Fast ORAM with Server-aided Preprocessing and Pragmatic Privacy-Efficiency Trade-off ⋆

Vladimir Kolesnikov[1], Stanislav Peceny[2], Ni Trieu[3], and Xiao Wang[4]

[1] kolesnikov@gatech.edu, Georgia Tech, Atlanta, GA, USA
[2] stan.peceny@gatech.edu, Georgia Tech, Atlanta, GA, USA
[3] nitrieu@asu.edu, Arizona State University, Tempe, AZ, USA
[4] wangxiao1254@gmail.com, Northwestern University, Evanston, IL, USA

Abstract. Data-dependent accesses to memory are necessary for many real-world applications, but their cost remains prohibitive in secure computation. Prior work either focused on minimizing the need for data-dependent access in these applications, or reduced its cost by improving oblivious RAM for secure computation (SC-ORAM). Despite extensive efforts to improve SC-ORAM, the most concretely efficient solutions still require ≈ 0.7s per access to arrays of 2^{30} entries. This plainly precludes using MPC in a number of settings.

In this work, we take a pragmatic approach, exploring how concretely cheap MPC RAM access could be made if we are willing to allow one of the participants to learn the access pattern. We design a highly efficient Shared-Output Client-Server ORAM (SOCS-ORAM) that has constant overhead, uses one round trip of interaction per access, and whose access cost is independent of array size. SOCS-ORAM is useful in settings with hard performance constraints, where one party in the computation is more trust-worthy and is allowed to learn the RAM access pattern. Our SOCS-ORAM is assisted by a third helper party that helps initialize (and reinitialize, as needed) the protocol and is designed for the honest-majority semi-honest corruption model.

We implement our construction in C++ and report its performance. For an array of length 2^{30} with 4B entries, we communicate 13B per access and take essentially no overhead beyond network latency.

Keywords: Cryptography, Secure computation, Efficient protocols, ORAM

1 Introduction

Real-world applications rely heavily on data-dependent accesses to memory. Despite many recent improvements, such accesses remain a bottleneck when evaluated in secure two-party, three-party, and the general multi-party computation (2PC, 3PC, MPC).[1] While in plaintext execution such accesses are cheap

⋆ This work is an extended version of [1].

[1] MPC refers to protocols involving more than one party. In our work, we use the term interchangeably to refer to 2PC or 3PC, depending on the context.

constant-time operations, they are expensive in MPC, since access pattern must remain hidden. A naive secure solution to this problem is linear scan, which hides the access pattern by touching every element in memory and multiplexing out the result. This, of course, incurs overhead linear in memory size for each access. A much more scalable approach is to instead use more complex Oblivious RAM (ORAM) protocols [2], which achieve polylog complexity, while still hiding access patterns.

The first ORAM considered the client-server setting [2], where a client wishes to store and access her private array on an untrusted server. Soon after, initiated by [3,4], ORAM was shown applicable to RAM-based 2PC: Secure RAM access was achieved for 2PC simply by having the parties execute ORAM client inside secure computation, while both parties share the state of the server.

Despite extensive research focused on optimizing ORAM for secure computation (SC-ORAM) and ORAM in general, the overhead remains prohibitive for many applications. For example, a recent SC-ORAM Floram [5] takes ≈ 2 seconds per access, communicates $\approx$ 5MBs, and requires 3 communication rounds on arrays of size 2^{30} with 4-byte elements.

Such ORAM performance is unacceptable in settings where many accesses of large arrays are needed. Examples include network traffic or financial markets analyses, where data is continuously generated and frequently accessed.

3PC: 2PC with a Helper Server. Fortunately, many real-world applications can use a third party to help with computation. This third party may already be a participant of the computation (e.g. provide input and/or receive output) or can be brought as an (oblivious) helper server. As secure computation of many functions is much faster in a 3-party honest-majority setting than in the two-party setting, [6] ask whether SC-ORAM can also be accelerated. [6] present a solution and report total wall-clock time of $1.62s$ on a 2^{36}-element array. The rest of the measurements focus on the online costs; based on the discussion in the paper we estimate the total cost for 2^{30}-element array is $\approx$ 1.25s. A follow-up work [7] then asymptotically reduces the bandwidth of [6], but still reports $\approx$ 0.7s CPU time per access on a 2^{30}-element array. Although this is an improvement over 2-party SC-ORAMs, a $0.7s$ RAM access time will still be considered prohibitive in many (most?) realistic use scenarios.[2]

Our Goals. In this work, we are interested in exploring what secure computation is possible in settings with hard performance constraints. We thus seek maximizing performance at the cost of relaxing the security guarantees.

We start in the easier 3-party setting, and ask whether we can get further significant improvement if one party in the computation is more trust-worthy and is allowed to *learn the access pattern.*

[2] Note, accessing smaller-size memories would be, of course, cheaper: [7] reports 0.1s CPU time per access on a 2^{10}-element array. For context, note that garbled circuit linear scan of 2^{10}-element array would require about 2^{15} gates and would take less than 0.1s on a 1Gbps LAN.

This trust model may naturally occur in real-world scenarios (see Section 1.1) e.g. if one of the parties is an established entity with trusted oversight, such as a government or a law enforcement agency.

Our Setting. We summarize our considered setting. Our Shared-Output Client-Server ORAM (SOCS-ORAM) protocol is run by three parties $\mathcal{A}$, $\mathcal{B}$, and $\mathcal{C}$. $\mathcal{B}$ holds an array $\mathbf{d}$ of n l-bit entries. $\mathcal{A}$, $\mathcal{B}$, and $\mathcal{C}$ first initialize SOCS-ORAM with $\mathbf{d}$. $\mathcal{A}$ then requests up to k read or write accesses to $\mathbf{d}$. After k accesses, SOCS-ORAM can be reinitialized to provision for up to k more accesses. The number of reinitializations is arbitrary. For each access, $\mathcal{A}$ inputs index $i \in [n]$ and operation *op* (`read` or `write`). $\mathcal{A}$ and $\mathcal{B}$ hold a sharing $[\![x]\!]$ of a value to be written. $\mathcal{C}$ holds no input and *does not* participate in ORAM access; he is used to help initialize and reinitialize SOCS-ORAM.

We stress that reinitialization is an important feature (e.g., vs. an initialization for more accesses). This reduces memory requirements of our implementation, as well as latency – MPC computation can commence sooner, as initialization is now shorter. This may be important for reactive functionalities. Importantly, in many computations (e.g. RAM-machine based MPC), execution depends on the input and the number of accesses *cannot be predicted*.

All parties are semi-honest and do not collude with one another. We allow $\mathcal{A}$ to learn the access pattern – indeed $\mathcal{A}$ can be viewed as ORAM client; $\mathcal{B}$ and $\mathcal{C}$ learn nothing from the computation.

1.1 Motivation

Recall that our work explores a trade-off between maximizing performance at the cost of relaxing security guarantees. This is a natural and pragmatic research direction. For example, a similar trade-off is also considered in Blind Seer [8], a scalable privacy-preserving database management system that supports a rich query set for database search and addresses query privacy. [8] motivate the trade-off, warn of potential pitfalls, and convincingly argue its benefits. Our work is complementary. SOCS-ORAM can be used as a drop-in *no-cost* replacement to improve security of Blind Seer's unprotected RAM access. Indeed, Blind Seer similarly uses three parties but allows two parties (i.e. all parties other than helper server (server in their notation)) to learn the access pattern, compared to only one party in our work. We believe this can be a crucial difference as trust is unbalanced in natural settings (e.g., bank may be trusted more than clients, wireless service provider – more than each individual customer, and government agency – more than private businesses).

We now briefly discuss several motivating applications spanning network security, financial markets, and review Blind Seer's air carrier's passenger manifest analysis.

Network Data Analysis. There is a significant benefit in operation of large-scale analysis centers, such as Symantec's DeepSight Intelligence Portal. These centers

collect network traffic information from a diverse pool of sources such as intrusion detection systems, firewalls, honeypots, and network sensors, and can be used to build analysis functions to detect network threats [9].

Network data is highly sensitive; revealing network configuration and other details may significantly weaken its defences. Using MPC instead to enable expert network analysis and vulnerability reporting is a (costly) solution. Network analysis works with large volumes of data (e.g. Symantec's DeepSight has billions of events) and requires a large number of RAM accesses. Paying $\approx 1s$ per RAM access is clearly not feasible for even trivial analyses.

Using SOCS-ORAM and placing, arguably, a reasonable trust in the analysis center (allowing it to learn RAM access pattern), may potentially enable this application.

Financial Markets Analysis. SOCS-ORAM can be used to identify fraudulent activity, such as insider trading in financial markets. In this use case, a regulatory agency such as SEC or FINRA investigates and analyzes data from brokerages. Typically SEC initiates its investigation based on suspicious activity in an individual security. SEC next makes a regulatory request. So-called *blue sheets data* brokerage response contains trading and account holder information. SEC's Market Abuse Unit (MAU) then runs complex analyses on the data, which may contain billions of rows. We note that there are privacy concerns for both parties. SEC does not want to reveal what they are investigating, while brokerages do not want to share their clients' data that is not essential for the investigation. This scenario is a fit for our SOCS-ORAM: The brokerage learns nothing about the investigation, while SEC learns only the output of the analysis functions, alongside the access pattern.

Passenger Manifests Analysis. Passenger manifests search and analysis is one of the motivating applications of the Blind Seer DBMS [8]. It considers a setting where a law enforcement agency wants to analyze or search air carrier's manifests for specific patterns or persons. The air carrier would like to protect its customers' data, and hence reveal only the data necessary for the investigation. The law enforcement agency would like to protect its query. Today's approach may be to simply provide the manifests to the agency. Using MPC (and keeping the private data private) would help allay the negative popular sentiment associated with large scale personal data collection by government.

1.2 Contributions

We present a highly efficient shared-output client-server ORAM (SOCS-ORAM) scheme. Here the client $\mathcal{A}$ knows the logical indices of the RAM queries, and the results are additively (XOR) secret-shared between her and the server $\mathcal{B}$, allowing them, unlike the output of classical ORAM, to be directly used in MPC.

This construction is suitable for secure computation applications with hard performance constraints where one party is more trustworthy. While in MPC none of the parties learns the set of queried RAM locations, we reveal them to

one of the parties. Further, our SOCS-ORAM uses a semi-honest third party who helps initialize and reinitialize our construction, but is not active when invoking `access`. In exchange, we achieve very high ORAM performance, whose *only* non-trivial cost is communication rounds. In particular, we present:

- **Efficient SOCS-ORAM Construction.** Our construction consists of efficient third-party-aided initialization and reinitialization protocols and an efficient 2-party access protocol.
  Our initialization protocol does not execute MPC; it runs PRG and generates a random permutation, all evaluated outside MPC. It requires 4 message flows (the first 2 and the last 2 can be parallelized). To set up SOCS-ORAM for k accesses to an array of size n, we require sending $2n + 2k$ l-bit array entries and three pseudo-random κ-bit seeds (κ is the computational security parameter). $nl + \kappa$ bits are sent by $\mathcal{B}$ (secret-sharing of the input array with one share sent by a seed), and the rest by $\mathcal{C}$. Reinitialization has the same communication and comparable computation to `Π-init`.
  Our access protocol communicates only 2 array elements, a single array index, and an additional bit, and requires a single roundtrip of interaction. No cryptography is involved in our access protocol: We only use the XOR operation and plaintext array access. The cost of our access protocol is independent of array size (but system-level implementation costs manifest for larger array sizes).
- **Resulting Efficient Implementation.** We implement and experimentally evaluate our approach. Our experimental results indicate that on an array with 2^{30} entries each of 4B, we communicate 13B per access and run in 2.13ms on a 2ms latency network (as set by the Linux `tc` command; the actual latency, due to system calls overhead is closer to 2.13ms).
  Thus, our wall-clock time is very close to latency cost. While our setting is much simpler than that of SC-ORAM, state-of-the-art 3-party SC-ORAM of [7] reports ≈ 0.7s CPU time for arrays of the same size, while all our runs ran in less than 0.019ms of computation. Similarly, our access communication is on the order of bytes instead of MBytes, and we use 1 round trip of interaction instead of $O(\log n)$. For a 2^{30} array of $4B$ entries (i.e. 4GB size array) and 2^{20} accesses, the cost to initialize our SOCS-ORAM (preprocessing) is 3.1 minutes and 8GB communication. The reinitialization cost is similar: setting up another 2^{20} accesses requires 3.7 minutes and 8GB communication.

2 Notation

- Party $\mathcal{A}$ (Alice, client) inputs access indices i.
- Party $\mathcal{B}$ (Bob, server) inputs array $\mathbf{d}$.
- Party $\mathcal{C}$ (Charlie, the third party helper).
- κ denotes the computational security parameter (e.g. 128).
- $[n]$ denotes the sequence of natural numbers $0, \ldots, n-1$. $[n, n+k]$ denotes the sequence $n, \ldots, n+k-1$.

- We denote arrays in bold, index them with subscripts, and use 0-based indexing. E.g., $\mathbf{d}_0$ is the first element of array $\mathbf{d}$.
- We sometimes add subscript notation to arrays to indicate that for a bit array $\mathbf{f}$ and two arrays $\mathbf{s_0}, \mathbf{s_1}$, the array $\mathbf{s_f}$ holds entries from $\mathbf{s}_{\mathbf{f}_i}$ at index i. Further, we double-index arrays with a ',' (e.g. $\mathbf{s}_{\mathbf{0},i}$ is i-th element of s_0).
- We denote negation of a bit b as $\bar{b}$.
- We manipulate XOR secret shares.
  - We use the shorthand $[\![\mathbf{d}]\!]$ to denote a (uniform) sharing of array $\mathbf{d}$.
  - Subscript notation associates shares with parties. E.g., $[\![\mathbf{d}]\!]_A$ is a share of $\mathbf{d}$ held by party A.

3 Oblivious RAM (ORAM) Review

Our notions of client-server oblivious RAM (ORAM) and secure-computation oblivious RAM (SC-ORAM) are standard.

Client-Server ORAM. A client-server ORAM [2] is a protocol that enables a *client* to outsource data to an untrusted *server* and perform arbitrary read and write operations on that outsourced data without leaking the data or access patterns to the server.

An ORAM specifies (1) an initialization protocol that takes as input an array of entries and initializes an oblivious working array with those entries, and (2) an access protocol that implements each *logical* (`read` and `write`) access on the oblivious structure with a sequence of polylog *physical* accesses. ORAM may optionally specify a dedicated reinitialization procedure as the working array output by initialization allows for a limited number of accesses.

We now present the ORAM functionality. Client inputs an array $\mathbf{d}$ of length n. For each access, client inputs operation op (`read` or `write`), index $i \in [n]$, and, if writing, the value x to write. Server inputs $\perp$. If $op = \texttt{read}$, client outputs $\mathbf{d}_i$ and server outputs $\perp$; if $op = \texttt{write}$, client and server set $\mathbf{d}_i = x$ and output $\perp$.

The ORAM's security guarantee is that the physical access patterns produced by the access protocol for any two sequences of logical accesses of the same length must be computationally indistinguishable. We take the security definition almost verbatim from [10].

Definition 1. *Let $\boldsymbol{y} := ((op_0, i_0, x_0), (op_1, i_1, x_1), \ldots, (op_{k-1}, i_{k-1}, x_{k-1}))$ denote a sequence of logical accesses of length k, where each op denotes* $\texttt{read}(i)$ *or* $\texttt{write}(i, x)$*. Specifically, i denotes the array index being read or written, and x denotes the data being written. Let $A(\boldsymbol{y})$ denote the (possibly randomized) sequence of physical accesses to the remote storage given the sequence of logical accesses $\boldsymbol{y}$. ORAM is said to be secure if for any two sequences of logical accesses $\boldsymbol{y}$ and $\boldsymbol{z}$ of the same length, their access patterns $A(\boldsymbol{y})$ and $A(\boldsymbol{z})$ are computationally indistinguishable by anyone but the client.*

RAM-Based Secure Computation. [3] noted the idea of using ORAM for secure multi-party computation (SC-ORAM). [4] proposed the first complete SC-ORAM construction. In SC-ORAM, the key idea is to have each party store a share of the server's ORAM state, and then execute the ORAM client access algorithms via a general-purpose secure computation protocol.

As the server's state is now secret-shared between both parties and the client is executed inside secure computation, we no longer refer to the physical parties as client and server but $\mathcal{A}$ and $\mathcal{B}$. In SC-ORAM, $\mathcal{A}$ and $\mathcal{B}$ input a sharing of an array $[\![\mathbf{d}]\!]$ of size n. For each access, they input a sharing of operation $[\![op]\!]$ (`read` or `write`), a sharing of index $[\![i]\!] \in [n]$, and a sharing of a value to write $[\![x]\!]$. If $op = \texttt{read}$, $\mathcal{A}$ and $\mathcal{B}$ output $[\![\mathbf{d}_i]\!]$; if $op = \texttt{write}$, set $[\![\mathbf{d}_i]\!] = x$ and output $\perp$.

There are a few key differences between client-server ORAM and SC-ORAM that [11] explicate:

- In the client-server ORAM, the client owns the array and also accesses it. Hence, the privacy requirement is unilateral. In SC-ORAM, both the array and the access are distributed and neither party should learn anything about the array or the access pattern.
- In the client-server ORAM, the client's storage should be sublinear, whereas in SC-ORAM, linear storage is distributed across both parties.
- Client-server ORAMs have traditionally been measured by their bandwidth overhead and client storage. [12] observed that for SC-ORAMs the size of the client circuits is more relevant to performance.
- In SC-ORAM, the initialization protocol must be executed securely; in 2PC this cost is often prohibitive.

4 Related Work

We present a highly efficient 3-party SOCS-ORAM with applications in secure computation. We therefore review related work that improves (1) SC-ORAMs in the standard 2-party setting, (2) SC-ORAMs in the 3-party setting, and (3) Garbled RAM schemes that equip Garbled Circuit with a sublinear cost RAM without adding rounds of interaction. We also briefly discuss (4) differential obliviousness (DO), (5) multi-server ORAMs in the client-server setting, and (6) private information retrieval (PIR).

2-*party SC-ORAM.* [3] proposed the basic idea of SC-ORAM, where the parties share the ORAM server role, while having the ORAM client algorithm executed via secure computation. [4] presented a specific SC-ORAM construction that started a long line of research to improve SC-ORAM. [12] observed that when using ORAMs for secure computation, the size of the circuits is more relevant to performance than the traditional metrics such as bandwidth overhead and client storage. Then they presented a heuristic SC-ORAM optimized for circuit complexity. [13] followed up with Circuit ORAM, which further reduced circuit complexity. [11] showed that by relaxing asymptotics, one can produce a scheme that outperforms Circuit ORAM for arrays of small to moderate sizes.

We note that all [4,12,13,11] are recursively structured and as a result require $O(\log n)$ rounds of communication per access; they have expensive initialization algorithms and high memory overhead. E.g., [5] observed they could not handle arrays of sizes larger than $\approx 2^{20}$ on standard hardware. With this in mind, [5] introduced Floram that requires 3 rounds per access and significantly decreases memory overhead and initialization cost. Floram requires linear work per access. Crucially, this work is inexpensive since it is local and executed outside secure computation, unlike in the MPC-run linear scan. Still, despite a large concrete improvement, [5] takes ≈ 2 seconds per access and communicates $\approx$ 5MBs in communication on arrays of size 2^{30} with 4-byte elements.

3-party SC-ORAM. [6] explore whether adding a third party to SC-ORAM can improve performance. They present a construction secure against semi-honest corruption of one party, which uses custom-made protocols to emulate the client algorithm of the binary tree client-server ORAM [14] in secure computation. For a 2^{36}-element array of 4-byte entries, their access runs in $1.62s$ wall-clock time when executed on two co-located EC2 t2.micro machines. Their solution further requires $O(\log n)$ communication rounds for an array of size n. [7] followed up on their work and designed custom-made protocols to instead emulate the Circuit ORAM [13] client. While their technique still requires $O(\log n)$ communication rounds per access, they asymptotically decrease the bandwidth of [6] by the statistical security parameter. Concretely, they report $\approx$ 0.7s CPU time per access on a 2^{30}-element array of 4-byte entries, when run on co-located AWS EC2 c4.2xlarge instances. While we are not directly comparable, we execute one access in one communication round and all our runs took less than 0.019ms on localhost on a same-size array.

[15] showed how to combine their 3-server distributed point function (DPF) with any 2-server PIR scheme to obtain a 3-server ORAM and then extended it to SC-ORAM. Their access protocol runs in constant rounds, requires sublinear communication and linear work, and makes only black-box use of cryptographic primitives. [16] present 3-party SC-ORAM from oblivious set membership that aims to minimize communication complexity. These works do not offer implementation and evaluation, and we do not directly compare with their performance.

Garbled RAM (GRAM). GRAM is a powerful technique that adds RAM to GC while preserving GC's constant rounds of interaction. This technique originated in [17] but was not suitable for practice until [18] introduced EpiGRAM. Although [18] do not implement EpiGRAM, they estimate that for an array of 2^{20} entries of 16B, the per-access communication amortized over 2^{20} accesses is $\approx$ 16MB. In comparison, our work communicates $\approx 0.09KB$ (initialization included) amortized over the same number of accesses. An EpiGRAM implementation is now available as part of an MPC compiler [19]. The authors do not evaluate EpiGRAM separately and only present benchmarks that evaluate entire programs.

Differential Obliviousness (DO). DO [20] is a relaxation of access pattern privacy. As opposed to simulation-based ORAM privacy guarantees, DO requires the program's access pattern to be differentially private. [20] showed that for some programs DO incurs $O(\log\log n)$ overhead in contrast to ORAM's polylog complexity. We forfeit access pattern privacy against $\mathcal{A}$.

Multi-Server Client-Server ORAM. [21] proposed exploring client-server ORAM in a model with two non-colluding servers storing the client's data. The client interacts with the servers to access data, while the servers do not interact with each other. Their solution achieved parameters that were asymptotically better than those realized by any single-server solution. It is an involved construction which requires $O(\log n)$ communication rounds, whereas we use a single round. A follow-up work [22] reduced the asymptotic communication bandwidth, but did not improve round complexity. [23] introduced a two-server ORAM that combines any tree-based ORAM with two-server PIR to get a one-round solution, but requires each server to perform linear scan over the entire data. Our work only requires constant work. Their construction also requires communicating $10\log n$ encrypted array entries per logical access, while ours requires only 2 array entries, a single position map entry, and one bit; i.e. it is independent of n. Further, the helper server is offline in our access protocol.

[24] presented the first protocol in the multi-server setting to achieve perfect security and explored whether there are any implicit advantages to the multi-server setting. They focused on optimizing communication bandwidth while maintaining perfect security and their construction achieved $\log n$ bandwidth for certain block sizes. [25] showed several constructions of which the most suitable for secure computation is a PIR-based 4-server construction that has constant overhead, but requires linear amount of local work on the servers. They did not implement their construction, but their PIR is constructed from a distributed point function (DPF), which requires $\log n$ sequential PRG evaluations, whereas we require only a constant number of plaintext array accesses and XORs.

Private Information Retrieval (PIR). PIR [26] enables a client to retrieve a selected entry from an array such that no information about the queried entry is revealed to the one (or multiple) server holding the array. Thus, PIR is concerned with the privacy of the client. There are many flavors of PIR, one of which is Symmetric PIR (SPIR) [27]. SPIR has an additional requirement that the client learns only about the elements she is querying, and nothing else. For our purposes, the main difference between PIR and ORAM is that PIR supports only read operations. While we do not further discuss PIR, we emphasize that PIR is sometimes used as a building block of ORAM constructions (e.g. in [5,23,7,25,15] discussed above).

5 Technical Overview

We introduce and construct, at the high-level, shared-output client-server oblivious RAM (SOCS-ORAM), a useful building block for efficient MPC. We present

our construction by first simply achieving a basic limited functionality, and then securely building on that to achieve the goal. Full formal algorithms, with accompanying proofs of correctness and security, are in Section 6.

Recall from Section 1, SOCS-ORAM is run by parties $\mathcal{A}$, $\mathcal{B}$, and $\mathcal{C}$, where $\mathcal{B}$ holds an array $\mathbf{d}$ of length n. On access, $\mathcal{A}$ inputs operation *op* (`read` or `write`) and an index $i \in [n]$. $\mathcal{A}$ and $\mathcal{B}$ also input a sharing of value $[\![x]\!]$ to write. $\mathcal{C}$ is a helper party that aids with SOCS-ORAM (re)initialization and is not active during array access. Initialization provisions for k *dynamic* accesses. Each reinitialization provisions for k additional accesses. We consider honest majority with security against semi-honest corruption and allow $\mathcal{A}$ to learn (or know) the access pattern.

Goal. We aim to build a concretely efficient SOCS-ORAM using plaintext array lookup, XOR masking, and PRGs, with constant access overhead and a single round trip of interaction, whose computational cost is close to plaintext array access. We design such SOCS-ORAM at the concession of allowing one party to learn the access pattern. We describe our construction next.

Basic Initialization for our SOCS-ORAM. $\mathcal{A}$ and $\mathcal{B}$, with the help of $\mathcal{C}$, initialize $\mathbf{D}$ with $\mathbf{d}$ (cf. Figure 1; $\mathbf{d}$ is $\mathcal{B}$'s input array used to initialize the working array $\mathbf{D}$). $\mathcal{A}$ and $\mathcal{B}$ receive $[\![\mathbf{D}]\!]$, which is permuted according to a random permutation π unknown to $\mathcal{B}$ and secret-shared using randomness neither party knows. Uniform secret sharing ensures that upon access neither party learns anything about the value of the array entry they are retrieving; permuting ensures logical index is hidden from $\mathcal{B}$. Clearly, this *initially* (i.e. before any accesses) hides array entries and their positions. With $\mathcal{C}$'s help, this structure can be set up cheaply.

Handling Repeated Accesses. Following the above initialization, $\mathcal{A}$ will access $[\![\mathbf{D}]\!]$, possibly accessing the same logical index multiple times. Recall, only $\mathcal{A}$ is allowed to learn the access pattern. $\mathcal{C}$ is oblivious by not participating in the access protocol. The challenge is to preclude $\mathcal{B}$ from learning the access pattern.

As hinted above, if no logical index is accessed twice, $\mathcal{B}$ learns nothing, since each entry $[\![\mathbf{D}_i]\!]$ is placed in a random physical position $\pi(i)$. To access a logical index more than once, each time the physical location must be different: the value must be copied to a *new random* location.

We modify initialization to create the space for copied values. We *extend* the working array $\mathbf{D}$ with space for k entries (*shelter*), and secret-share and permute the *extended* $\mathbf{D}$ according to $\pi : [n+k] \mapsto [n+k]$. This is cheap with $\mathcal{C}$'s help.

We next show how to copy the read entry to a new index (corresponding to the next available shelter entry) in $[\![\mathbf{D}]\!]$, *obliviously* to $\mathcal{B}$. Then, at the next access to this element, $\mathcal{B}$ is accessing a random share at a random-looking index.

`read` *Access.* To clarify and extend the previous discussion, we allow for `read` in SOCS-ORAM as follows. Recall that $\mathcal{A}$ is allowed to learn the access pattern, and hence she can be given π. $\mathcal{A}$ can then track the position of each element

in (extended) $[\![\mathbf{D}]\!]$ in a position map $\mathbf{pos}$, mapping logical indices $i \in [n]$ to physical indices $j \in [n+k]$. Initially $\mathbf{pos}_i := \pi_i \stackrel{\Delta}{=} \pi(i)$ for all $i \in [n]$. $\mathcal{A}$ uses $\mathbf{pos}$ at each access to find her share of the sought entry i at position $\mathbf{pos}_i$ in $[\![\mathbf{D}]\!]_{\mathcal{A}}$ (i.e. $[\![\mathbf{D}_{\mathbf{pos}_i}]\!]_{\mathcal{A}}$). Since π is a random permutation, $\mathcal{A}$ simply gives $\mathcal{B}$ $\mathbf{pos}_i$, and $\mathcal{B}$ retrieves his share $[\![\mathbf{D}_{\mathbf{pos}_i}]\!]_{\mathcal{B}}$. $\mathcal{A}$ and $\mathcal{B}$ can now use $\mathbf{D}_{\mathbf{pos}_i}$ inside MPC.

We now explain how to arrange that the read entry at logical index i, stored at physical index $\mathbf{D}_{\mathbf{pos}_i}$, is prepared for a subsequent access. Intuitively, after the qth access (out of total k provisioned), entry's value is copied to position π_{n+q}. This is done as follows. $\mathcal{A}$ arranges that $\mathbf{D}_{\pi_{n+q}} = \mathbf{D}_{\mathbf{pos}_i}$ *solely* by updating her share $[\![\mathbf{D}_{\pi_{n+q}}]\!]_{\mathcal{A}}$. $\mathcal{A}$ can do this because at initialization $\mathcal{C}$ will perform an additional step: He generates a k-element random mask vector $\mathbf{m}$ and secret shares it into the shelter positions $[\![\mathbf{D}_{\pi_i}]\!]$ (i.e. for $i \in [n, n+k]$). $\mathcal{C}$ sends $\mathbf{m}$ to $\mathcal{B}$. During the q-th access, where logical index i is read, $\mathcal{B}$ sends $[\![\mathbf{D}_{\mathbf{pos}_i}]\!]_{\mathcal{B}} \oplus \mathbf{m}_q$ to $\mathcal{A}$, who then XORs it with her share $[\![\mathbf{D}_{\mathbf{pos}_i}]\!]_{\mathcal{A}}$ and XORs the result into $[\![\mathbf{D}_{\pi_{n+q}}]\!]_{\mathcal{A}}$. It is easy to see that this arranges for a correct sharing of $\mathbf{D}_{\mathbf{pos}_i}$ in physical position $n+q$.

Finally, $\mathcal{A}$ updates her map $\mathbf{pos}_i := \pi_{n+q}$. Next access to logical index i is set up to be read from $\mathbf{D}_{\mathbf{pos}_i}$, a new and random-looking location for $\mathcal{B}$.

General `read`/`write` *Access* is an easy extension of `read`. For access, in addition to opcode $op = (\texttt{read}, \texttt{write})$ known to $\mathcal{A}$, both parties also input $[\![x]\!]$, a sharing of the element to be written. `write` differs from `read` only in that $[\![x]\!]$, and not $[\![\mathbf{D}_{\mathbf{pos}_i}]\!]$, is used to arrange $[\![\mathbf{D}_{\pi_{n+q}}]\!]$. This extension is simple to achieve with an oblivious transfer (OT), which we implement efficiently with correlated randomness provided by $\mathcal{C}$ during initialization. One pedantic nuance we must address is that `write` must return a value. We set it to be the value previously stored in that location.

Reinitializing SOCS-ORAM to provision for k more accesses is straightforward and reduces to invoking the initialization protocol. After the first k accesses, $\mathcal{A}$ and $\mathcal{B}$ hold a modified working array $[\![\mathbf{D}]\!]$. $[\![\mathbf{D}]\!]$ contains k outdated entries as each access moves the accessed element to a new shelter entry. The goal is now to transform $\mathbf{D}$ such that all k outdated entries are removed and only the remaining array of size n is reinitialized. Luckily, both $\mathcal{A}$ and $\mathcal{B}$ know which physical entries of $[\![\mathbf{D}]\!]$ are output in the first k accesses, and hence can locally remove them. Now, $|\mathbf{D}| = n$ and holds only the n entries of the latest $\mathbf{d}$.

We next invoke the initialization protocol on $[\![\mathbf{D}]\!]_{\mathcal{B}}$ (in a moment, we show how to convert initialized $[\![\mathbf{D}]\!]_{\mathcal{B}}$ to initialized $[\![\mathbf{D}]\!]$). As the input $[\![\mathbf{D}]\!]_{\mathcal{B}}$ is permuted, we reconcile the ($\mathcal{A}$-held) position map $\mathbf{pos}_{\text{prev}}$ from before the initialization call and $\mathbf{pos}$ output by the initialization. This is a straightforward combination of $\mathbf{pos}$ and $\mathbf{pos}_{\text{prev}}$, computed as $\mathbf{pos} := \mathbf{pos}_{\mathbf{pos}_{\text{prev}}}$.

Note, parties now hold the initialization of $[\![\mathbf{D}]\!]_{\mathcal{B}}$, not $[\![\mathbf{D}]\!]$. $\mathcal{A}$ locally resolves this by XORing each $[\![\mathbf{D}_{\text{prev},\mathbf{pos}_{\text{prev},i}}]\!]_{\mathcal{A}}$ ($[\![\mathbf{D}_{\text{prev}}]\!]_{\mathcal{A}}$ is $\mathcal{A}$'s share *before* the initialization) with the corresponding element $[\![\mathbf{D}_{\mathbf{pos}_i}]\!]_{\mathcal{A}}$ ($[\![\mathbf{D}]\!]_{\mathcal{A}}$ is $\mathcal{A}$'s share *output* by the initialization).

Reinitialization can be repeated arbitrary number of times; the procedure is identical after the first initialization and the later reinitializations.

Optimizing Communication via PRG Seeds. Secret sharing and sending randomness are central in our constructions. We optimize their required communication by sending and expanding PRG seeds. This must be done with care, especially in our setting where the output of the computation is the secret shares of the accessed array elements. Because the simulator of our ORAM is constrained by the *fixed* and externally-provided array shares (the output of the SOCS-ORAM functionality), we may not, for example, generate these shares from a PRG. We *are* able to use PRG to significantly reduce communication and achieve simulatability. For example, the helper server $\mathcal{C}$, who does not have any output, receives and expands PRG seed in its computation.

Note that this optimization is one of our key improvements over the original SOCS-ORAM in [1]. While we get significantly better performance, we are no longer unconditionally secure as we rely on a PRG. The original construction relies only on OT, which is performed without cryptographic assumptions with the help of $\mathcal{C}$.

6 Our SOCS-ORAM

We now formally present our scheme. In Section 6.1, we define SOCS-ORAM's cleartext semantics. In Section 6.2, we specify Π-SOCS-ORAM, our protocol implementing SOCS-ORAM, and present analytical costs. We prove Π-SOCS-ORAM correct and secure in Section 6.3.

6.1 Cleartext Semantics: SOCS-ORAM

Definition 2. *(Cleartext Semantics SOCS-ORAM) SOCS-ORAM$(\boldsymbol{d})_{e,k,l,n}$ is a 3-party stateful functionality executed between parties $\mathcal{A}$, $\mathcal{B}$, and $\mathcal{C}$ that consists of e (dynamically determined) epochs. Each epoch comprises a sequence of $k+1$ instructions except for the last epoch which has at most $k+1$ instructions. The first epoch is special and treated separately from the remaining epochs. Its first instruction is* `init`$(\boldsymbol{d})$*, where $\boldsymbol{d}$ is an array of n l-bit values and is input by $\mathcal{B}$.* `init` *sets $\boldsymbol{D} := \boldsymbol{d}$ to initialize the working array $\boldsymbol{D}$ with the input array $\boldsymbol{d}$ and provision for k instructions. The remaining k instructions are* `access`$_{\boldsymbol{D}}(op, i, [\![x]\!])$ *instructions. Unlike* `init`*, which is executed by $\mathcal{A}$, $\mathcal{B}$, and $\mathcal{C}$,* `access` *is executed between $\mathcal{A}$ and $\mathcal{B}$ only. $\mathcal{A}$ inputs op, i and both input $[\![x]\!]$. Depending on op, they read the value at index i or write $[\![x]\!]$ to the value at index i in $\boldsymbol{D}$. After an epoch is completed, $\mathcal{A}$ determines if another one is needed. $\mathcal{A}$ then requests k more accesses for the next epoch. Additional epochs differ from the first in that the first instruction is* `reinit()`*. Unlike* `init`*, which starts with array $\boldsymbol{d}$ input by $\mathcal{B}$,* `reinit` *starts with a working array $\boldsymbol{D}$ secret-shared between $\mathcal{A}$ and $\mathcal{B}$. With the help of $\mathcal{C}$, $\mathcal{A}$ updates it to provision for k additional* `access`$_{\boldsymbol{D}}$ *instructions. See Figure 1 for the* `init`*,* `reinit`*, and* `access` *functionalities.*

`init` **Functionality (3 Parties $\mathcal{A}$, $\mathcal{B}$, and $\mathcal{C}$)**

`init`($\mathbf{d}$):

- INPUT: Party $\mathcal{B}$ inputs an array $\mathbf{d}$ of length n s.t. $\mathbf{d}_i \in \{0,1\}^l$. $\mathcal{A}$ and $\mathcal{C}$ input $\perp$
- Set $\mathbf{D} := \mathbf{d}$ s.t. $\forall i \in [n]$, $\mathbf{D}_i := \mathbf{d}_i$ and $\mathbf{D}$ allows for k calls to `access`

`reinit` **Functionality (3 Parties $\mathcal{A}$, $\mathcal{B}$, and $\mathcal{C}$)**

`reinit`():

- INPUT: Parties $\mathcal{A}$, $\mathcal{B}$, and $\mathcal{C}$ input $\perp$.
- Update $\mathbf{D}$ s.t. it allows for k *additional* calls to `access`

`access` **Functionality (2 Parties $\mathcal{A}$ and $\mathcal{B}$)**

`read`(i):

- INPUT: Party $\mathcal{A}$ inputs an index $i \in [n]$. $\mathcal{B}$ inputs $\perp$
- OUTPUT: $\mathcal{A}$ and $\mathcal{B}$ output $[\![\mathbf{D}_i]\!]$

`write`($i, [\![x]\!]$):

- INPUT: Party $\mathcal{A}$ inputs an index $i \in [n]$. $\mathcal{A}$ and $\mathcal{B}$ input an element $[\![x]\!]$
- Set $out := \mathbf{D}_i$
- Set $\mathbf{D}_i := x$
- OUTPUT: $\mathcal{A}$ and $\mathcal{B}$ output $[\![out]\!]$

`access`($op, i, [\![x]\!]$):

- INPUT:
  - Party $\mathcal{A}$ inputs operation op (`read` or `write`) and an index $i \in [n]$
  - Parties $\mathcal{A}$ and $\mathcal{B}$ input additive sharing of an element $[\![x]\!]$
- OUTPUT:

$$\begin{cases} [\![\mathbf{D}_i]\!] \leftarrow \texttt{read}(i) & \text{if } op = \texttt{read} \\ [\![out]\!] \leftarrow \texttt{write}(i, [\![x]\!]) & \text{if } op = \texttt{write} \end{cases}$$

Fig. 1. The `init`, `reinit`, and `access` functionalities for our SOCS-ORAM. $\mathbf{d}$ is the input array used to initialize the SOCS-ORAM working array $\mathbf{D}$, which is then used for access and reinitialization.

Note that Figure 1 assumes each epoch (with the exception of the last one) has k `access` instructions. This is not a limitation of our scheme; our scheme supports arbitrary number of instructions in each epoch. We make this assumption for notational convenience.

6.2 Protocol: Π-SOCS-ORAM

In this section, we formalize our protocol Π-SOCS-ORAM and discuss optimization options based on using PRG expansion. We present a possible extension to Π-SOCS-ORAM that hides the operation *op* from $\mathcal{A}$. We show Π-SOCS-ORAM's analytical costs in Table 1.

Π-SOCS-ORAM securely implements the semantics of SOCS-ORAM (Definition 2):

Construction 1. *(Protocol Π-SOCS-ORAM) Π-SOCS-ORAM$(\boldsymbol{d})_{e,k,l,n}$ implements SOCS-ORAM by executing* `init` *with* Π-`init` *(Figure 2),* `access` *with* Π-`access` *(Figure 3), and* `reinit` *with* Π-`reinit` *(Figure 4).*

Theorems in Section 6.3 imply the following:

Theorem 1. *Construction 1 implements the functionality SOCS-ORAM (Definition 2) and is secure in the honest-majority semi-honest setting.*

As Π-SOCS-ORAM consists of separate invocations to Π-`init`, Π-`access`, and Π-`reinit` (see Construction 1), we describe Π-SOCS-ORAM by describing each of these procedures separately.

Π-`init`*.* Π-`init` sets up Π-SOCS-ORAM working data structures used by $\mathcal{A}$, $\mathcal{B}$ to `access` $\mathbf{d}$ (see `init` in Figure 1). It is a 3-party protocol, where $\mathcal{A}$, $\mathcal{B}$ are aided by helper $\mathcal{C}$.

$\mathcal{B}$ inputs array $\mathbf{d}$ of n l-bit entries, sets $\mathbf{D} := \mathbf{d}$, and secret shares $\mathbf{D}$ between $\mathcal{A}$ and $\mathcal{C}$: $\mathcal{A}$ receives $[\![\mathbf{D}]\!]_{\mathcal{A}}$; $\mathcal{C}$ receives a pseudo-random seed $\mathbf{sd}^{\mathcal{C}}$ that when input to a pseudo-random generator (PRG) expands to $[\![\mathbf{D}]\!]_{\mathcal{B}}$. $\mathcal{C}$ now helps construct the Π-SOCS-ORAM working data structures, used in Π-`access`, for $\mathcal{A}$ and $\mathcal{B}$.

$\mathcal{C}$ samples two pseudo-random seeds $\mathbf{sd}^{\mathcal{A}}$ (sent to $\mathcal{A}$) and $\mathbf{sd}^{\mathcal{B}}$ (sent to $\mathcal{B}$). $\mathcal{C}$ uses them to generate working data structures and help $\mathcal{A}$ and $\mathcal{B}$ obtain correlated randomness. $\mathcal{C}$ expands from $\mathbf{sd}^{\mathcal{A}}$ array $\mathbf{r}$ of the same size as $\mathbf{D}$. He masks the share $[\![\mathbf{D}]\!]_{\mathcal{B}}$ with $\mathbf{r}$, i.e. he computes $[\![\mathbf{D}]\!]_{\mathcal{B}} := [\![\mathbf{D}]\!]_{\mathcal{B}} \oplus \mathbf{r}$. Simultaneously, $\mathcal{C}$ expands from $\mathbf{sd}^{\mathcal{B}}$ array $\mathbf{m}$, which will hold shelter values, where array elements will be written once they are accessed. $\mathbf{m}$ has k l-bit entries, where k determines the maximum number of array accesses. $\mathcal{C}$ now secret-shares $\mathbf{m}$ by expanding $[\![\mathbf{m}]\!]_{\mathcal{A}}$ from $\mathbf{sd}^{\mathcal{A}}$ and computing $[\![\mathbf{m}]\!]_{\mathcal{B}} := \mathbf{m} \oplus [\![\mathbf{m}]\!]_{\mathcal{A}}$. Now $\mathcal{C}$ appends $[\![\mathbf{D}]\!]_{\mathcal{B}} := [\![\mathbf{D}]\!]_{\mathcal{B}} || [\![\mathbf{m}]\!]_{\mathcal{B}}$, draws a random permutation $\pi : [n+k] \to [n+k]$ from seed $\mathbf{sd}^{\mathcal{A}}$, and permutes $[\![\mathbf{D}]\!]_{\mathcal{B}}$ according to π. $\mathcal{C}$ also samples a random 1-out-of-2 OT, which will be converted into a chosen 1-out-of-2 OT using Beaver's trick [28] during `access`. This will help $\mathcal{A}$ obliviously retrieve the message corresponding to either the `read` or `write` operation. Namely, $\mathcal{C}$ expands from $\mathbf{sd}^{\mathcal{B}}$ two arrays $\mathbf{s}_0, \mathbf{s}_1$ of k l-bit entries. He also expands k-bit $\mathbf{f}$ from $\mathbf{sd}^{\mathcal{A}}$ and constructs $\mathbf{s}_f$ such that for all $i \in [k]$ it contains $\mathbf{s}_{0,i}$ or $\mathbf{s}_{1,i}$ depending on $\mathbf{f}_i$.

At this point, $\mathcal{C}$ generated the correlated randomness necessary to construct all working data structures for $\mathcal{A}$ and $\mathcal{B}$. He sends them this correlated randomness. First, $\mathcal{C}$ sends the seeds $\mathbf{sd}^{\mathcal{A}}$ to $\mathcal{A}$ and $\mathbf{sd}^{\mathcal{B}}$ to $\mathcal{B}$. These will help $\mathcal{A}$

locally reconstruct $\mathbf{r}$, $[\![\mathbf{m}]\!]_{\mathcal{A}}$, π, $\mathbf{f}$ and $\mathcal{B}$ reconstruct $\mathbf{m}$, $\mathbf{s}_0$, and $\mathbf{s}_1$. Additionally, $\mathcal{C}$ also sends to $\mathcal{A}$ and $\mathcal{B}$ the data structures they cannot reconstruct from the individual seeds. Namely, $\mathcal{A}$ receives $\mathbf{s}_f$ and $\mathcal{B}$ receives $[\![\mathbf{D}]\!]_{\mathcal{B}}$. Recall $[\![\mathbf{D}]\!]_{\mathcal{B}}$ was masked and permuted, and hence now looks random to $\mathcal{B}$.

Now both $\mathcal{A}$ and $\mathcal{B}$ have everything necessary to regenerate all working data structures. First, both parties set a counter $q := 0$ that counts the number of accesses. Then, $\mathcal{B}$ takes $[\![\mathbf{D}]\!]_{\mathcal{B}}$ along with the masks $\mathbf{m}$, $\mathbf{s}_0$, $\mathbf{s}_1$ and the counter q and stores them for `Π-access`. $\mathcal{A}$'s steps are a little more complicated. $\mathcal{A}$ first masks her share of $\mathbf{D}$ with $\mathbf{r}$, i.e. computes $[\![\mathbf{D}]\!]_{\mathcal{A}} := [\![\mathbf{D}]\!]_{\mathcal{A}} \oplus \mathbf{r}$, appends $[\![\mathbf{D}]\!]_{\mathcal{A}} := [\![\mathbf{D}]\!]_{\mathcal{A}} || [\![\mathbf{m}]\!]_{\mathcal{A}}$, and permutes $[\![\mathbf{D}]\!]_{\mathcal{A}}$ according to π. Then she computes a position map $\mathbf{pos}$ that tracks the position of the original n entries across accesses by setting $\mathbf{pos}_i := \pi_i$ for all $i \in [n]$. $\mathcal{A}$ then stores $[\![\mathbf{D}]\!]_{\mathcal{A}}, [\![\mathbf{m}]\!]_{\mathcal{A}}, \pi, \mathbf{pos}, \mathbf{f}, \mathbf{s}_f$, and q for `Π-access`.

Optimizing `Π-init` *by Sending Randomness via Seeds.* Our `Π-init` extensively relies on sending short κ-bit pseudo-random seeds across parties, who locally expand them with a pseudo-random generator (PRG) to construct intermediary Π-SOCS-ORAM arrays. This saves a large amount of communication. Sending the full arrays across the network for array $\mathbf{d}$ of n l-bit entries and k accesses would require that `Π-init` communicates $4n + 6k$ l-bit array entries, k bits, and a permutation (transferred as a table of length $n + k$). With our seed-based optimization, we reduce communication to $2n + 2k$ l-bit array entries and 3κ bits.

As discussed in Section 5, sending randomness and secret shares via PRG is difficult in our functionality. A subtle technical issue here is that the output of Π-SOCS-ORAM is *shares* of the returned values. Because shares are explicit output of the parties, simulating above optimized protocol would require that the PRG output matches the fixed shares of the output. This can be solved by using programmable primitives (such as programmable random oracle), or considering the complete MPC problem, where the wire shares are not part of the output. Instead, as outlined in Section 5, we arrange our protocol so that using expanded PRG seeds for shares is simulatable.

One other important aspect of this optimization is that we no longer send the permutation π across the network from $\mathcal{C}$ to $\mathcal{A}$. I.e., the permutation can be regenerated on $\mathcal{A}$ from the seed $\mathbf{sd}^{\mathcal{A}}$ instead of being sent as a table of size $(n+k)\log(n+k)$ bits. While we save communication, generating a permutation is one of the bottlenecks of Π-SOCS-ORAM. Depending on the network settings, it can be preferable to send π as a table. Another option is to make system optimizations to ensure the seed is sent to $\mathcal{A}$ as soon as $\mathcal{C}$ computes it so that the permutation can be generated in parallel.

`Π-access`*.* `Π-access` implements `access` (see Figure 1). It is a 2-party protocol, run between $\mathcal{A}$ and $\mathcal{B}$, where $\mathcal{A}$ requests `read` or `write` to working array $\mathbf{D}$.

Recall that $\mathcal{A}$ inputs logical index i and operation *op* (`read` or `write`). Both input a sharing $[\![x]\!]$. $[\![x]\!]$ is input even if $op =$ `read` because $\mathcal{B}$ cannot learn *op*.

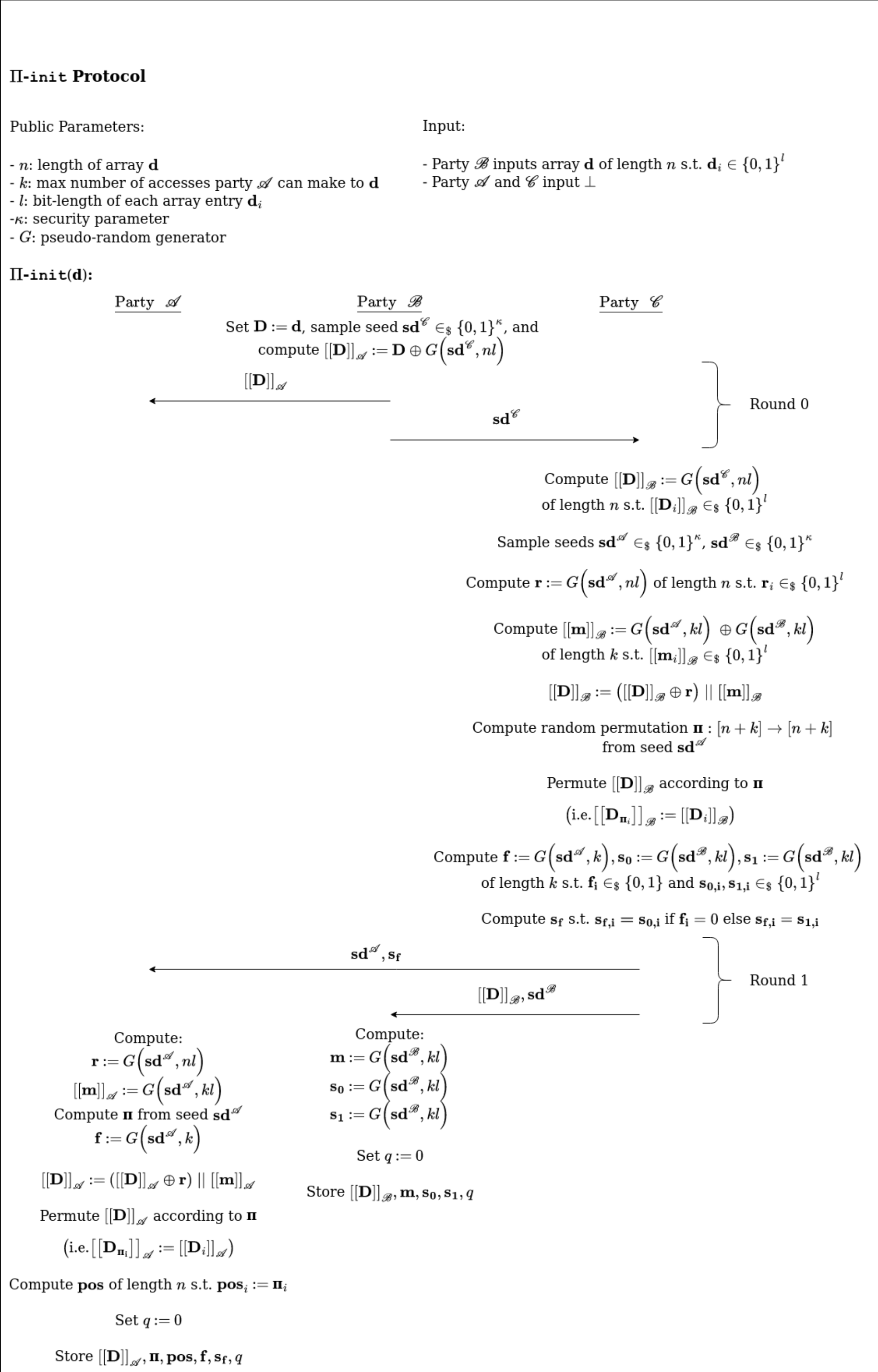


Fig. 2. Π-init is a subroutine of Π-SOCS-ORAM.

$\mathcal{A}$ retrieves $\mathbf{pos}_i$, which represents the physical location of i in the shuffled $\mathbf{D}$, and computes bit $b := \mathbf{f}_q \oplus op$, which will help $\mathcal{A}$ select $\mathcal{B}$'s message corresponding to op. She sends $\mathbf{pos}_i, b$ to $\mathcal{B}$. $\mathcal{A}$ also retrieves her read share $[\![\mathbf{D}_{\mathbf{pos}_i}]\!]_\mathcal{A}$; her write share $[\![x]\!]_\mathcal{A}$ is input to `Π-access`.

$\mathcal{B}$ now constructs two messages: the first is for $op =$ `read` and the latter for $op =$ `write`. For $op =$ `read`, $\mathcal{B}$ retrieves his read share $[\![\mathbf{D}_{\mathbf{pos}_i}]\!]_\mathcal{B}$. For $op =$ `write`, he already holds his write share $[\![x]\!]_\mathcal{B}$ from the `Π-access` input. He cannot send his shares to $\mathcal{A}$ for security, and thus masks each with $\mathbf{m}_q$. $\mathcal{A}$ is only supposed to learn (i.e. unmask) one of these messages and so $\mathcal{B}$ adds another mask. I.e., he adds $\mathbf{s}_{\mathbf{b},q}$ to the `read` message and $\mathbf{s}_{\overline{\mathbf{b}},q}$ to the `write` message. Recall $\mathcal{A}$ holds only one of $\mathbf{s}_{\mathbf{b},q}$ and $\mathbf{s}_{\overline{\mathbf{b}},q}$, and hence will be able to remove the mask only from one of the messages. Then he sends both messages to $\mathcal{A}$.

$\mathcal{A}$ now selects the message corresponding to op and adds $\mathbf{s}_{\mathbf{f},q}$ to unmask it. She then adds its `read` (or `write`) share along with the unmasked message to the next free shelter position $[\![\mathbf{D}_{\pi_{n+q}}]\!]_\mathcal{A}$.

$\mathcal{A}$ and $\mathcal{B}$ now set their output share $[\![out]\!] := [\![\mathbf{D}_{\mathbf{pos}_i}]\!]$. $\mathcal{A}$ updates the position map such that i points to the assigned shelter entry $\mathbf{pos}_i := \pi_{n+q}$. Then both increment access counter $q \mathrel{+}= 1$ and output $[\![out]\!]$.

`Π-reinit` . `Π-reinit` (Figure 4) implements `reinit` defined in Figure 1. It is a 3-party protocol, run between $\mathcal{A}$, $\mathcal{B}$, and $\mathcal{C}$, and provisions for k additional accesses to $\mathbf{d}$. There is no restriction on how many times we can invoke `reinit`. The protocol is the same after the first epoch, which starts with `Π-init`, and after the following epochs, which start with `Π-reinit`.

Recall that at the end of each epoch $\mathcal{A}$ and $\mathcal{B}$ hold a secret-shared working array $\mathbf{D}$ of size $n + k$. k of these elements are inoperative (i.e. no element in the position map $\mathbf{pos}$ points to them) as they were accessed during the epoch and their values were moved to the shelter. Let $\mathbf{idx}$ represent the physical indices of the elements accessed in $\mathbf{D}$ at this epoch. $\mathcal{A}$ and $\mathcal{B}$ remove $\mathbf{D}_{\mathbf{idx}_i}$, $\forall i \in [k]$, from their respective shares of $\mathbf{D}$. Now they hold $\mathbf{D}$ of size n. Since the inoperative elements are removed, $\mathcal{A}$'s position map $\mathbf{pos}$ needs to be adjusted to account for the removed elements. This is a simple local step done by $\mathcal{A}$.

Next, $\mathcal{A}$ and $\mathcal{B}$ save the updated $\mathbf{D}$, $\mathbf{pos}$ arrays in $\mathbf{D}_{\text{prev}}$, $\mathbf{pos}_{\text{prev}}$, respectively. They treat $[\![\mathbf{D}_{\text{prev}}]\!]_\mathcal{B}$ as a new input array and invoke `Π-init` on it. This returns new $\mathbf{D}$, $\mathbf{pos}$ arrays, separate from $\mathbf{D}_{\text{prev}}$, $\mathbf{pos}_{\text{prev}}$. Parties then can use these arrays to complete the reinitialization of $\mathbf{D}$ as follows. Note, the elements in the input to the `Π-init` call are already permuted, and their order is saved in $\mathbf{pos}_{\text{prev}}$. Thus $\mathcal{A}$ needs to update $\mathbf{pos}$ (returned by `Π-init`) such that it accounts for the initial order $\mathbf{pos}_{\text{prev}}$ of the input to `Π-init`. To do this $\mathcal{A}$ simply composes the two position arrays by setting each $\mathbf{pos}_i$ to $\mathbf{pos}_{\mathbf{pos}_{\text{prev},i}}$. Additionally, recall we invoke initialization only for $[\![\mathbf{D}_{\text{prev}}]\!]_\mathcal{B}$, not $[\![\mathbf{D}_{\text{prev}}]\!]$. Hence, $\mathcal{A}$ needs to add her $[\![\mathbf{D}_{\text{prev}}]\!]_\mathcal{A}$ into the corresponding positions in $[\![\mathbf{D}]\!]_\mathcal{A}$. As $\mathcal{A}$ holds both $\mathbf{pos}_{\text{prev}}$ and the new and updated $\mathbf{pos}$, this is straightforward. $\mathcal{A}$ retrieves, $\forall i \in [n]$, $[\![\mathbf{D}_{\mathbf{pos}_i}]\!]_\mathcal{A}$ and XORs $[\![\mathbf{D}_{\text{prev},\mathbf{pos}_{\text{prev},i}}]\!]_\mathcal{A}$ into it.

Π-access **Protocol**

PARAMETERS (from Π-init):

- Parties $\mathcal{A}$ and $\mathcal{B}$ hold an array $[\![\mathbf{D}]\!]$ (processed in Π-init) of $(n+k)$ l-bit entries
- $\mathcal{A}$ and $\mathcal{B}$ access at most k elements; $q \in [k]$ is the current access number
- $\mathcal{A}$ holds position map $\mathbf{pos}$ of length n
- $\mathcal{A}$ holds a random permutation $\pi : [n+k] \to [n+k]$
- $\mathcal{B}$ holds two random arrays $\mathbf{s_0}, \mathbf{s_1}$ of k l-bit masks
- $\mathcal{A}$ holds random k-bit array $\mathbf{f}$ and array $\mathbf{s_f}$ of k l-bit masks
- $\mathcal{B}$ holds array $\mathbf{m}$ of k l-bit masks s.t. $\mathbf{m}_q := \mathbf{D}_{\pi_{n+q}}$

INPUT:

- $\mathcal{A}$ inputs op (`read` or `write`) and i s.t. $i \in [n]$; $\mathcal{A}$ and $\mathcal{B}$ input $[\![x]\!]$

Π-access$(op, i, [\![x]\!])$:

$\mathcal{A}$ sets $b := \mathbf{f}_q \oplus op$

$\mathcal{A}$ sends $\mathbf{pos}_i$, b to $\mathcal{B}$

$\mathcal{B}$ sets:

$$\begin{cases} md_0 := \mathbf{m}_q \oplus [\![\mathbf{D}_{\mathbf{pos}_i}]\!]_{\mathcal{B}} & \text{if } op = \texttt{read} \quad \text{// } \mathbf{m}_q \text{ masks } [\![\mathbf{D}_{\mathbf{pos}_i}]\!]_{\mathcal{B}}.\ \mathcal{A} \text{ cannot learn } [\![\mathbf{D}_{\mathbf{pos}_i}]\!]_{\mathcal{B}} \\ md_1 := \mathbf{m}_q \oplus [\![x]\!]_{\mathcal{B}} & \text{if } op = \texttt{write} \quad \text{// Similarly, } \mathbf{m}_q \text{ masks } [\![x]\!]_{\mathcal{B}} \end{cases}$$

$\mathcal{B}$ sets: // This step ensures $\mathcal{A}$ learns only the message corresponding to op

$$\begin{cases} ms_0 := md_0 \oplus \mathbf{s}_{\mathbf{b},q} \\ ms_1 := md_1 \oplus \mathbf{s}_{\overline{\mathbf{b}},q} \end{cases}$$

$\mathcal{B}$ sends ms_0, ms_1 to $\mathcal{A}$

$\mathcal{A}$ unmasks exactly one of md_0 or md_1 depending on op:

$$md_{op} := \mathbf{s}_{\mathbf{f},q} \oplus \begin{cases} ms_0 & \text{if } op = \texttt{read} \\ ms_1 & \text{if } op = \texttt{write} \end{cases}$$

$\mathcal{A}$ sets:

$$tmp := md_{op} \oplus \begin{cases} [\![\mathbf{D}_{\mathbf{pos}_i}]\!]_{\mathcal{A}} & \text{if } op = \texttt{read} \quad \text{// } tmp = \mathbf{D}_{\mathbf{pos}_i} \oplus \mathbf{m}_q \\ [\![x]\!]_{\mathcal{A}} & \text{if } op = \texttt{write} \quad \text{// } tmp = x \oplus \mathbf{m}_q \end{cases}$$

$\mathcal{A}$ sets $[\![\mathbf{D}_{\pi_{n+q}}]\!]_{\mathcal{A}} := [\![\mathbf{D}_{\pi_{n+q}}]\!]_{\mathcal{A}} \oplus tmp$ // $\mathbf{D}_{\pi_{n+q}}$ now holds not permuted $\mathbf{D}_i$ (or x)

$\mathcal{A}$ and $\mathcal{B}$ set $[\![out]\!] := [\![\mathbf{D}_{\mathbf{pos}_i}]\!]$

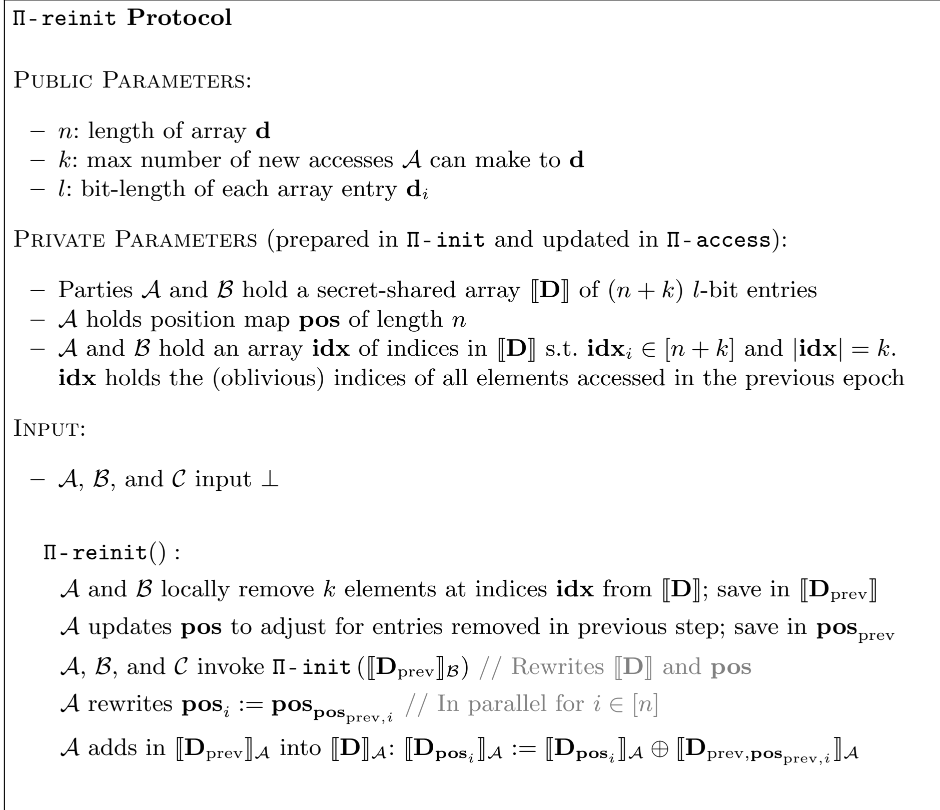
$\mathcal{A}$ sets $\mathbf{pos}(i) := \pi_{n+q}$ // π_{n+q} is the new location of not permuted $\mathbf{D}_i$ (or x)

$\mathcal{A}$ and $\mathcal{B}$ increment $q \mathrel{+}= 1$

$\mathcal{A}$ and $\mathcal{B}$ return $[\![out]\!]$

Fig. 3. Π-access is a subroutine of Π-SOCS-ORAM.

Hiding Opcode op from $\mathcal{A}$. We sketch here, but do not formalize in a protocol, an optional extension of SOCS-ORAM that hides from $\mathcal{A}$ whether op is a `read` or a



Π-`reinit` Protocol

Public Parameters:

- n: length of array $\mathbf{d}$
- k: max number of new accesses $\mathcal{A}$ can make to $\mathbf{d}$
- l: bit-length of each array entry $\mathbf{d}_i$

Private Parameters (prepared in Π-`init` and updated in Π-`access`):

- Parties $\mathcal{A}$ and $\mathcal{B}$ hold a secret-shared array $[\![\mathbf{D}]\!]$ of $(n+k)$ l-bit entries
- $\mathcal{A}$ holds position map $\mathbf{pos}$ of length n
- $\mathcal{A}$ and $\mathcal{B}$ hold an array $\mathbf{idx}$ of indices in $[\![\mathbf{D}]\!]$ s.t. $\mathbf{idx}_i \in [n+k]$ and $|\mathbf{idx}| = k$. $\mathbf{idx}$ holds the (oblivious) indices of all elements accessed in the previous epoch

Input:

- $\mathcal{A}$, $\mathcal{B}$, and $\mathcal{C}$ input $\perp$

Π-`reinit`() :

$\mathcal{A}$ and $\mathcal{B}$ locally remove k elements at indices $\mathbf{idx}$ from $[\![\mathbf{D}]\!]$; save in $[\![\mathbf{D}_{\text{prev}}]\!]$

$\mathcal{A}$ updates $\mathbf{pos}$ to adjust for entries removed in previous step; save in $\mathbf{pos}_{\text{prev}}$

$\mathcal{A}$, $\mathcal{B}$, and $\mathcal{C}$ invoke Π-`init` $([\![\mathbf{D}_{\text{prev}}]\!]_{\mathcal{B}})$ // Rewrites $[\![\mathbf{D}]\!]$ and $\mathbf{pos}$

$\mathcal{A}$ rewrites $\mathbf{pos}_i := \mathbf{pos}_{\mathbf{pos}_{\text{prev},i}}$ // In parallel for $i \in [n]$

$\mathcal{A}$ adds in $[\![\mathbf{D}_{\text{prev}}]\!]_{\mathcal{A}}$ into $[\![\mathbf{D}]\!]_{\mathcal{A}}$: $[\![\mathbf{D}_{\mathbf{pos}_i}]\!]_{\mathcal{A}} := [\![\mathbf{D}_{\mathbf{pos}_i}]\!]_{\mathcal{A}} \oplus [\![\mathbf{D}_{\text{prev},\mathbf{pos}_{\text{prev},i}}]\!]_{\mathcal{A}}$

Fig. 4. Π-`reinit` is a subroutine of Π-SOCS-ORAM.

`write`. That is, *op* is secret-shared between $\mathcal{A}$ and $\mathcal{B}$, $\mathcal{A}$ still learns the positions of all accessed elements, but does not learn whether they are read or new values are written. Access without opcode hiding runs in 2 sequential communication flows (i.e. 1 round trip) and communicates 2 array elements, a single entry in a position map, and a single bit. By adding 1 communication flow and additionally communicating 2 array elements and a bit, we can achieve `access` while hiding *op* from $\mathcal{A}$ as well as $\mathcal{B}$.

This extension is simple to achieve with one additional OT, which is cheap with $\mathcal{C}$'s help during initialization. At a high level, $\mathcal{A}$ and $\mathcal{B}$ use the first OT to deliver to $\mathcal{B}$ either $\mathcal{A}$'s read or write share *masked by a random mask c*. This can be achieved by the classical OT trick where the sender $\mathcal{A}$ permutes her inputs by $[\![op]\!]_{\mathcal{A}}$ and the receiver $\mathcal{B}$ asks for $[\![op]\!]_{\mathcal{B}}$. $\mathcal{B}$ adds the received value to both inputs of the next OT, which is already part of our protocol. This OT delivers to $\mathcal{A}$ the `read` or the `write` value masked by the shelter element $\mathbf{m}_q$ *and additionally c*. $\mathcal{A}$ next removes c and proceeds as in the current protocol.

Algorithm	Comm. (bits)	# Comm. Flows	Comm. Depth
`Π-init`	$2nl + 2kl + 3\kappa$	4	2
`Π-reinit`	$2nl + 2kl + 3\kappa$	4	2
`Π-access`	$2l + \log(n + k) + 1$	2	2

Table 1. Total communication cost across all parties, number of flows, and communication depth (number of consecutive flows). The costs are for input array $\mathbf{d}$ of size n s.t. $\mathbf{d}_i \in \{0,1\}^l$ and we provision for k accesses in each `Π-init`/ `Π-reinit`.

Cost Analysis. We calculate the costs of Π-SOCS-ORAM in Table 1. We evaluate each `Π-init`, `Π-reinit`, and `Π-access` separately and express the costs in terms of total communication (# bits) across all parties, number of communication flows (e.g. $\mathcal{A} \rightarrow \mathcal{B}$ is a single flow), and communication depth (number of consecutive flows). Note that the communication cost of `Π-reinit` is the same as the cost of `Π-init`. This is because all operations of `Π-reinit` are local operations apart from the invocation of `Π-init`. We experimentally show in Section 7 that the `Π-access` wall-clock time is almost fully due to latency, as is expected based on its low computational and communication complexity.

6.3 Π-SOCS-ORAM Proofs

Now that we introduced Π-SOCS-ORAM, we prove it correct and secure.

Proof of Correctness

Π-SOCS-ORAM implements the functionality SOCS-ORAM (Definition 2):

Theorem 2 (Π-SOCS-ORAM Correctness). *Let $e, k, l, n \in \mathbb{N}$. Let $\mathcal{D}$ be a space of all arrays with n l-bit entries. For all arrays $\boldsymbol{d} \in \mathcal{D}$, for all number of epochs e with $k+1$ instructions (except for the last epoch with at most $k+1$ instructions), with the first epoch starting with* `init`/ `Π-init` *and the subsequent epochs starting with* `reinit`/ `Π-reinit`*, followed by* `access`/ `Π-access` *instructions not necessarily known a priori:*

$$\mathsf{SOCS\text{-}ORAM}_{e,k,l,n}(\boldsymbol{d}) = \Pi\text{-}\mathsf{SOCS\text{-}ORAM}_{e,k,l,n}(\boldsymbol{d})$$

Proof. We prove Π-SOCS-ORAM correct by inspection. We show that `Π-init` establishes a valid XOR sharing of each element in the input array $\mathbf{d}$ that `Π-access` then uses for access. Importantly, `Π-access` maintains a valid XOR sharing when it writes back the same read element ($op =$ `read`) or a new element ($op =$ `write`). The same applies to `Π-reinit` when we set up the working array $\mathbf{D}$ for k more accesses.

We start by showing that `Π-init` establishes a valid XOR sharing of all $\mathbf{d}_i$. `Π-init` first sets $\mathbf{D} := \mathbf{d}$ and secret shares $\mathbf{D}$. Each share is then XORed with the same mask $\mathbf{r}$, which does not change $\mathbf{D}$. $[\![\mathbf{D}]\!]$ is next extended with a

secret-shared shelter $[\![\mathbf{m}]\!]$, which also does not change any original element of $\mathbf{D}$. Next, $[\![\mathbf{D}]\!]$ is permuted according to the *same* random permutation π. Hence, elements in both shares are shifted to a same new position. Therefore, $\mathbf{D}_i$ before permutation equals $\mathbf{D}_{\pi_i}$ after the elements are permuted.

We now show that `Π-access` can use the initialized $[\![\mathbf{D}]\!]$ to access a valid XOR sharing. As $\mathcal{A}$ knows π, she knows the position of each element in $[\![\mathbf{D}]\!]$. She can share this position with $\mathcal{B}$, and they both retrieve a correct share.

So far, we have shown that we retrieve a valid XOR sharing only the *first time any index is accessed*. We also need to show that we maintain a valid XOR sharing in repeated accesses to the same element. After each access, the retrieved element (or a new element if $op = \texttt{write}$) get assigned to next available position in the shelter, which was added and permuted within $[\![\mathbf{D}]\!]$ during `Π-init`. $\mathcal{A}$'s knowledge of π implies the knowledge of position of all shelter elements. What we need to show is that we maintain correct $[\![\mathbf{D}_{\mathbf{pos}_i}]\!]$ if $op = \texttt{read}$ and write $[\![x]\!]$ if $op = \texttt{write}$.

Recall q represents the access number and $\mathbf{pos}$ the position map that tracks the location of each array element $\mathbf{d}_i$ (initially $\mathbf{pos}_i := \pi_i$). In `Π-init`, shelter is set to $[\![\mathbf{m}]\!]$. The next available shelter entry at π_{n+q} must be updated after each access to contain $[\![\mathbf{D}_{\mathbf{pos}_i}]\!]$ or $[\![x]\!]$ (depending on op). Recall that during `Π-init` $\mathcal{B}$ is given masks $\mathbf{m}$. $\mathcal{B}$ takes $\mathbf{m}_q$ and constructs two messages by masking both the `read` share $[\![\mathbf{D}_{\mathbf{pos}_i}]\!]_{\mathcal{B}}$ and the `write` share $[\![x]\!]$ with $\mathbf{m}_q$. I.e., he computes $\mathbf{m}_q \oplus [\![\mathbf{D}_{\mathbf{pos}_i}]\!]_{\mathcal{B}}$ and $\mathbf{m}_q \oplus [\![x]\!]_{\mathcal{B}}$, respectively. He needs to take one of these messages corresponding to the operation op and send it to $\mathcal{A}$. Now, assume that $\mathcal{B}$ knows which message to send; we will handle the case when $\mathcal{B}$ does not know op later. $\mathcal{A}$ receives the masked share and adds her own share. I.e., $\mathcal{A}$ holds $\mathbf{m}_q \oplus [\![\mathbf{D}_{\mathbf{pos}_i}]\!]_{\mathcal{A}} \oplus [\![\mathbf{D}_{\mathbf{pos}_i}]\!]_{\mathcal{B}} = \mathbf{m}_q \oplus \mathbf{D}_{\mathbf{pos}_i}$ if $op = \texttt{read}$, and $\mathbf{m}_q \oplus [\![x]\!]_{\mathcal{A}} \oplus [\![x]\!]_{\mathcal{B}} = \mathbf{m}_q \oplus x$ if $op = \texttt{write}$. As the shelter currently holds $[\![\mathbf{m}_q]\!]$, adding these messages into $\mathcal{A}$'s share of the shelter cancels out $\mathbf{m}_q$, leaving $[\![\mathbf{D}_{\mathbf{pos}_i}]\!]$ and $[\![x]\!]$, respectively.

In the real execution $\mathcal{B}$ does not hold op, and hence does not know which message to send. We now show that our technique sends the correct message corresponding to op to $\mathcal{A}$. Note that this is the classical trick for converting random OTs to chosen OTs due to Beaver [28]. In `Π-init`, $\mathcal{B}$ receives two masks $\mathbf{s}_{\mathbf{0},q}$ and $\mathbf{s}_{\mathbf{1},q}$. $\mathcal{A}$ receives only one of those masks $\mathbf{s}_{\mathbf{f},q}$ depending on a random bit $\mathbf{f}_q$. I.e. $\mathcal{A}$ and $\mathcal{B}$ execute random OT with $\mathcal{C}$'s help in `Π-init`. Now during access they transform the random OT into chosen OT. The key idea is to give some information to $\mathcal{B}$ that will allow him to mask the message corresponding to op with $\mathbf{s}_{\mathbf{f},q}$, which $\mathcal{A}$ can then remove. $\mathcal{A}$ sends $\mathbf{f}_q \oplus op$ to $\mathcal{B}$. $\mathcal{B}$ then masks the first (`read`) message with $\mathbf{s}_{\mathbf{f}_q \oplus op,q}$ and the second (`write`) message with $\mathbf{s}_{\overline{\mathbf{f}_q \oplus op},q}$. If $op = \texttt{read}$, then the `read` message is masked with $\mathbf{s}_{\mathbf{f}_q \oplus op,q} = \mathbf{s}_{\mathbf{f}_q \oplus 0,q} = \mathbf{s}_{\mathbf{f},q}$, which $\mathcal{A}$ can remove. If $op = \texttt{write}$, then the `write` message is masked with $\mathbf{s}_{\overline{\mathbf{f}_q \oplus op},q} = \mathbf{s}_{\overline{\mathbf{f}_q \oplus 1},q} = \mathbf{s}_{\overline{\overline{\mathbf{f}_q}},q} = \mathbf{s}_{\mathbf{f},q}$, which $\mathcal{A}$ can also remove. $\mathcal{A}$ receives the right message, uses it to update the next available element in the shelter, and maintains a valid XOR sharing for both `read` and `write`.

We now show that `Π-reinit` re-establishes a valid XOR sharing for the next k accesses. We have already shown that at the end of the previous k accesses, $\mathcal{A}$

and $\mathcal{B}$ hold a valid sharing $[\![\mathbf{D}]\!]$. As a result of those k accesses, $[\![\mathbf{D}]\!]$ contains k outdated entries as each accessed element is moved to a next available position in the shelter. The parties remove these entries and then invoke Π-init on $[\![\mathbf{D}]\!]_{\mathcal{B}}$ of size n. As we showed earlier, Π-init establishes a valid XOR sharing of its input. Hence, the output is a valid XOR sharing of $[\![\mathbf{D}]\!]_{\mathcal{B}}$. Let $[\![\mathbf{D}_{\text{prev}}]\!]$ now refer to the sharing held at the end of the previous epoch and $[\![\mathbf{D}]\!]$ the sharing output by Π-init (i.e. Π-init is invoked on $[\![\mathbf{D}_{\text{prev}}]\!]_{\mathcal{B}}$). If $\mathcal{A}$ can XOR her share $[\![\mathbf{D}_{\text{prev}}]\!]_{\mathcal{A}}$ into her Π-init's output $[\![\mathbf{D}]\!]_{\mathcal{A}}$, $\mathcal{A}$ and $\mathcal{B}$ will hold a valid XOR sharing $[\![\mathbf{D}]\!]$. Fortunately, $\mathcal{A}$ holds the position maps $\mathbf{pos}_{\text{prev}}$ for the previous epoch (note that we adjust $\mathbf{pos}_{\text{prev}}$ in Figure 4 because we remove the k outdated elements) and $\mathbf{pos}$ for the new epoch. The new $\mathbf{pos}$ is incomplete as it does not take into account that $[\![\mathbf{D}_{\text{prev}}]\!]_{\mathcal{B}}$ input to Π-init is ordered according to $\mathbf{pos}_{\text{prev}}$. $\mathcal{A}$ thus needs to reconcile the two maps to obtain the correct $\mathbf{pos}$ for the new epoch. I.e., she computes $\mathbf{pos}_i := \mathbf{pos}_{\mathbf{pos}_{\text{prev},i}}$. With the reconciled $\mathbf{pos}$ and $\mathbf{pos}_{\text{prev}}$, $\mathcal{A}$ simply XORs $[\![\mathbf{D}_{\text{prev},\mathbf{pos}_{\text{prev},i}}]\!]_{\mathcal{A}}$ into $[\![\mathbf{D}_{\mathbf{pos}_i}]\!]_{\mathcal{A}}$ for $\mathcal{A}$ & $\mathcal{B}$ to get a valid sharing $[\![\mathbf{D}]\!]$.

Π-SOCS-ORAM is correct.

□

Proof of Security

We now prove Π-SOCS-ORAM secure.

Theorem 3 (Π-SOCS-ORAM Security). *Π-SOCS-ORAM is secure against semi-honest corruption of one party.*

Proof. By construction of 3 simulators $\mathcal{S}_{\mathcal{A}}$, $\mathcal{S}_{\mathcal{B}}$, and $\mathcal{S}_{\mathcal{C}}$ that simulate the view of each party $\mathcal{A}$, $\mathcal{B}$, and $\mathcal{C}$, and an argument that the joint distribution of each simulator's output and SOCS-ORAM's output is indistinguishable from that party's real view and Π-SOCS-ORAM's output. A key observation is that all messages, except inputs and outputs belonging to each party, are indistinguishable from uniform bits.

We first construct $\mathcal{S}_{\mathcal{A}}(\mathbf{op}, \mathbf{i}, [\![\mathbf{x}]\!]_{\mathcal{A}}, [\![\mathbf{out}]\!]_{\mathcal{A}})$ that for each access gets operation op, index i, value $[\![x]\!]_{\mathcal{A}}$, and output $[\![out]\!]_{\mathcal{A}}$. We start by showing how $\mathcal{S}_{\mathcal{A}}$ simulates the first epoch (i.e. the first k accesses) and then show how she simulates each following epoch. $\mathcal{S}_{\mathcal{A}}$ now simulates $\mathcal{A}$'s view.

Simulating $\mathcal{A}$'s view in the *first* epoch:

- Consider Π-init. $\mathcal{A}$ receives an XOR share $[\![\mathbf{D}]\!]_{\mathcal{A}}$ from $\mathcal{B}$. From $\mathcal{C}$, $\mathcal{A}$ also receives a seed $\mathbf{sd}^{\mathcal{A}}$, which she uses to generate $\mathbf{r}, [\![\mathbf{m}]\!]_{\mathcal{A}}, \pi$, & $\mathbf{f}$, and she receives $\mathbf{s_f}$, which helps her unmask the message corresponding to op upon access. Note that all are indistinguishable from random bits. $[\![\mathbf{D}]\!]_{\mathcal{A}}$ is generated by $\mathcal{B}$ uniformly secret sharing $\mathbf{D}$. $\mathbf{sd}^{\mathcal{A}}$ is a seed uniformly sampled

by $\mathcal{C}$. Depending on $\mathbf{f}$, $\mathbf{s_f}$ has entries from $\mathbf{s_0}$ or $\mathbf{s_1}$, which are both derived from a uniformly sampled seed unknown to $\mathcal{A}$. Thus, $\mathbf{sd}^{\mathcal{A}}$ and $\mathbf{s_f}$ can be simulated by uniformly drawing bits; $[\![\mathbf{D}]\!]_{\mathcal{A}}$ is more complex to simulate as it must result in $[\![\mathbf{D}]\!]_{\mathcal{A}}$ that is consistent with Π-SOCS-ORAM's output. We thus need to simulate it alongside the calls to `Π-access`.

– Consider the first k calls to `Π-access`. Note that $\mathcal{S}_{\mathcal{A}}$ gets a list of indices as input. Thus, $\mathcal{S}_{\mathcal{A}}$ knows which k entries of $[\![\mathbf{D}]\!]_{\mathcal{A}}$ (combined with the shelter $[\![\mathbf{m}]\!]_{\mathcal{A}}$) must match the protocol output. She goes through these entries one by one, taking care that the simulation is consistent with the output.

  Before going through these entries, observe that at each access, $\mathcal{A}$ receives a 2-part message ms_0 and ms_1 from $\mathcal{B}$. Both parts are masked by $\mathbf{m}_q$, which is derived from a uniformly drawn seed $\mathbf{sd}^{\mathcal{B}}$ unknown to $\mathcal{A}$. Additionally, ms_0 is masked by $\mathbf{s}_{\mathbf{b},q}$, ms_1 by $\mathbf{s}_{\bar{\mathbf{b}},q}$, which are both generated from a uniform seed $\mathbf{sd}^{\mathcal{B}}$ only known to $\mathcal{B}$, and $\mathcal{A}$ knows *exactly one* of them. Note that both messages look uniform to $\mathcal{A}$ because even after removing $\mathbf{s}_{\mathbf{f},q}$ (i.e. $\mathbf{s}_{\mathbf{b},q}$ or $\mathbf{s}_{\bar{\mathbf{b}},q}$) from one of the messages, the message is still masked with $\mathbf{m}_q$. Similarly to $[\![\mathbf{D}]\!]_{\mathcal{A}}$, the simulation is not a straightforward sampling of uniform bits. This is because we add these messages into $[\![\mathbf{m}_q]\!]_{\mathcal{A}}$ ($\mathcal{A}$'s shelter), which is simulated by expanding a uniformly drawn seed. We must ensure to simulate the two messages such that the output is consistent with Π-SOCS-ORAM's output.

  Hence, we observe that all messages received by $\mathcal{A}$ during `Π-init` and `Π-access` are indistinguishable from random. We now go through the accesses one by one, simulate $[\![\mathbf{D}]\!]_{\mathcal{A}}$, and ms_0, ms_1 for each access:

  - Consider the first access index i. $\mathcal{S}_{\mathcal{A}}$ sets the entry at this index in $[\![\mathbf{D}]\!]_{\mathcal{A}}$ such that after adding $\mathbf{r}$, expanded from the simulated $\mathbf{sd}^{\mathcal{A}}$, the two XOR to the first Π-SOCS-ORAM output share. $\mathcal{S}_{\mathcal{A}}$ then checks if the same index is accessed again in the first epoch:
    - ∗ If yes, check if the *nearest* `access` (let it be j^{th} access) corresponds to a `read` or a `write` (recall that $\mathcal{S}_{\mathcal{A}}$ gets *op* as part of the protocol input). If it is a `read`, draw ms_1 uniformly at random and set ms_0 such that it results in the right output share, i.e., $ms_0 \oplus [\![\mathbf{D}_i]\!]_{\mathcal{A}} \oplus \mathbf{s}_{\mathbf{f},0} \oplus [\![\mathbf{m}_0]\!]_{\mathcal{A}} = [\![\mathbf{out}_j]\!]_{\mathcal{A}}$ at the next access when added to $[\![\mathbf{D}_i]\!]_{\mathcal{A}} \oplus ms_0 \oplus \mathbf{s}_{\mathbf{f},q}$. If it is a `write`, draw ms_0 uniformly at random and set ms_1 similarly such that $ms_1 \oplus [\![x]\!]_{\mathcal{A}} \oplus \mathbf{s}_{\mathbf{f},0} \oplus [\![\mathbf{m}_0]\!]_{\mathcal{A}} = [\![\mathbf{out}_j]\!]_{\mathcal{A}}$.
    - ∗ Otherwise, draw both ms_0 and ms_1 uniformly at random.
  - Repeat this process until $\mathcal{S}_{\mathcal{A}}$ gets through all the first k accesses while taking care that the next access may already be in the shelter $[\![\mathbf{m}]\!]_{\mathcal{A}}$ (and hence already simulated) rather than $[\![\mathbf{D}]\!]_{\mathcal{A}}$.
  - Uniformly draw all the entries not accessed in $[\![\mathbf{D}]\!]_{\mathcal{A}}$ and $[\![\mathbf{m}]\!]_{\mathcal{A}}$.

Simulating $\mathcal{A}$'s view in any *following* epoch:

Simulating the following epochs is *almost* identical to the first. Instead of `Π-init`, we start by invoking `Π-reinit`, which invokes `Π-init` as a subprocedure. In fact, this is the only place where `Π-reinit` communicates. As in `Π-init`, we simulate the seed $\mathbf{sd}^{\mathcal{A}}$ uniformly at random. With $[\![\mathbf{D}]\!]_{\mathcal{A}}$ we need to be once again more careful. In the first epoch, we simulate any accessed $[\![\mathbf{D}_i]\!]_{\mathcal{A}}$ by ensuring that $[\![\mathbf{D}_i]\!]_{\mathcal{A}} \oplus \mathbf{r}_i$ equals to the corresponding output share. Now, we additionally need to take care that any accessed $[\![\mathbf{D}_i]\!]_{\mathcal{A}} \oplus \mathbf{r}_i \oplus [\![\mathbf{D}_{\text{prev},i}]\!]_{\mathcal{A}}$ (we assume $\mathcal{S}_{\mathcal{A}}$ aligned $[\![\mathbf{D}]\!]_{\mathcal{A}}$, $\mathbf{r}$, and $[\![\mathbf{D}_{\text{prev}}]\!]_{\mathcal{A}}$ for the simulation) equals to the corresponding output share. As before, the unaccessed elements of $[\![\mathbf{D}]\!]_{\mathcal{A}}$ can be simulated with uniformly drawn bits. As for `Π-access`, we simulate ms_0 and ms_1 just as in the first epoch.

Thus, $\mathcal{S}_{\mathcal{A}}$ simulates $\mathcal{A}$'s view and $\mathcal{S}_{\mathcal{A}}$'s output is consistent with $\mathcal{A}$'s output.

We next construct $\mathcal{S}_{\mathcal{B}}(\mathbf{d}, [\![\mathbf{x}]\!]_{\mathcal{B}}, [\![\mathbf{out}]\!]_{\mathcal{B}})$ that gets array $\mathbf{d}$. For each access, he also gets value $[\![x]\!]_{\mathcal{B}}$ and output $[\![out]\!]_{\mathcal{B}}$. We first show how $\mathcal{S}_{\mathcal{B}}$ simulates the first epoch (i.e. the first k accesses) and then argue the messages in the following epochs are simulated identically. $\mathcal{S}_{\mathcal{B}}$ now simulates $\mathcal{B}$'s view:

Simulating $\mathcal{B}$'s view in the *first* epoch:

- Consider `Π-init`. $\mathcal{B}$ receives from $\mathcal{C}$ a seed $\mathbf{sd}^{\mathcal{B}}$, which he uses to generate $\mathbf{m}, \mathbf{s_0}$ & $\mathbf{s_1}$, and an array $[\![\mathbf{D}]\!]_{\mathcal{B}}$, which is the main SOCS-ORAM working array, from which $\mathcal{B}$ accesses elements of $\mathbf{d}$. As in $\mathcal{A}$'s case, all are indistinguishable from uniform bits. $\mathbf{sd}^{\mathcal{B}}$ is uniformly sampled by $\mathcal{C}$ and we simulate it by drawing uniform bits.

  $[\![\mathbf{D}]\!]_{\mathcal{B}}$ is a modified $[\![\mathbf{D}]\!]_{\mathcal{B}}$ that $\mathcal{B}$ initially sends to $\mathcal{C}$. $\mathcal{C}$ masks each entry of $[\![\mathbf{D}]\!]_{\mathcal{B}}$ with $\mathbf{r}$ that he generates from a uniformly drawn seed unknown to $\mathcal{B}$. He then extends $[\![\mathbf{D}]\!]_{\mathcal{B}}$ with a share of $\mathbf{m}$ that is also indistinguishable from uniform and is unknown to $\mathcal{B}$. Although each entry of $[\![\mathbf{D}]\!]_{\mathcal{B}}$ was now masked with a mask or set to a value both indistinguishable from uniform, $\mathcal{B}$ could still learn the access pattern as the entries of $[\![\mathbf{D}]\!]_{\mathcal{B}}$ are not permuted. Thus, $\mathcal{A}$ permutes $[\![\mathbf{D}]\!]_{\mathcal{B}}$ according to a random permutation π generated with randomness unknown to $\mathcal{B}$ before sending it to $\mathcal{B}$. $[\![\mathbf{D}]\!]_{\mathcal{B}}$ is now indistinguishable from uniform.

  Recall that $[\![\mathbf{D}]\!]_{\mathcal{B}}$ contains shares of the entries that will be output by Π-SOCS-ORAM. These entries must be consistent with the protocol's output; the remaining entries can be drawn uniformly at random.

- To simulate $[\![\mathbf{D}]\!]_{\mathcal{B}}$, consider `Π-access`. In each invocation of `Π-access`, $\mathcal{B}$ receives *physical* index $\mathbf{pos}_i$ to retrieve from $[\![\mathbf{D}]\!]_{\mathcal{B}}$ and output. These physical indices are determined by $\mathcal{A}$ inputting a *logical* index into a random permutation π, which $\mathcal{B}$ does not know. If a logical index is requested more than once, the entry at that index is moved to an unused location in the shelter

within the same array that is also permuted with π. Thus, each physical index looks random and is unique across all accesses. Hence, $\mathcal{B}$ simulates $\mathbf{pos}_i$ by uniformly drawing indices in $[|[\![\mathbf{D}]\!]_\mathcal{B}|]$ without replacement.

Now that the accessed $\mathbf{pos}_i$ indices are sampled, $\mathcal{S}_\mathcal{B}$ simulates $[\![\mathbf{D}]\!]_\mathcal{B}$ by:

- Setting entries at $\mathbf{pos}_i$ to `Π-`SOCS-ORAM's output shares.
- Drawing the remaining entries uniformly at random.

In each call to `Π-access`, $\mathcal{B}$ also receives bit b from $\mathcal{A}$, which is the opcode *op* XORed with $\mathbf{f}_q$. $\mathbf{f}_q$ is derived from a uniformly drawn seed unknown to $\mathcal{B}$, and hence $\mathcal{S}_\mathcal{B}$ simulates b by drawing a random bit.

Simulating $\mathcal{B}$'s view in any *following* epoch:

Simulating the messages in the following epochs is identical to the first epoch. As before, simulating `Π-access` is trivial. All messages in `Π-access` look random; the challenge for $\mathcal{S}_\mathcal{B}$ is that $[\![\mathbf{D}]\!]_\mathcal{B}$, output by `Π-reinit`, is consistent with `Π-`SOCS-ORAM's output. Recall from our construction of $\mathcal{S}_\mathcal{A}$ that `Π-reinit` communicates only when it invokes `Π-init`. Hence, to simulate `Π-reinit`, we need to simulate only the messages sent to $\mathcal{B}$ by `Π-init`. The key difference from constructing $\mathcal{S}_\mathcal{A}$ is that $\mathcal{B}$'s state from the previous epoch does not modify the messages output by `Π-init` (for $\mathcal{A}$ we XOR $[\![\mathbf{D}_{\text{prev}}]\!]_\mathcal{A}$ from previous epoch into $[\![\mathbf{D}]\!]_\mathcal{A}$ output by `Π-init`). Hence, we simulate the messages just as in `Π-init` in the first epoch. Recall we do that in conjunction with `Π-access` as we need to sample the positions $\mathbf{pos}_i$ that need to match the output shares $[\![\mathbf{out}]\!]_\mathcal{B}$.

Thus, $\mathcal{S}_\mathcal{B}$ simulates $\mathcal{B}$'s view and $\mathcal{S}_\mathcal{B}$'s output is consistent with $\mathcal{B}$'s output.

We now construct $\mathcal{S}_\mathcal{C}(\bot, \bot)$ that receives no input nor output. $\mathcal{S}_\mathcal{C}$ now simulates $\mathcal{C}$'s view:

- Consider `Π-init`. $\mathcal{C}$ receives a uniform seed $\mathbf{sd}^\mathcal{C}$ from $\mathcal{B}$. $\mathcal{B}$ samples this seed uniformly, and hence $\mathcal{S}_\mathcal{C}$ simulates it with uniform bits.
- Consider `Π-reinit`. As in `Π-init`, $\mathcal{C}$ receives a uniformly sampled seed. Hence, $\mathcal{S}_\mathcal{C}$ simulates it with uniform bits.
- Consider `Π-access`. $\mathcal{C}$ is not involved in `Π-access`. Hence, $\mathcal{C}$ needs not simulate any messages.

Thus, $\mathcal{S}_\mathcal{C}$ simulates $\mathcal{C}$'s view and $\mathcal{S}_\mathcal{C}$'s output is consistent with $\mathcal{C}$'s output ($\bot$).

Putting it all together, our simulators exhibited above produce output that is indistinguishable from the corresponding party's real view. Further, the output of $\mathcal{S}_\mathcal{A}$ (resp. $\mathcal{S}_\mathcal{B}$ and $\mathcal{S}_\mathcal{C}$) is equal to the expected output of party $\mathcal{A}$ (resp. $\mathcal{B}$ and $\mathcal{C}$). Hence, the joint distribution of each simulator's output and SOCS-ORAM's output is indistinguishable from that party's real view and `Π-`SOCS-ORAM's output.

Π-SOCS-ORAM is secure against semi-honest corruption of one party. □

7 Experimental Evaluation

We now experimentally evaluate our construction.

Implementation. We implement our Π-SOCS-ORAM (i.e. we implement `Π-init`, `Π-reinit`, and `Π-access`) in 525 lines of C++ and compile our code with the CMake build tool. Our implementation is natural, but we note some of its interesting aspects. For randomness, we use the PRG implementation of EMP [29]. We parameterize our construction over array entry types via function templates and test our construction with native C++ types (e.g. `uint32_t`). We implement a batched version of `Π-access`, and thus can execute multiple accesses in a single round trip of communication. Our implementation runs on a single thread. In our implementation, we send the seed $\mathbf{sd}^{\mathcal{A}}$ from $\mathcal{C}$ to $\mathcal{A}$ as soon as $\mathcal{C}$ samples it. This is because generating a permutation π from $\mathbf{sd}^{\mathcal{A}}$ is the bottleneck of `Π-init`, and hence we need $\mathcal{A}$ to start generating it as soon as possible.

Experimental Setup. We run all experiments on a machine with Ubuntu 22.04.1 LTS, Intel(R) Core(TM) i7-7800X CPU @ 3.50GHz, and 64GB RAM. All parties run on the same laptop, and network settings are configured with the `tc` command (bandwidth is verified with the `iperf` network performance tool and round-trip latency with the `ping` command). Communication measurements represent the sum across all three parties; wall-clock time represents the maximum among the three parties. We sample each data point over 10 runs and present their arithmetic mean.

Experiments. We report on two experiments. The first evaluates our initialization protocol `Π-init` and our reinitialization protocol `Π-reinit` (see Section 7.1) while the second evaluates our access protocol `Π-access` (see Section 7.2). In both experiments, we measure communication and wall-clock time as a function of array size, which ranges from 2^{20} to 2^{30} with fixed $4B$ array entry size (i.e. `uint32_t`) and $4B$ position map entry size. We measure wall-clock time on 2 different simulated network settings:

1. **LAN 1:** A low latency 1Gbps network with 2ms round-trip latency.
2. **LAN 2:** An ultra low latency network also with 1Gbps bandwidth but with 0.25ms round-trip latency.

7.1 Initialization and Reinitialization Protocols

We now demonstrate that our `Π-init` is efficient for both small and large array sizes. We also show that `Π-reinit` has cost similar to `Π-init`. In this experiment, we fix the number of array accesses to 2^{20}. Figure 5 plots the total communication and the wall-clock time in each network setting.

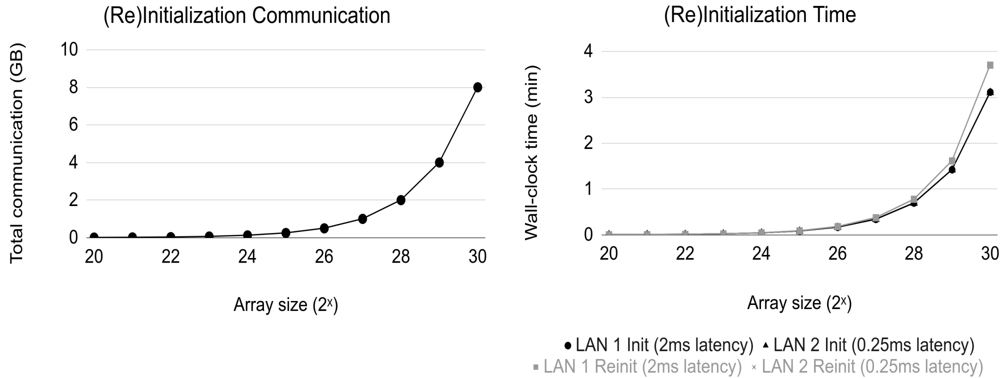


Fig. 5. `Π-init` and `Π-reinit` performance. We fix the number of accesses to 2^{20} and plot the following metrics as functions of the *binary logarithm of the array size*: the overall communication (left) and the wall-clock time to complete both protocols on LAN 1 and LAN 2 (right). Note that the wall-clock time plots of `Π-init` are nearly identical for the two network settings LAN 1 and LAN 2. The `Π-reinit` plots are also nearly identical for the two LAN settings. The communication of `Π-init` and `Π-reinit` protocols is the same; wall-clock time is moderately higher for `Π-reinit` for large array sizes.

Discussion.

- **Communication.** For an array of 2^{30} entries and for 2^{20} accesses, our implementation of `Π-init`/`Π-reinit` communicates 8GB (our plaintext array is 4GB). As expected, the communication costs of `Π-init` and `Π-reinit` are identical. For all runs of `Π-init`/`Π-reinit`, our implementation matches exactly the number of bits incurred by the algorithms.
- **Wall-clock time.** For a large 2^{30}-entry array and for 2^{20} accesses, initialization runs for ≈ 3.1 minutes[3] and re-initialization runs for 3.7 minutes. The cost difference between `Π-init` and `Π-reinit` is primarily due to additional computation on the client, which becomes more pronounced in large arrays. For a small 2^{20}-entry array with the same 2^{20} number of accesses, both initialization and reinitialization take ≈ 0.25 second (≈ 0.004 minute in the plot). The wall-clock time is almost identical for both network settings as (re)initialization consist from algorithmic perspective of only 4 flows of communication; the first 2 and last 2 can be executed in parallel (5 flows in our implementation). Hence, initialization and reinitialization are not sensitive to latency.

[3] Generating permutation takes ≈ 81s. Remaining bottlenecks are ≈ 64s for sending 8GB on 1Gbps network and ≈ 24s for permuting array according to a permutation.

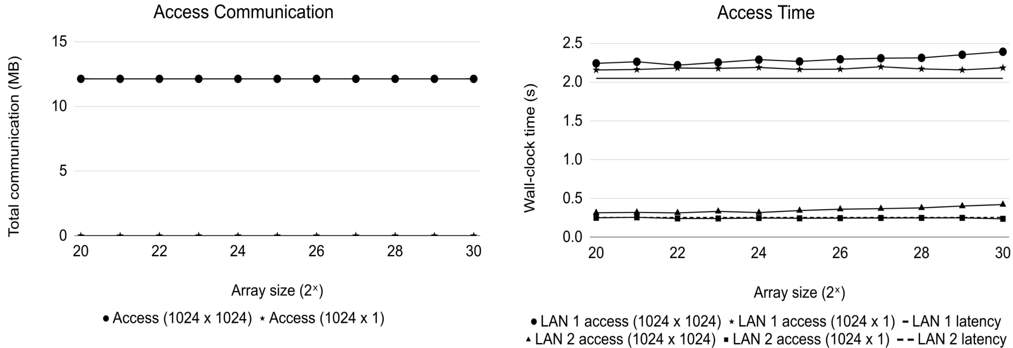


Fig. 6. `Π-access` performance. We consider two parameter regimes for the number of accesses: (1024×1024) and (1024×1). Then we plot the following metrics as functions of the *binary logarithm of the array size*: the overall communication (left) and the wall-clock time to complete the protocol on LAN 1 and LAN 2 (right). For the wall-clock time, we also plot cost because of latency on LAN 1 and LAN 2 to demonstrate our technique incurs almost no overhead beyond latency. Note that LAN 2 latency almost exactly overlaps with LAN 2 access (1024×1).

7.2 Access Protocol

For our second experiment, we show that `Π-access` is fast and its performance is (almost) independent of array size. On localhost, wall-clock time is less than 0.019ms per access for all runs and for all tested array sizes. Communication is 13B[4] per access.

In this experiment, we consider 2 different parameter regimes for the number of accesses. The first (1024×1024) considers 1024 *sequential* accesses with each sequential access containing 1024 *batched* accesses. The second (1024×1) considers 1024 sequential accesses executed in batches of only 1 access. Figure 6 plots the total communication and the wall-clock time in each network setting.

Discussion.

– **Communication.** In `Π-access` communication is independent of array size.[5] In the (1024×1024) access number configuration, we use 12.125MB of communication. This matches exactly the theoretical communication in Figure 3. In the (1024×1) setting, we communicate 13KB (i.e. 13B per access). Note that in this configuration we are losing 7 bits per access on the theoretical communication. This is because we send a single bit as one byte, which we package with other bits in the batched setting.

[4] Note that this applies only to $4B$ array entries and $4B$ position map entries. The communication consists of sending two array entries ($8B$), a single entry in a position map ($4B$), and a single Boolean ($1B$).

[5] This is true as long as the array size stays small enough so that the entries in the position map need not increase (e.g. to 8B i.e. `uint64_t`).

– **Wall-clock time.** First note that in the (1024×1024) configuration and on a 2ms round-trip latency network, `Π-access` takes $\approx$ 2.24s on a 2^{20}-entry array (2.19ms per 1024 parallel accesses) and $\approx$ 2.39s on a 2^{30}-entry array (2.33ms per 1024 parallel accesses). We believe the difference between the two experiments (and over the 2ms latency baseline) is due to low-level costs such as effects of caching, system calls, interprocess communication, precision of `tc` timing, etc. From algorithmic perspective, the performed work is independent of array size.

Acknowledgments: Work of Vladimir Kolesnikov and Stanislav Peceny is supported in part by Visa research award, Cisco research award and NSF awards CNS-2246354 and CCF-2217070. Work of Ni Trieu is supported in part by NSF #2101052, #2200161, and #2115075. Work of Xiao Wang is supported in part by NSF #2016240 and #2236819.

References

1. Kolesnikov V, Peceny S, Trieu N, Wang X. Fast ORAM with Server-Aided Preprocessing and Pragmatic Privacy-Efficiency Trade-Off. In: Dolev S, Gudes E, Paillier P, editors. Cyber Security, Cryptology, and Machine Learning. Cham: Springer Nature Switzerland; 2023. p. 439-57.
2. Goldreich O, Ostrovsky R. Software Protection and Simulation on Oblivious RAMs. J ACM. 1996;43(3):431-73. Available from: `http://doi.acm.org/10.1145/233551.233553`.
3. Ostrovsky R, Shoup V. Private Information Storage (Extended Abstract). In: 29th ACM STOC. ACM Press; 1997. p. 294-303.
4. Gordon SD, Katz J, Kolesnikov V, Krell F, Malkin T, Raykova M, et al. Secure two-party computation in sublinear (amortized) time. In: Yu T, Danezis G, Gligor VD, editors. ACM CCS 2012. ACM Press; 2012. p. 513-24.
5. Doerner J, shelat a. Scaling ORAM for Secure Computation. In: Thuraisingham BM, Evans D, Malkin T, Xu D, editors. ACM CCS 2017. ACM Press; 2017. p. 523-35.
6. Faber S, Jarecki S, Kentros S, Wei B. Three-Party ORAM for Secure Computation. In: Iwata T, Cheon JH, editors. ASIACRYPT 2015, Part I. vol. 9452 of LNCS. Springer, Heidelberg; 2015. p. 360-85.
7. Jarecki S, Wei B. 3PC ORAM with Low Latency, Low Bandwidth, and Fast Batch Retrieval. In: Preneel B, Vercauteren F, editors. ACNS 18. vol. 10892 of LNCS. Springer, Heidelberg; 2018. p. 360-78.
8. Pappas V, Krell F, Vo B, Kolesnikov V, Malkin T, Choi SG, et al. Blind Seer: A Scalable Private DBMS. In: 2014 IEEE Symposium on Security and Privacy. IEEE Computer Society Press; 2014. p. 359-74.
9. Porras P, Shmatikov V. Large-scale collection and sanitization of network security data: risks and challenges. NSPW. 2006. Available from: `https://www.cs.cornell.edu/~shmat/shmat_nspw06.pdf`.
10. Stefanov E, Shi E, Song DX. Towards Practical Oblivious RAM. In: NDSS 2012. The Internet Society; 2012. .

11. Zahur S, Wang XS, Raykova M, Gascón A, Doerner J, Evans D, et al. Revisiting Square-Root ORAM: Efficient Random Access in Multi-party Computation. In: 2016 IEEE Symposium on Security and Privacy. IEEE Computer Society Press; 2016. p. 218-34.
12. Wang XS, Huang Y, Chan THH, shelat a, Shi E. SCORAM: Oblivious RAM for Secure Computation. In: Ahn GJ, Yung M, Li N, editors. ACM CCS 2014. ACM Press; 2014. p. 191-202.
13. Wang X, Chan THH, Shi E. Circuit ORAM: On Tightness of the Goldreich-Ostrovsky Lower Bound. In: Ray I, Li N, Kruegel C, editors. ACM CCS 2015. ACM Press; 2015. p. 850-61.
14. Shi E, Chan THH, Stefanov E, Li M. Oblivious RAM with $O((\log N)^3)$ Worst-Case Cost. In: Lee DH, Wang X, editors. ASIACRYPT 2011. vol. 7073 of LNCS. Springer, Heidelberg; 2011. p. 197-214.
15. Bunn P, Katz J, Kushilevitz E, Ostrovsky R. Efficient 3-Party Distributed ORAM. In: Galdi C, Kolesnikov V, editors. SCN 20. vol. 12238 of LNCS. Springer, Heidelberg; 2020. p. 215-32.
16. Falk BH, Noble D, Ostrovsky R. 3-Party Distributed ORAM from Oblivious Set Membership. SN. 2022.
17. Lu S, Ostrovsky R. How to Garble RAM Programs. In: Johansson T, Nguyen PQ, editors. EUROCRYPT 2013. vol. 7881 of LNCS. Springer, Heidelberg; 2013. p. 719-34.
18. Heath D, Kolesnikov V, Ostrovsky R. EpiGRAM: Practical Garbled RAM. In: Dunkelman O, Dziembowski S, editors. EUROCRYPT 2022, Part I. vol. 13275 of LNCS. Springer, Heidelberg; 2022. p. 3-33.
19. Yang Y, Peceny S, Heath D, Kolesnikov V. Towards Generic MPC Compilers via Variable Instruction Set Architectures (VISAs). In: Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security. CCS '23. New York, NY, USA: Association for Computing Machinery; 2023. p. 2516–2530. Available from: `https://doi.org/10.1145/3576915.3616664`.
20. Chan THH, Chung KM, Maggs BM, Shi E. Foundations of Differentially Oblivious Algorithms. In: Chan TM, editor. 30th SODA. ACM-SIAM; 2019. p. 2448-67.
21. Lu S, Ostrovsky R. Distributed Oblivious RAM for Secure Two-Party Computation. In: Sahai A, editor. TCC 2013. vol. 7785 of LNCS. Springer, Heidelberg; 2013. p. 377-96.
22. Abraham I, Fletcher CW, Nayak K, Pinkas B, Ren L. Asymptotically Tight Bounds for Composing ORAM with PIR. In: Fehr S, editor. PKC 2017, Part I. vol. 10174 of LNCS. Springer, Heidelberg; 2017. p. 91-120.
23. Gordon SD, Katz J, Wang X. Simple and Efficient Two-Server ORAM. In: Peyrin T, Galbraith S, editors. ASIACRYPT 2018, Part III. vol. 11274 of LNCS. Springer, Heidelberg; 2018. p. 141-57.
24. Chan THH, Katz J, Nayak K, Polychroniadou A, Shi E. More is Less: Perfectly Secure Oblivious Algorithms in the Multi-server Setting. In: Peyrin T, Galbraith S, editors. ASIACRYPT 2018, Part III. vol. 11274 of LNCS. Springer, Heidelberg; 2018. p. 158-88.
25. Kushilevitz E, Mour T. Sub-logarithmic Distributed Oblivious RAM with Small Block Size. In: Lin D, Sako K, editors. PKC 2019, Part I. vol. 11442 of LNCS. Springer, Heidelberg; 2019. p. 3-33.
26. Chor B, Kushilevitz E, Goldreich O, Sudan M. Private information retrieval. FOCS. 1995.
27. Gertner Y, Ishai Y, Kushilevitz E, Malkin T. Protecting Data Privacy in Private Information Retrieval Schemes. In: 30th ACM STOC. ACM Press; 1998. p. 151-60.

28. Beaver D. Precomputing Oblivious Transfer. In: Coppersmith D, editor. CRYPTO'95. vol. 963 of LNCS. Springer, Heidelberg; 1995. p. 97-109.
29. Wang X, Malozemoff AJ, Katz J. EMP-toolkit: Efficient MultiParty computation toolkit; 2016. `https://github.com/emp-toolkit`.