics and enjoy relatively low constants. For sparse graphs, these techniques are less suitable, as they incur polynomial blow-up, whereas a SAM-based (or RAM-based) approach incurs worst-case poly-log overhead. In some cases, such as our DFS example, our approach shows that online communication blow-up can be constant.

The Scatter/Gather/Apply Paradigm. GraphSC [NWI+15] and its follow-on works [AFO+21, KKPG24] provide elegant MPC solutions to a particular class of graph problems. These works target so-called *scatter/gather/apply* problems, where each graph vertex is updated by its neighbors at each step. Namely, at each step a graph vertex (1) sends messages on its outgoing edges, (2) receives messages on its incoming edges, and (3) updates its label according to the messages it received. Because these steps happen across all vertices in parallel, the MPC protocol can be highly efficient, especially in terms of round complexity. This, for example, is ideal for certain problems of interest, such as PageRank, where the authors can repeatedly apply the above three steps to achieve the desired computation. Each iteration of the scatter/gather/apply paradigm (where vertices/edges have constant-sized labels) incurs $O(V + E)$ words of communication and $O(1)$ rounds [KKPG24]. Our SAM-based graph handling is not well suited to the scatter/gather/apply paradigm.

Other graph techniques. [Ost24] provides a custom MPC algorithm for positive-weight SSSP which involves an elegant rearrangement of the input graph; it incurs $O((V + E) \lg V)$ words of communication cost. This custom technique is better suited than our SAM-based approach because our SAM does not provide efficient priority queue operations, needed for Dijkstra's algorithm. Namely, one would need to compile priority queue operations to SAM operations, then build Dijkstra's on top of that. Using a simple binary heap priority queue, this would match [Ost24]'s online communication cost, but would have worse computation and offline costs.

Acknowledgements

This work was supported in part by NSF award CNS-2246353.

References

[AC22] Abdelrahaman Aly and Sara Cleemput. A fast, practical and simple shortest path protocol for multiparty computation. In Vijayalakshmi Atluri, Roberto Di Pietro, Christian Damsgaard Jensen, and Weizhi Meng, editors, *ESORICS 2022, Part III*, volume 13556 of *LNCS*, pages 749–755. Springer, Cham, September 2022.

[ACM+13] Abdelrahaman Aly, Edouard Cuvelier, Sophie Mawet, Olivier Pereira, and Mathieu Van Vyve. Securely solving simple combinatorial graph problems. In Ahmad-Reza Sadeghi, editor, *FC 2013*, volume 7859 of *LNCS*, pages 239–257. Springer, Berlin, Heidelberg, April 2013.

[AFO+21] Toshinori Araki, Jun Furukawa, Kazuma Ohara, Benny Pinkas, Hanan Rosemarin, and Hikaru Tsuchida. Secure graph analysis at scale. In Giovanni Vigna and Elaine Shi, editors, *ACM CCS 2021*, pages 610–629. ACM Press, November 2021.

[AH25] Ananya Appan and David Heath. Oblivious immutable memory. Cryptology ePrint Archive, Report 2025/593, 2025.

[AHR24] Ananya Appan, David Heath, and Ling Ren. Oblivious single access machines - A new model for oblivious computation. In Bo Luo, Xiaojing Liao, Jun Xu, Engin Kirda, and David Lie, editors, *ACM CCS 2024*, pages 3080–3094. ACM Press, October 2024.

[AKL+20] Gilad Asharov, Ilan Komargodski, Wei-Kai Lin, Kartik Nayak, Enoch Peserico, and Elaine Shi. OptORAMa: Optimal oblivious RAM. In Anne Canteaut and Yuval Ishai, editors, *EUROCRYPT 2020, Part II*, volume 12106 of *LNCS*, pages 403–432. Springer, Cham, May 2020.

[AV15] Abdelrahaman Aly and Mathieu Van Vyve. Securely solving classical network flow problems. In Jooyoung Lee and Jongsung Kim, editors, *ICISC 14*, volume 8949 of *LNCS*, pages 205–221. Springer, Cham, December 2015.

[BCG+19] Elette Boyle, Geoffroy Couteau, Niv Gilboa, Yuval Ishai, Lisa Kohl, and Peter Scholl. Efficient pseudorandom correlation generators: Silent OT extension and more. In Alexandra Boldyreva and Daniele Micciancio, editors, *CRYPTO 2019, Part III*, volume 11694 of *LNCS*, pages 489–518. Springer, Cham, August 2019.

[Ben64] Václav E Beneš. Permutation groups, complexes, and rearrangeable connecting networks. *Bell System Technical Journal*, 43(4):1619–1640, 1964.

[BGI15] Elette Boyle, Niv Gilboa, and Yuval Ishai. Function secret sharing. In Elisabeth Oswald and Marc Fischlin, editors, *EUROCRYPT 2015, Part II*, volume 9057 of *LNCS*, pages 337–367. Springer, Berlin, Heidelberg, April 2015.

[BGI16] Elette Boyle, Niv Gilboa, and Yuval Ishai. Function secret sharing: Improvements and extensions. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *ACM CCS 2016*, pages 1292–1303. ACM Press, October 2016.

[BIM04] Amos Beimel, Yuval Ishai, and Tal Malkin. Reducing the servers' computation in private information retrieval: PIR with preprocessing. *Journal of Cryptology*, 17(2):125–151, March 2004.

[BKKO20] Paul Bunn, Jonathan Katz, Eyal Kushilevitz, and Rafail Ostrovsky. Efficient 3-party distributed ORAM. In Clemente Galdi and Vladimir Kolesnikov, editors, *SCN 20*, volume 12238 of *LNCS*, pages 215–232. Springer, Cham, September 2020.

[BSA13] Marina Blanton, Aaron Steele, and Mehrdad Aliasgari. Data-oblivious graph algorithms for secure computation and outsourcing. In Kefei Chen, Qi Xie, Weidong Qiu, Ninghui Li, and Wen-Guey Tzeng, editors, *ASIACCS 13*, pages 207–218. ACM Press, May 2013.

[Ds17] Jack Doerner and abhi shelat. Scaling ORAM for secure computation. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *ACM CCS 2017*, pages 523–535. ACM Press, October / November 2017.

[DvDF+16] Srinivas Devadas, Marten van Dijk, Christopher W. Fletcher, Ling Ren, Elaine Shi, and Daniel Wichs. Onion ORAM: A constant bandwidth blowup oblivious RAM. In Eyal Kushilevitz and Tal Malkin, editors, *TCC 2016-A, Part II*, volume 9563 of *LNCS*, pages 145–174. Springer, Berlin, Heidelberg, January 2016.

[FIPR05] Michael J. Freedman, Yuval Ishai, Benny Pinkas, and Omer Reingold. Keyword search and oblivious pseudorandom functions. In Joe Kilian, editor, *TCC 2005*, volume 3378 of *LNCS*, pages 303–324. Springer, Berlin, Heidelberg, February 2005.

[FNO22] Brett Hemenway Falk, Daniel Noble, and Rafail Ostrovsky. 3-party distributed ORAM from oblivious set membership. In Clemente Galdi and Stanislaw Jarecki, editors, *SCN 22*, volume 13409 of *LNCS*, pages 437–461. Springer, Cham, September 2022.

[FNO25] Brett Hemenway Falk, Daniel Noble, and Rafail Ostrovsky. MetaDORAM: Info-theoretic distributed ORAM with less communication. In Niv Gilboa, editor, *ITC 2025*, volume 343 of *LIPIcs*, pages 6:1–6:23. Schloss Dagstuhl, August 2025.

[FOSZ23] Brett Hemenway Falk, Rafail Ostrovsky, Matan Shtepel, and Jacob Zhang. GigaDORAM: Breaking the billion address barrier. In Joseph A. Calandrino and Carmela Troncoso, editors, *USENIX Security 2023*, pages 3871–3888. USENIX Association, August 2023.

[GGH+13] Craig Gentry, Kenny A. Goldman, Shai Halevi, Charanjit S. Jutla, Mariana Raykova, and Daniel Wichs. Optimizing ORAM and using it efficiently for secure computation. In Emiliano De Cristofaro and Matthew K. Wright, editors, *PETS 2013*, volume 7981 of *LNCS*, pages 1–18. Springer, Berlin, Heidelberg, July 2013.

[GGM84] Oded Goldreich, Shafi Goldwasser, and Silvio Micali. How to construct random functions (extended abstract). In *25th FOCS*, pages 464–479. IEEE Computer Society Press, October 1984.

[GI14] Niv Gilboa and Yuval Ishai. Distributed point functions and their applications. In Phong Q. Nguyen and Elisabeth Oswald, editors, *EUROCRYPT 2014*, volume 8441 of *LNCS*, pages 640–658. Springer, Berlin, Heidelberg, May 2014.

[GKK+12] S. Dov Gordon, Jonathan Katz, Vladimir Kolesnikov, Fernando Krell, Tal Malkin, Mariana Raykova, and Yevgeniy Vahlis. Secure two-party computation in sublinear (amortized) time. In Ting Yu, George Danezis, and Virgil D. Gligor, editors, *ACM CCS 2012*, pages 513–524. ACM Press, October 2012.

[GKW18] S. Dov Gordon, Jonathan Katz, and Xiao Wang. Simple and efficient two-server ORAM. In Thomas Peyrin and Steven Galbraith, editors, *ASIACRYPT 2018, Part III*, volume 11274 of *LNCS*, pages 141–157. Springer, Cham, December 2018.

[GMW87] Oded Goldreich, Silvio Micali, and Avi Wigderson. How to play any mental game or A completeness theorem for protocols with honest majority. In Alfred Aho, editor, *19th ACM STOC*, pages 218–229. ACM Press, May 1987.

[GO96] Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious RAMs. *Journal of the ACM*, 43(3):431–473, 1996.

[GTRS25] Tianyao Gu, Afonso Tinoco, Sri Harish G. Rajan, and Elaine Shi. PicoGRAM: Practical garbled RAM from decisional Diffie-Hellman. In Yael Tauman Kalai and Seny F. Kamara, editors, *CRYPTO 2025, Part VIII*, volume 16007 of *LNCS*, pages 247–281. Springer, Cham, August 2025.

[HKO22] David Heath, Vladimir Kolesnikov, and Rafail Ostrovsky. EpiGRAM: Practical garbled RAM. In Orr Dunkelman and Stefan Dziembowski, editors, *EUROCRYPT 2022, Part I*, volume 13275 of *LNCS*, pages 3–33. Springer, Cham, May / June 2022.

[HKO23] David Heath, Vladimir Kolesnikov, and Rafail Ostrovsky. Tri-state circuits - A circuit model that captures RAM. In Helena Handschuh and Anna Lysyanskaya, editors, *CRYPTO 2023, Part IV*, volume 14084 of *LNCS*, pages 128–160. Springer, Cham, August 2023.

[HV21] Ariel Hamlin and Mayank Varia. Two-server distributed ORAM with sublinear computation and constant rounds. In Juan Garay, editor, *PKC 2021, Part II*, volume 12711 of *LNCS*, pages 499–527. Springer, Cham, May 2021.

[IKNP03] Yuval Ishai, Joe Kilian, Kobbi Nissim, and Erez Petrank. Extending oblivious transfers efficiently. In Dan Boneh, editor, *CRYPTO 2003*, volume 2729 of *LNCS*, pages 145–161. Springer, Berlin, Heidelberg, August 2003.

[JLS21] Zahra Jafargholi, Kasper Green Larsen, and Mark Simkin. Optimal oblivious priority queues. In Dániel Marx, editor, *32nd SODA*, pages 2366–2383. ACM-SIAM, January 2021.

[JW18] Stanislaw Jarecki and Boyang Wei. 3PC ORAM with low latency, low bandwidth, and fast batch retrieval. In Bart Preneel and Frederik Vercauteren, editors, *ACNS 2018*, volume 10892 of *LNCS*, pages 360–378. Springer, Cham, July 2018.

[KKPG24] Nishat Koti, Varsha Bhat Kukkala, Arpita Patra, and Bhavish Raj Gopal. Graphiti: Secure graph computation made more scalable. In Bo Luo, Xiaojing Liao, Jun Xu, Engin Kirda, and David Lie, editors, *ACM CCS 2024*, pages 4017–4031. ACM Press, October 2024.

[KM19] Eyal Kushilevitz and Tamer Mour. Sub-logarithmic distributed oblivious RAM with small block size. In Dongdai Lin and Kazue Sako, editors, *PKC 2019, Part I*, volume 11442 of *LNCS*, pages 3–33. Springer, Cham, April 2019.

[KS14] Marcel Keller and Peter Scholl. Efficient, oblivious data structures for MPC. In Palash Sarkar and Tetsu Iwata, editors, *ASIACRYPT 2014, Part II*, volume 8874 of *LNCS*, pages 506–525. Springer, Berlin, Heidelberg, December 2014.

[KU58] Andrei Nikolaevich Kolmogorov and Vladimir Andreevich Uspenskii. On the definition of an algorithm. *Uspekhi Matematicheskikh Nauk*, 13(4):3–28, 1958.

[LN18] Kasper Green Larsen and Jesper Buus Nielsen. Yes, there is an oblivious RAM lower bound! In Hovav Shacham and Alexandra Boldyreva, editors, *CRYPTO 2018, Part II*, volume 10992 of *LNCS*, pages 523–542. Springer, Cham, August 2018.

[LO13a] Steve Lu and Rafail Ostrovsky. Distributed oblivious RAM for secure two-party computation. In Amit Sahai, editor, *TCC 2013*, volume 7785 of *LNCS*, pages 377–396. Springer, Berlin, Heidelberg, March 2013.

[LO13b] Steve Lu and Rafail Ostrovsky. How to garble RAM programs. In Thomas Johansson and Phong Q. Nguyen, editors, *EUROCRYPT 2013*, volume 7881 of *LNCS*, pages 719–734. Springer, Berlin, Heidelberg, May 2013.

[LWN+15] Chang Liu, Xiao Shaun Wang, Kartik Nayak, Yan Huang, and Elaine Shi. ObliVM: A programming framework for secure computation. In *2015 IEEE Symposium on Security and Privacy*, pages 359–376. IEEE Computer Society Press, May 2015.

[MW22] Samir Jordan Menon and David J. Wu. SPIRAL: Fast, high-rate single-server PIR via FHE composition. In *2022 IEEE Symposium on Security and Privacy*, pages 930–947. IEEE Computer Society Press, May 2022.

[NWI+15] Kartik Nayak, Xiao Shaun Wang, Stratis Ioannidis, Udi Weinsberg, Nina Taft, and Elaine Shi. GraphSC: Parallel secure computation made easy. In *2015 IEEE Symposium on Security and Privacy*, pages 377–394. IEEE Computer Society Press, May 2015.

[OS97] Rafail Ostrovsky and Victor Shoup. Private information storage (extended abstract). In *29th ACM STOC*, pages 294–303. ACM Press, May 1997.

[Ost24] Benjamin Ostrovsky. Privacy-preserving dijkstra. In Leonid Reyzin and Douglas Stebila, editors, *CRYPTO 2024, Part IX*, volume 14928 of *LNCS*, pages 74–110. Springer, Cham, August 2024.

[PAR+26] Sage Pia, Ananya Appan, Maryam Rezapour, Amey Shukla, Nikhil Date, Benjamin Fuller, Ling Ren, and David Heath. Oblivious single access machines are concretely efficient. Cryptology ePrint Archive, Report 2026/451, 2026.

[Roy22] Lawrence Roy. SoftSpokenOT: Quieter OT extension from small-field silent VOLE in the minicrypt model. In Yevgeniy Dodis and Thomas Shrimpton, editors, *CRYPTO 2022, Part I*, volume 13507 of *LNCS*, pages 657–687. Springer, Cham, August 2022.

[Sch80] Arnold Schönhage. Storage modification machines. *SIAM Journal on Computing*, 9(3):490–508, 1980.

[Shi20] Elaine Shi. Path oblivious heap: Optimal and practical oblivious priority queue. In *2020 IEEE Symposium on Security and Privacy*, pages 842–858. IEEE Computer Society Press, May 2020.

[SvDS+13] Emil Stefanov, Marten van Dijk, Elaine Shi, Christopher W. Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. Path ORAM: an extremely simple oblivious RAM protocol. In Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung, editors, *ACM CCS 2013*, pages 299–310. ACM Press, November 2013.

[VHG23] Adithya Vadapalli, Ryan Henry, and Ian Goldberg. Duoram: A bandwidth-efficient distributed ORAM for 2- and 3-party computation. In Joseph A. Calandrino and Carmela Troncoso, editors, *USENIX Security 2023*, pages 3907–3924. USENIX Association, August 2023.

[Wak68] Abraham Waksman. A permutation network. *J. ACM*, 15(1):159–163, January 1968.

[WCS15] Xiao Wang, T.-H. Hubert Chan, and Elaine Shi. Circuit ORAM: On tightness of the Goldreich-Ostrovsky lower bound. In Indrajit Ray, Ninghui Li, and Christopher Kruegel, editors, *ACM CCS 2015*, pages 850–861. ACM Press, October 2015.

[WMK16] Xiao Wang, Alex J. Malozemoff, and Jonathan Katz. EMP-toolkit: Efficient MultiParty computation toolkit. `https://github.com/emp-toolkit`, 2016.

[WNL+14] Xiao Shaun Wang, Kartik Nayak, Chang Liu, T.-H. Hubert Chan, Elaine Shi, Emil Stefanov, and Yan Huang. Oblivious data structures. In Gail-Joon Ahn, Moti Yung, and Ninghui Li, editors, *ACM CCS 2014*, pages 215–226. ACM Press, November 2014.

[Yao86] Andrew Chi-Chih Yao. How to generate and exchange secrets (extended abstract). In *27th FOCS*, pages 162–167. IEEE Computer Society Press, October 1986.

[YWL+20] Kang Yang, Chenkai Weng, Xiao Lan, Jiang Zhang, and Xiao Wang. Ferret: Fast extension for correlated OT with small communication. In Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna, editors, *ACM CCS 2020*, pages 1607–1626. ACM Press, November 2020.

[ZE13] Samee Zahur and David Evans. Circuit structures for improving efficiency of security and privacy tools. In *2013 IEEE Symposium on Security and Privacy*, pages 493–507. IEEE Computer Society Press, May 2013.

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

A Details for Periodic RAM Resets

Recall from Section 3.1 that to support long-running RAM programs, we need a mechanism by which to periodically reset levels of our RAM trie. This section describes how to perform such resets.

First, recall that we fix a constant factor c (e.g. our implementation sets $c = 3$), and each level i of the RAM trie holding n_i trie nodes is assigned an *epoch length* $N_i = c \cdot n_i$.

As its main invariant, the SAM-based RAM ensures that each level-i SAM contains exactly n_i nodes of the RAM trie, stored at unread SAM addresses. Parent nodes at level $i-1$ store the SAM addresses of their children at level i.

To describe how we reset level i, we first describe two helper procedures `flush` and `initialize`, then we stitch these together into `reset`.

Algorithm 2 Flushing level i of a SAM-based RAM

Input: A SAM-based RAM represented by per-level SAM structures $\texttt{SAM}_j$. In the base case, $\texttt{SAM}_0$ is simply an array consisting of addresses into $\texttt{SAM}_1$. The name of a level i which should be flushed.
Output: The logical content of the RAM trie's level i in lexicographic order by logical position.

1: **if** $i = 0$ **then return** the array $\texttt{SAM}_0$
2: **else**
3: Recursively `flush` level $i - 1$; Let the resulting contents be an array `addrs`
4: **for** each SAM address `addr` in `addrs` **do**
5: call $\texttt{SAM}_i$.`Read`(`addr`)
6: **end for**
7: **return** the concatenation of all values read in the previous step.
8: **end if**

Algorithm 3 Reinitializing level i of a SAM-based RAM

Input: A level identifier i; desired node counts n_j for $j \leq i$; epoch constant c; an array $\texttt{data}_i$ of n_i values.
Output: Fresh SAMs for levels $1, ..., i$ and a level-0 array of addresses.

1: **if** $i = 0$ **then return** the array $\texttt{data}_0$.
2: **else**
3: Instantiate a fresh SAM $\texttt{SAM}_i$ of size $n_i \cdot (c + 1)$
4: **for** $j \in [n_i]$ **do**
5: Call $\texttt{SAM}_i$.`Alloc` to obtain a fresh address α_j
6: Call $\texttt{SAM}_i$.`Write`$(\alpha_j, \texttt{data}_i[j])$
7: **end for**
8: Concatenate all addresses produced in the loop above; let $\texttt{data}_{i-1}$ be the concatenation
9: Recursively `initialize` level $i - 1$ with content $\texttt{data}_{i-1}$
10: **return** all SAMs produced by the recursive call, together with $\texttt{SAM}_i$
11: **end if**

Flush Algorithm 2 formalizes `flush`. `flush` conceptually "destroys" the trie from level 0 to level i, reading the content on level i into an array. This involves first recursively flushing the parent level $i - 1$ into an array (notice that the manner by which we choose epoch lengths ensures all levels $j < i$ are flushed each time level i is flushed). By the above main invariant, that flushed content contains the n_i addresses of the live nodes on level i, so we can simply use `SAM.Read` to read each of them in turn, allowing us to arrange level i's content into an array. As the recursion base case, flushing the root of the tree is trivial.

Note that `flush` involves n_i SAM reads and one recursive call. Since we use flush on a trie, and since trie levels decrease in size geometrically as we move towards the root, the total cost of `flush` is $O(n_i)$ SAM reads.

Initialize Algorithm 3 formalizes `initialize`. In short, `initialize` takes as input an array of contents `data` which should be stored at the leaves of a trie with specified fan-out at each level. It then uses SAM operations to construct such a trie. Initialization proceeds by (1) constructing a fresh SAM structure of size $n_i + N_i$, (2) repeatedly calling `SAM.Alloc` on the fresh SAM structure to sample n_i addresses, (3) writing the initial contents into those n_i addresses, and (4) recursively initializing level $i - 1$ with those n_i addresses such that nodes stored in level $i - 1$ indeed are parent nodes of nodes in level i. As the recursion base case, initializing the root of the tree is trivial. Note that one key detail here is that while the epoch is of length $c \cdot n_i$, the SAM has size $(c + 1) \cdot n_i$. The extra n_i SAM operations are needed to support flush/initialize themselves.

Note that `initialize` involves n_i SAM allocations and n_i SAM writes, plus a recursive call. Since initialization constructs a trie, and since trie levels decrease in size geometrically as we move towards the root, the total cost of `initialize` is $O(n_i)$ SAM allocations/writes.

Algorithm 4 Resetting level i of a SAM-based RAM

Input: A level identifier i. A SAM-based RAM.

Output: The SAM-based RAM is updated such that level i is prepared for N_i more RAM accesses.

1: **if** $i = 0$ **then** do nothing
2: **else**
3: `flush` level i; let the result be $\texttt{data}_i$
4: `initialize` level i with content $\texttt{data}_i$ and the existing trie dimensions.
5: **end if**

Reset With `flush` and `initialize` formalized, it is easy to describe how to reset a level of the RAM trie; Algorithm 4 gives the procedure. In short, to reset level i, we flush it, and then we reinitialize it. This ensures level i is ready for N_i more RAM accesses.

The parties call `reset` on a fixed schedule. In particular, they keep a public timer `clock` that tracks the total number of RAM accesses that have occurred. Before each RAM access, they check to see which levels of RAM are *ready* to be reset. We say a level i is ready for reset if N_i divides `clock`. The parties then reset the largest level that is ready. Also, recall that resetting the largest ready level implicitly resets all smaller levels through recursive calls to `flush` and `initialize`.

Crucially, the fact that parties reset according to a fixed schedule means that obliviousness is immediate. Indeed, `reset` is clearly oblivious, since it simply calls `flush` and `initialize`, and each of these issues a publicly-known number of SAM operations in a publicly-known sequence.

The reset cost is amortized constant per level. To see this, consider a level i of the RAM trie. Denote the fan-out from level $i-1$ to i by f_i, i.e. $f_i = n_i/n_{i-1}$. Note that by our reset schedule, we reset level $i-1$ $f_i - 1$ times before we reset level i. Thus, we can determine the number of SAM operations consumed by calls to reset via a simple recurrence:

$$\texttt{Cost}(i) = (f_i - 1)\texttt{Cost}(i-1) + O(n_i)$$

Note that the $O(n_i)$ term emerges from the $O(n_i)$ SAM operations consumed by `flush` and `initialize`. Since $n_{i-1} = n_i/f_i$, the subproblem sizes decrease geometrically as we move toward the root. Thus the recurrence solves to $\texttt{Cost}(i) = O(n_i)$. Therefore, over any $N_i = \Theta(n_i)$-access epoch ending with a reset of level i, all resets of levels at most i consume $O(n_i)$ total SAM operations, or $O(1)$ amortized SAM operation per RAM operation. By applying this reasoning to the leaf level of our RAM trie, we observe that the reset overhead is a small constant factor.

B Details of DPF Shifting

We now explain Fact 2 in detail.

The key detail is that the output of the DPF is padded with an extra bit 1, and we can use this to adjust the DPF's output value.

More precisely, to adjust the evaluation point α by distance α', the parties simply evaluate the underlying DPF key at point $x \oplus \alpha'$ rather than at point x. Now, the parties hold the following by correctness of the DPF:

$$\begin{cases} (\llbracket \beta \rrbracket, \llbracket 1 \rrbracket) & \text{if } x = \alpha \oplus \alpha' \\ (\llbracket 0^m \rrbracket, \llbracket 0 \rrbracket) & \text{otherwise} \end{cases}$$

That is, they obtain at the point $\alpha \oplus \alpha'$ a secret share of the DPF payload and an extra 1. They locally scale their shares of this shared 1 by β', then add it to the payload:

$$\begin{cases} \llbracket \beta \rrbracket \oplus (\beta' \cdot \llbracket 1 \rrbracket) & \text{if } x = \alpha \oplus \alpha' \\ \llbracket 0^m \rrbracket \oplus (\beta' \cdot \llbracket 0 \rrbracket) & \text{otherwise} \end{cases} = \begin{cases} \llbracket \beta \oplus \beta' \rrbracket & \text{if } x = \alpha \oplus \alpha' \\ \llbracket 0^m \rrbracket & \text{otherwise} \end{cases} = \llbracket \texttt{Point}(\alpha \oplus \alpha', \beta \oplus \beta', x) \rrbracket$$

Thus, `Shift` is correct.

C Security Proof

We now provide a proof for Theorem 1.

Proof. By constructing a simulator for each party P_i.

The protocol is symmetric, so without loss of generality, assume P_0 is corrupt, and let $\mathcal{S}$ be P_0's simulator. In the semi-honest model, the simulator $\mathcal{S}$ interacts with the ideal functionality $\mathcal{F}_{\texttt{sam}}$ and the hybrid-world adversary $\mathcal{A}$. The proof goal is to show that the output of $\mathcal{S}$, together with the output of an ideal-world honest party P_1, is indistinguishable from P_0's real-world view, together with the output of real-world honest party P_1. In other words, we wish to show we can simulate P_0's view, in the context of the honest party's output.

Our simulator $\mathcal{S}$ forges a convincing protocol transcript by (1) playing the role of the preprocessing functionality $\mathcal{F}_{\texttt{pre}}$ and (2) keeping track of simulated party states. Precisely, $\mathcal{S}$ handles each of the four SAM operations as follows:

- **Initialize:** $\mathcal{S}$ simulates initialization by playing the preprocessing functionality. $\mathcal{S}$ first samples a size-n array of $(\lg n)$-bit strings R, where each entry R_i is uniform. R is meant to act as P_0's secret share of the uniformly random permutation π. It attaches R to P_0's simulated view. $\mathcal{S}$ n times runs the DPF simulator $\mathcal{S}_{\texttt{dpf}}$. For each $i \in [n]$, it calls:

  $$k_{0,i} \leftarrow \mathcal{S}_{\texttt{dpf}}(1^\lambda, 0)$$

  It attaches these DPF keys to P_0's simulated view. In addition, for each $i \in [n]$, $\mathcal{S}$ samples shares of random address α'_i and random value β'_i. Due to the structure of the preprocessing functionality, the hybrid-world adversary will at this point try to send its shares of α'_i, β'_i to the $\mathcal{F}_{\texttt{pre}}$ functionality. $\mathcal{S}$ simply collects and attaches these to the simulated view. Recall that in the protocol, the parties maintain a secret-shared state $[\![\sigma]\!]$. To mirror this, $\mathcal{S}$ also initializes an all-zeros array σ_0 that will simulate P_0's share of the state.

- **Allocate:** $\mathcal{S}$ fetches the i-th entry of the simulated permutation share R_i. Recall that on the allocate operation, the ideal functionality outputs a secret sharing, and it allows the adversary (i.e. $\mathcal{S}$) to choose its share. $\mathcal{S}$ sends $\texttt{Alloc}$ to $\mathcal{F}_{\texttt{sam}}$, choosing as its output share R_i. Accordingly, the functionality chooses a uniformly random address (without replacement) α_i, and it sends $R_i \oplus \alpha_i$ to the honest party P_1.

- **Write:** For a write operation, $\mathcal{S}$ holds input shares $[\![\alpha]\!]_0, [\![\beta]\!]_0$ and P_1 holds $[\![\alpha]\!]_1, [\![\beta]\!]_1$. $\mathcal{S}$ and P_1 send their respective shares to the ideal functionality $\mathcal{F}_{\texttt{sam}}$. The ideal functionality updates $A_\alpha \leftarrow A_\alpha \oplus \beta$. Let this be the i-th write. Recall that in the protocol, the parties exchange shares to reveal $\alpha \oplus \alpha'_i$ and $\beta \oplus \beta'_i$. But α'_i and β'_i are uniform, so $\mathcal{S}$ simulates the shares sent by P_1 via uniform strings, and it attaches those to P_0's simulated view. Let $\tilde{\alpha}$ (resp. $\tilde{\beta}$) denote the simulated revealed address difference (resp. simulated revealed value difference). $\mathcal{S}$ updates each index j of the simulated state share σ as follows:

  $$\sigma_j \leftarrow \sigma_j \oplus \texttt{Shift}(k_{0,i}, \tilde{\alpha}, \tilde{\beta}, j)$$

  This update to σ reflects the change to P_0's local share of state in the protocol.

- **Read:** For a read operation, $\mathcal{S}$ holds input share $[\![\alpha]\!]_0$. It sends $(\texttt{Read}, [\![\alpha]\!]_0)$ to the ideal functionality $\mathcal{F}_{\texttt{sam}}$. The honest party P_1 sends $(\texttt{Read}, [\![\alpha]\!]_1)$ to $\mathcal{F}_{\texttt{sam}}$. Recall that $\mathcal{F}_{\texttt{sam}}$ sends the address α to the adversary (i.e., $\mathcal{S}$) in the clear, so $\mathcal{S}$ receives α from the functionality. Recall also that on a read $\mathcal{F}_{\texttt{sam}}$ outputs a secret sharing, and it allows the adversary to choose its share. $\mathcal{S}$ fetches from its simulated state σ_α and chooses this value as its output share. Accordingly, the ideal functionality fetches A_α and sends $\sigma_\alpha \oplus A_\alpha$ to the honest party P_1. $\mathcal{S}$ computes and attaches to P_0's simulated view the particular value $[\![\alpha]\!]_0 \oplus \alpha$, consistent with the real-world message that would reveal address α to P_0.

Claim: The following two quantities are indistinguishable:

- **Ideal** P_0's simulated view paired with P_1's output.
- **Real** P_0's view in $\Pi_{\texttt{sam}}$ paired with P_1's output from $\Pi_{\texttt{sam}}$.

Indeed, this essentially follows by construction of the above simulator. We note the important points of comparison:

- **DPF keys:** In the real world, P_0 obtains a DPF key from $\mathcal{F}_{\texttt{pre}}$; in the ideal world, P_0 obtains simulated DPF keys from the simulator. But those DPF keys are generated by $\mathcal{S}_{\texttt{dpf}}$, and Definition 2 states that real keys are indistinguishable from simulated keys (note that P_1's keys never appear in P_0's view).
- **Allocate:** In our protocol, allocation is a local operation, so simulation is trivial.
- **Write:** In both worlds, P_0 observes two uniformly random strings incoming from P_1, one for a difference in addresses, one for a difference in values.
- **Read:** In both worlds, P_0 receives from P_1 a share consistent with an address that has never been read before.

Thus, the real and ideal worlds are indistinguishable, so $\Pi_{\texttt{sam}}$ indeed realizes the ideal functionality $\mathcal{F}_{\texttt{sam}}$. □

D Noisiness in round complexity of SAM-based RAM

The noise in Figure 4's plot of rounds results from a complex interaction of two implementation details. For the curious reader, we explain. First, as n increases, the SAM-based trie increases in depth to accommodate the n leaves. Note that this happens in discrete break-points: a certain depth of RAM trie can support several sizes n. When depth increases, so does round complexity. Second, we implement an optimization on our DPF trees whereby if a tree encodes leaf data of size less than the AES block size, e.g. size 64 leaves instead of size 128, then we *pack* that data into leaves, e.g. placing two slots into a single DPF leaf. This improves cost of DPF generation/evaluation, but it costs extra rounds to *shift* (see Section 2.3) a DPF's stored value, because we must conditionally write to only one "part" of the DPF's stored value. In the SAM-based trie, there are sometimes opportunities to use smaller words in internal metadata SAMs. This specifically occurs when a level of the SAM-based trie happens to have low fan-out. In this case, we can use our packing optimization on one of the internal SAMs, and round complexity slightly increases. For larger n, the trie's fan-out increases further, the packing optimization can no longer be used, and the added rounds are removed.