Fast ORAM with Server-Aided Preprocessing and Pragmatic Privacy-Efficiency Trade-Off

Vladimir Kolesnikov[1], Stanislav Peceny[1](✉), Ni Trieu[2], and Xiao Wang[3]

[1] Georgia Tech, Atlanta, GA, USA
{kolesnikov,stan.peceny}@gatech.edu
[2] Arizona State University, Tempe, AZ, USA
[3] 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 ≈0.7 s 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 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: Secure Computation · Oblivious RAM

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 and multiparty computation (2PC, MPC). While in plaintext execution such accesses are cheap 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 [GO96], which achieve polylog complexity, while still hiding access patterns.

The first ORAM considered the client-server setting [GO96], where a client wishes to store and access her private array on an untrusted server. Soon after, initiated by [OS97,GKK+12], ORAM was shown applicable to RAM-based MPC: Secure RAM access was achieved for MPC 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 [Ds17] takes $\approx$2 s per access, communicates $\approx$5 MBs, 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 MPC of many functions is much faster in a 3-party honest-majority setting than in the two-party setting, [FJKW15] ask whether SC-ORAM can also be accelerated. [FJKW15] present a solution and report total wall-clock time of 1.62 s 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$1.25 s. A follow-up work [JW18] then asymptotically reduced the bandwidth of [FJKW15], but still reports $\approx$0.7 s CPU time per access on a 2^{30}-element array.

Although 3-party SC-ORAM improves over 2-party SC-ORAMs, a 0.7 s RAM access time will still be considered prohibitive in many (most?) realistic use scenarios. Note, accessing smaller-size memories would be, of course, cheaper: [JW18] reports 0.1 s 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 0.1 s on a 1 Gbps LAN.

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*.

This trust model may naturally occur in real-world scenarios (see Sect. 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. $\mathcal{A}$ requests up to k read or write accesses to $\mathbf{d}$. For each access, $\mathcal{A}$ inputs index $i \in [n]$ and operation *op* (`read` or `write`). $\mathcal{A}$ and $\mathcal{B}$ hold a sharing of a value to write $[\![x]\!]$. $\mathcal{C}$ holds no input and *does not* participate in ORAM access; it is used to help initialize SOCS-ORAM.

All parties are semi-honest and do not collude with one another. We allow $\mathcal{A}$ to learn the access pattern – indeed Alice 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 was also considered in Blind Seer [PKV+14], a scalable privacy-preserving database management system that supports a rich query set for database search and addresses query privacy. [PKV+14] 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 [PS06].

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$1 s 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 [PKV+14]. 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 Alice knows the logical indices of the RAM queries, and the results are additively (XOR) secret-shared between her and the server Bob, 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 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 an efficient third-party aided initialization protocol 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 $4n + 6k$ array entries, k bits, and a single permutation of size $n + k$, sent as a table. $2n$ entries are sent by $\mathcal{B}$, and the rest by $\mathcal{C}$.
Our access protocol communicates only 2 array entries, a single array index, and an additional bit, and requires a single round trip 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 2.13 ms on a 2 ms latency network (as set by the Linux `tc` command; the actual latency, due to system calls overhead is closer to 2.13 ms).
Thus, our wall-clock time is extremely close to latency cost. While our setting is much simpler than that of SC-ORAM, state-of-the-art 3-party SC-ORAM [JW18] reports ≈0.7 s CPU time for arrays of the same size, while all our runs ran in less than 0.019 ms 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. 4 GB size array) and 2^{20} accesses, the cost to initialize our SOCS-ORAM (preprocessing) is 4.8 min and 20 GB communication. In Sect. 6.2, we discuss a natural optimization that would reduce communication to 8 GB.

2 Notation

– Party $\mathcal{A}$ (Alice) inputs access indices i (i.e. client).
– Party $\mathcal{B}$ (Bob) inputs array $\mathbf{d}$ (i.e. server).
– Party $\mathcal{C}$ (Charlie) is a 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. We index these arrays with a ',' (e.g. $\mathbf{s}_{\mathbf{0},i}$).
– We denote negation of a bit b as $\bar{b}$.
– We manipulate XOR secret shares.
  - We use the shorthand $[\![\mathbf{d}]\!]$ to denote a 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 [GO96] 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 an initialization protocol that takes as input an array of entries and initializes an oblivious structure with those entries, as well as an access protocol that implements each *logical* (`read` and `write`) access on the oblivious structure with a sequence of polylog *physical* 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 [SSS12].

Definition 1. *Let* $y := ((op_0, i_0, x_0), (op_1, i_1, x_1), \ldots, (op_{m-1}, i_{m-1}, x_{m-1}))$ *denote a sequence of logical accesses of length* m*, 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* $A(\vec{y})$ *denote the (possibly randomized) sequence of physical accesses to the remote storage given the sequence of logical accesses* $\vec{y}$*. ORAM is said to be secure if for any two sequences of logical accesses* $\vec{y}$ *and* $\vec{z}$ *of the same length, their access patterns* $A(\vec{y})$ *and* $A(\vec{z})$ *are computationally indistinguishable by anyone but the client.*

RAM-Based Secure Computation. [OS97] noted the idea of using ORAM for secure multi-party computation (SC-ORAM). [GKK+12] 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 length 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 [ZWR+16] explicates:

- 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. [WHC+14] 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) and (5) private information retrieval (PIR).

2-Party SC-ORAM. [OS97] 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. [GKK+12] presented a specific SC-ORAM construction that started a long line of research to improve SC-ORAM. [WHC+14] 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. [WCS15] followed up with Circuit ORAM, which further reduced circuit complexity. [ZWR+16] 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 [GKK+12, WHC+14, WCS15, ZWR+16] are recursively structured and as a result require $O(\log n)$ rounds of communication per access; they have expensive initialization algorithms, high memory overhead. E.g., [Ds17] observed they could not handle arrays of sizes larger than $\approx 2^{20}$ on standard hardware. With this in mind, [Ds17] 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, [Ds17] takes ≈ 2 s per access and communicates ≈ 5 MBs in communication on arrays of size 2^{30} with 4-byte elements.

3-Party SC-ORAM. [FJKW15] 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 [SCSL11] in secure computation. For a 2^{36}-element array of 4-byte entries, their access runs in 1.62 s 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. [JW18] followed up on their work and designed custom-made protocols to instead emulate the Circuit ORAM [WCS15] client. While their technique still requires $O(\log n)$ communication rounds per access, they asymptotically decrease the bandwidth of [FJKW15] by the statistical security parameter. Concretely, they report $\approx$0.7 s 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 0.019 ms on localhost on a same-size array.

[BKKO20] 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. [FNO22] 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 [LO13] but was not suitable for practice until [HKO22] introduced EpiGRAM. While EpiGRAM was not implemented, [HKO22] estimate that for an array of 2^{20} entries of 16B, the per-access communication amortized over 2^{20} accesses is $\approx$16 MB. In comparison, our work communicates $\approx$0.2 KB (initialization included) amortized over the same number of accesses.

Differential Obliviousness (DO). DO [CCMS19] 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. [CCMS19] showed that for some programs DO incurs only $O(\log \log n)$ overhead in contrast to ORAM's polylog complexity. We forfeit access pattern privacy against Alice.

Private Information Retrieval (PIR). PIR [CKGS95] 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) [GIKM98]. 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 [Ds17, GKW18, JW18, KM19, BKKO20] 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 are in Sect. 6. For accompanying proofs of correctness and security see the full version.

Recall from Sect. 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 initialization and is not active during array access. Initialization provisions for k *dynamic* 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, 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 Fig. 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 $\mathbf{pos}_i := \pi_i \overset{\Delta}{=} \pi(i)$ 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 q^{th} 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 $\mathbf{D}_{\mathbf{pos}_{n+q}} = \mathbf{D}_{\mathbf{pos}_i}$ *solely* by updating her share $[\![\mathbf{D}_{\mathbf{pos}_{n+q}}]\!]_{\mathcal{A}}$. $\mathcal{A}$ can do this because at initialization $\mathcal{C}$ will perform an additional step: He generates a k-element random mask vector $\mathbf{m}$ and XORs it into the shelter positions of $[\![\mathbf{D}_{\mathbf{pos}_i}]\!]_{\mathcal{A}}$ (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 $[\![\mathbf{D}_{\mathbf{pos}_i}]\!]_{\mathcal{A}}$ to obtain $[\![\mathbf{D}_{\mathbf{pos}_{n+q}}]\!]_{\mathcal{A}}$. 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 $[\![\mathbf{D}_{\mathbf{pos}_{n+q}}]\!]$. This extension is simple to achieve with an 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.

6 Our SOCS-ORAM

We now formally present our scheme. In Sect. 6.1, we define SOCS-ORAM's cleartext semantics. In Sect. 6.2, we specify Π-SOCS-ORAM, our protocol implementing SOCS-ORAM. We defer proofs of correctness and security to the full version of our paper.

6.1 Cleartext Semantics: SOCS-ORAM

Definition 2 *(Cleartext Semantics SOCS-ORAM).* *SOCS-ORAM$(\boldsymbol{d})_{n,k,l}$ is a 3-party stateful functionality executed between parties $\mathcal{A}$, $\mathcal{B}$, and $\mathcal{C}$ that receives a sequence of up to $k+1$ instructions. The first instruction is* `init`$(\boldsymbol{d})$*, where $\boldsymbol{d}$ is an array of n l-bit values and is input by $\mathcal{B}$.* `init` *associates the values in $\boldsymbol{d}$ with their corresponding indices in $[n]$ and sets $\boldsymbol{D} := \boldsymbol{d}$. The remaining up to k instructions are* $\texttt{access}_{\boldsymbol{D}}(op, i, [\![x]\!])$ *instructions.* `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 $\boldsymbol{D}$. See Fig. 1 for the* `init` *and* `access` *functionalities.*

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

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

- 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 $\mathbf{D} := \mathbf{d}$ $\quad // \forall i \in [n], \quad \mathbf{D}_i := \mathbf{d}_i$

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

$\texttt{read}(i)$:

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

$\texttt{write}(i, [\![x]\!])$:

- 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 $[\![out]\!]$

$\texttt{access}(op, i, [\![x]\!])$:

- 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` and `access` functionalities for our SOCS-ORAM.

6.2 Protocol: Π-SOCS-ORAM

In this section, we formalize our protocol Π-SOCS-ORAM, which securely implements the semantics of SOCS-ORAM (Definition 2):

Construction 1 *(Protocol Π-SOCS-ORAM).* Π-*SOCS-ORAM*$(\mathbf{d})_{n,k,l}$ *is defined by first invoking* Π-`init` *in Fig. 2 and then up to k invocations to* Π-`access` *in Fig. 3.*

We include the proof of the following theorem in the full version of this paper:

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` and Π-`access` (see Construction 1), we separate Π-SOCS-ORAM's description into Π-`init` (Fig. 2) and Π-`access`(Fig. 3), respectively.

Π-`init`. Π-`init` sets up data structures necessary to `access` $\mathbf{d}$ (see `init` in Fig. 1). It is a 3-party protocol, where $\mathcal{A}$ and $\mathcal{B}$ 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 $[\![\mathbf{D}]\!]_{\mathcal{B}}$. $\mathcal{C}$ then helps to construct the data structures used in Π-`access` for $\mathcal{A}$ and $\mathcal{B}$.

$\mathcal{C}$ first generates the data structures for $\mathcal{B}$. He uniformly samples array $\mathbf{r}$ of same size as $\mathbf{D}$. $\mathcal{C}$ masks his share of $\mathbf{D}$ with $\mathbf{r}$, i.e. computes $[\![\mathbf{D}]\!]_{\mathcal{B}} := [\![\mathbf{D}]\!]_{\mathcal{B}} \oplus \mathbf{r}$. Simultaneously, $\mathcal{C}$ uniformly samples array $\mathbf{m}$, which will hold shelter values, where array entries will be written once they are accessed. $\mathbf{m}$ has k l-bit entries, where k determines the maximum number of array accesses. $\mathcal{C}$ secret-shares $\mathbf{m}$, and appends $[\![\mathbf{D}]\!]_{\mathcal{B}} := [\![\mathbf{D}]\!]_{\mathcal{B}} || [\![\mathbf{m}]\!]_{\mathcal{B}}$. Now $\mathcal{C}$ draws a random permutation $\pi : [n+k] \rightarrow [n+k]$ and permutes $[\![\mathbf{D}]\!]_{\mathcal{B}}$ according to π. $\mathcal{C}$ also uniformly samples two arrays $\mathbf{s}_0, \mathbf{s}_1$ of k l-bit entries, which will help $\mathcal{A}$ obliviously retrieve the message corresponding to either `read` or `write` operation during `access`. $\mathcal{C}$ then sends the masked and permuted $[\![\mathbf{D}]\!]_{\mathcal{B}}$ along with the masks $\mathbf{m}$, $\mathbf{s}_0$, and $\mathbf{s}_1$ to $\mathcal{B}$. $\mathcal{B}$ stores them for Π-`access` and additionally sets a counter $q := 0$ that counts the number of accesses.

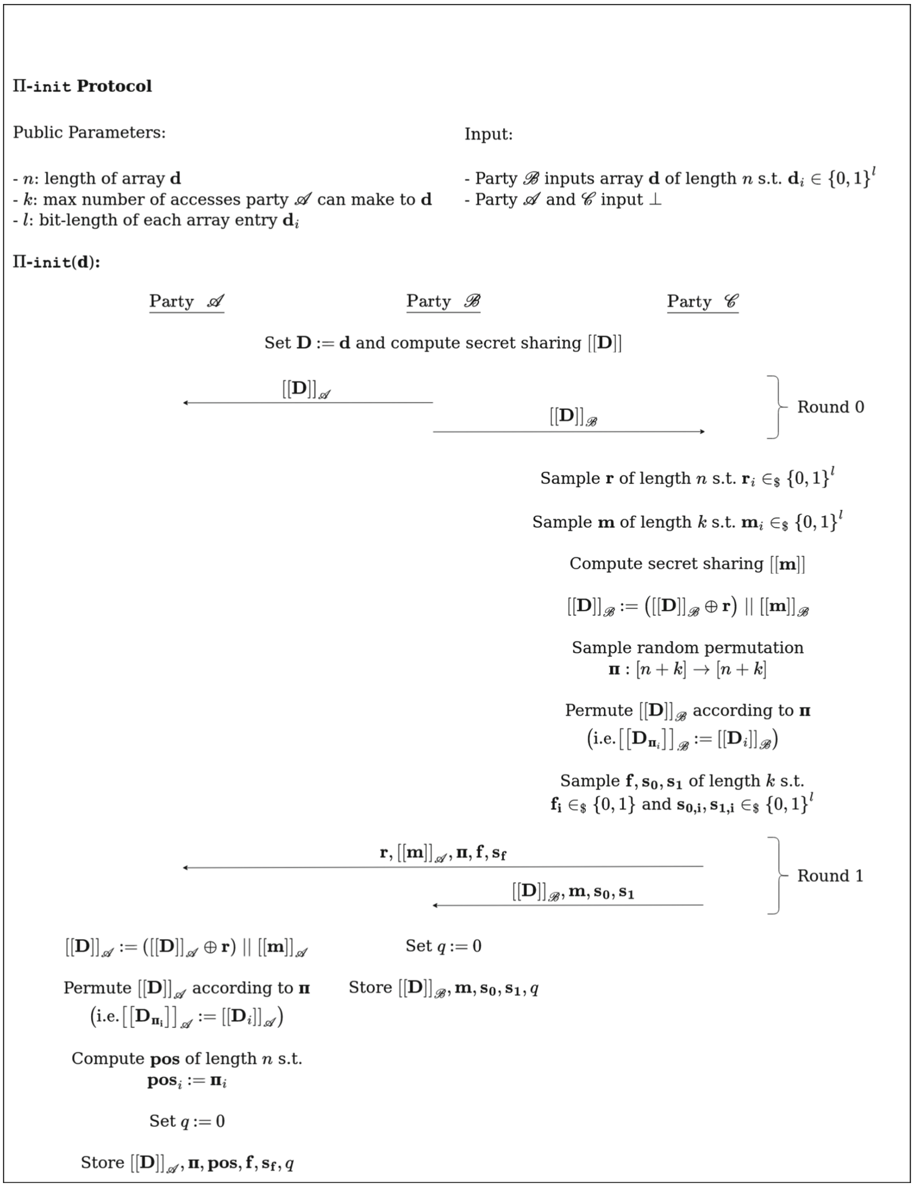
$\mathcal{C}$ next generates and sends randomness to $\mathcal{A}$ that will enable it to construct its data structures. $\mathcal{C}$ already generated $[\![\mathbf{r}]\!]_{\mathcal{A}}$, $[\![\mathbf{m}]\!]_{\mathcal{A}}$, and π. He also uniformly samples k-bit $\mathbf{f}$ and constructs $\mathbf{s}_f$ such that for all $i \in [k]$ it contains $\mathbf{s}_{0,i}$ or $\mathbf{s}_{1,i}$ depending on $\mathbf{f}_i$. $\mathcal{C}$ then sends $[\![\mathbf{r}]\!]_{\mathcal{A}}, [\![\mathbf{m}]\!]_{\mathcal{A}}, \pi, \mathbf{f}, \mathbf{s}_f$ to $\mathcal{A}$.

$\mathcal{A}$ now constructs its data structures. First, she 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 $i \in [n]$. $\mathcal{A}$ stores $[\![\mathbf{D}]\!]_{\mathcal{A}}, [\![\mathbf{m}]\!]_{\mathcal{A}}, \pi, \mathbf{pos}, \mathbf{f}, \mathbf{s}_f$ along with a counter $q := 0$. As for $\mathcal{B}$, q tracks the number of accesses.

Optimizing Π-`init` *by Sending Randomness via Seeds.* For array $\mathbf{d}$ of n l-bit entries and k accesses, Π-`init` communicates $4n + 6k$ l-bit array entries, k bits, and a permutation (transferred as an array of length $n+k$). Communication can be reduced to $2n + 2k$ l-bit array entries and 7κ bits by sending randomness via short κ-bit pseudo-random seeds rather than large arrays, and locally expanding them with a pseudo-random generator[1]. In Π-`init`, this technique can be used when sending secret-shared arrays (i.e. send one of the secret shares as a seed), random arrays, and a permutation.

One must take care when using this optimization that the resulting protocol remains simulatable. A subtle technical issue here is that for modularity, we present and prove secure standalone Π-SOCS-ORAM, whose output 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. The solution is either to use programmable primitives (such as programmable random oracle), or to consider the complete MPC problem, where the wire shares are not part of the output.

[1] We did not implement this optimization as our focus was on efficient Π-`access`.



Π-`init` Protocol

Public Parameters:

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

Input:

- Party $\mathscr{B}$ inputs array $\mathbf{d}$ of length n s.t. $\mathbf{d}_i \in \{0,1\}^l$
- Party $\mathscr{A}$ and $\mathscr{C}$ input $\perp$

Π-`init`($\mathbf{d}$):

Party $\mathscr{A}$ | Party $\mathscr{B}$ | Party $\mathscr{C}$

Set $\mathbf{D} := \mathbf{d}$ and compute secret sharing $[[\mathbf{D}]]$

$\xleftarrow{[[\mathbf{D}]]_{\mathscr{A}}}$

$\xrightarrow{[[\mathbf{D}]]_{\mathscr{B}}}$

Round 0

Sample $\mathbf{r}$ of length n s.t. $\mathbf{r}_i \in_\$ \{0,1\}^l$

Sample $\mathbf{m}$ of length k s.t. $\mathbf{m}_i \in_\$ \{0,1\}^l$

Compute secret sharing $[[\mathbf{m}]]$

$[[\mathbf{D}]]_{\mathscr{B}} := \left([[\mathbf{D}]]_{\mathscr{B}} \oplus \mathbf{r}\right) \mid\mid [[\mathbf{m}]]_{\mathscr{B}}$

Sample random permutation
$\boldsymbol{\pi} : [n+k] \to [n+k]$

Permute $[[\mathbf{D}]]_{\mathscr{B}}$ according to $\boldsymbol{\pi}$
$\left(\text{i.e.} \left[[\mathbf{D}_{\boldsymbol{\pi}_i}]\right]_{\mathscr{B}} := [[\mathbf{D}_i]]_{\mathscr{B}}\right)$

Sample $\mathbf{f}, \mathbf{s_0}, \mathbf{s_1}$ of length k s.t.
$\mathbf{f_i} \in_\$ \{0,1\}$ and $\mathbf{s_{0,i}}, \mathbf{s_{1,i}} \in_\$ \{0,1\}^l$

$\xleftarrow{\mathbf{r}, [[\mathbf{m}]]_{\mathscr{A}}, \boldsymbol{\pi}, \mathbf{f}, \mathbf{s_f}}$

$\xleftarrow{[[\mathbf{D}]]_{\mathscr{B}}, \mathbf{m}, \mathbf{s_0}, \mathbf{s_1}}$

Round 1

Party $\mathscr{A}$:

$[[\mathbf{D}]]_{\mathscr{A}} := ([[\mathbf{D}]]_{\mathscr{A}} \oplus \mathbf{r}) \mid\mid [[\mathbf{m}]]_{\mathscr{A}}$

Permute $[[\mathbf{D}]]_{\mathscr{A}}$ according to $\boldsymbol{\pi}$
$\left(\text{i.e.} \left[[\mathbf{D}_{\boldsymbol{\pi}_i}]\right]_{\mathscr{A}} := [[\mathbf{D}_i]]_{\mathscr{A}}\right)$

Compute $\mathbf{pos}$ of length n s.t.
$\mathbf{pos}_i := \boldsymbol{\pi}_i$

Set $q := 0$

Store $[[\mathbf{D}]]_{\mathscr{A}}, \boldsymbol{\pi}, \mathbf{pos}, \mathbf{f}, \mathbf{s_f}, q$

Party $\mathscr{B}$:

Set $q := 0$

Store $[[\mathbf{D}]]_{\mathscr{B}}, \mathbf{m}, \mathbf{s_0}, \mathbf{s_1}, q$

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

Π-`access`. Π-`access` accesses $\mathbf{D}$ (see `access` in Fig. 1). It is a 2-party protocol, run between $\mathcal{A}$ and $\mathcal{B}$, where $\mathcal{A}$ requests `read` or `write` to $\mathbf{D}_i$.

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

$\mathcal{A}$ 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 $\mathcal{B}$.

$\mathcal{B}$ now constructs two messages: the first is for op = `read` and the second for op = `write`. $\mathcal{B}$ retrieves $[\![\mathbf{D}_{\mathbf{pos}_i}]\!]_{\mathcal{B}}$ for his read share and $[\![x]\!]_{\mathcal{B}}$ for his write share. 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. Then he sends both 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 += 1 and output $[\![out]\!]$.

7 Experimental Evaluation

We now experimentally evaluate our construction.

Implementation. We implemented our approach (i.e. `Π-init` and `Π-access`) in 437 lines of C++ and compiled 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 [WMK16]. We parameterize our construction over array entry types via function templates and test our construction with native C++ types (e.g. `uint32_t`). We implemented a batched version of `Π-access`, and thus can execute multiple accesses in a single round of communication. Our implementation runs on a single thread.

Experimental Setup. All experiments were run on a machine running Ubuntu 22.04.1 LTS with Intel(R) Core(TM) i7-7800X CPU @ 3.50 GHz and 64 GB RAM. All parties were run on the same laptop, and network settings were configured with the `tc` command (bandwidth was 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 sampled each data point over 10 runs and present their arithmetic mean.

Experiments. We performed and report on two experiments. The first evaluates our initialization protocol `Π-init` (see Sect. 7.1) while the second evaluates our access protocol `Π-access`(see Sect. 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:

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

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

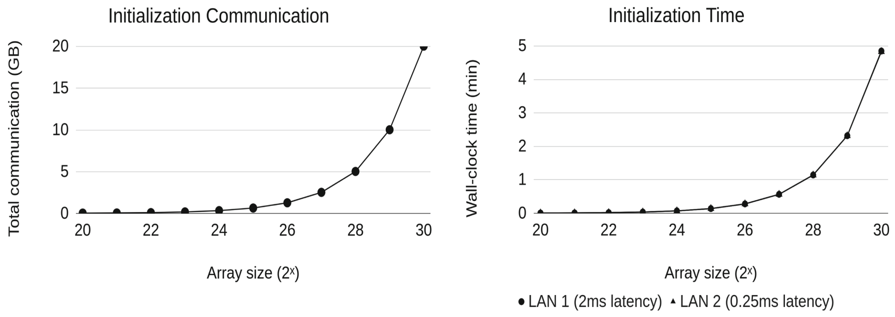


Fig. 4. Π-`init` performance. We fix the number of accesses to 2^{20} and plot the following metrics as functions of the *binary logarithm of the array size*: the overall communication (left) and the wall-clock time to complete the protocol on LAN 1 and LAN 2 (right). Note that the plots for LAN 1 and LAN 2 overlap.

7.1 Initialization Protocol

We first demonstrate that our Π-`init` is efficient for both small and large array sizes. In this experiment, we fix the number of array accesses to 2^{20}. Figure 4 plots the total communication and the wall-clock time in each network setting.

Discussion

- **Communication.** For an array of 2^{30} entries and for 2^{20} accesses, our implementation of Π-`init` communicates 20 GB (our plaintext array is 4 GB). For all runs of the initialization algorithm, our implementation matches exactly the number of bits incurred by our algorithm.
- **Wall-clock time.** For a large 2^{30}-entry array and for 2^{20} accesses, initialization runs for $\approx$4.8 min[2]. For a small 2^{20}-entry array with the same number of accesses, initialization takes $\approx$0.5 s. The wall-clock time is almost identical for both network settings as initialization consists of only 4 flows of communication (the first 2 and last 2 can be executed in parallel). Hence, initialization is not sensitive to latency.

7.2 Access Protocol

For our second experiment, we show that Π-`access` is fast and its performance is (almost) independent of array size. We show that wall-clock time is less than 0.019 ms per access on localhost for all runs and for all tested array sizes. Communication is 13B[3] per access.

[2] Sending 20 GB on 1 Gbps network takes $\approx$2.7 min. Remaining bottlenecks are generating permutation $\approx$71 s and permuting array according to a permutation $\approx$24 s.

[3] 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 ($1B$).

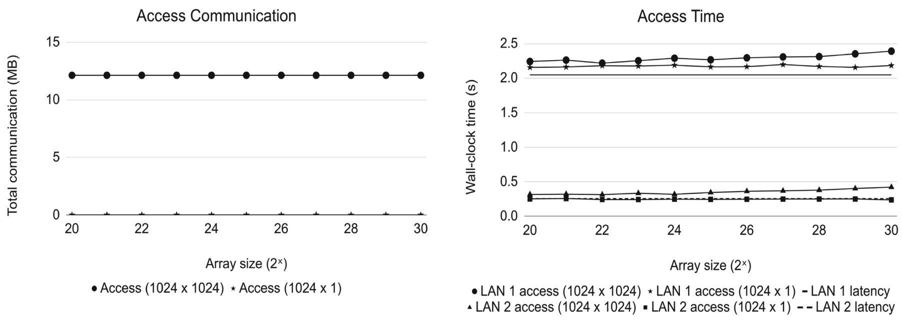


Fig. 5. `Π-access` performance. We consider two parameter regimes for the number of accesses: (1024 × 1024) and 1024 × 1. Then we plot the following metrics as functions of the *binary logarithm of the array size*: the overall communication (left) and the wall-clock time to complete the protocol on LAN 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 over latency. Note that LAN 2 latency almost exactly overlaps with LAN 2 access (1024 × 1).

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 5 plots the total communication and the wall-clock time in each network setting.

Discussion.

- **Communication.** In `Π-access` communication is independent of array size[4]. In the (1024 × 1024) access number configuration, we use 12.125MB of communication. This matches exactly the theoretical communication in Fig. 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 can be packaged with other bits in the batched setting.
- **Wall-clock time.** First note that in the (1024 × 1024) configuration and on a 2 ms round-trip latency network, `Π-access` takes ≈2.24 s on a 2^{20}-entry array (2.19 ms per 1024 parallel accesses) and ≈2.39 s on a 2^{30}-entry array (2.33 ms per 1024 parallel accesses). We believe the difference between the two experiments (and over the 2 ms 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.

[4] 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`).

Acknowledgments. Work of Vlad Kolesnikov is supported in part by 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

[BKKO20] Bunn, P., Katz, J., Kushilevitz, E., Ostrovsky, R.: Efficient 3-party distributed ORAM. In: Galdi, C., Kolesnikov, V. (eds.) SCN 2020. LNCS, vol. 12238, pp. 215–232. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-57990-6_11

[CCMS19] Chan, T.-H.H., Chung, K.-M., Maggs, B.M., Shi, E.: Foundations of differentially oblivious algorithms. In: Chan, T.M. (ed.) 30th SODA, pp. 2448–2467. ACM-SIAM (2019)

[CKGS95] Chor, B., Kushilevitz, E., Goldreich, O., Sudan, M.: Private information retrieval. In: FOCS (1995)

[Ds17] Doerner, J., Shelat, A.: Scaling ORAM for secure computation. In: Thuraisingham, B.M., Evans, D., Malkin, T., Xu, D. (eds.) ACM CCS 2017, pp. 523–535. ACM Press (2017)

[FJKW15] Faber, S., Jarecki, S., Kentros, S., Wei, B.: Three-party ORAM for secure computation. In: Iwata, T., Cheon, J.H. (eds.) ASIACRYPT 2015. LNCS, vol. 9452, pp. 360–385. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-48797-6_16

[FNO22] Falk, B.H., Noble, D., Ostrovsky, R.: 3-party distributed ORAM from oblivious set membership. In: Galdi, C., Jarecki, S. (eds.) SCN 2022. LNCS, vol. 13409, pp. 437–461. Springer, Cham (2022). https://doi.org/10.1007/978-3-031-14791-3_19

[GIKM98] Gertner, Y., Ishai, Y., Kushilevitz, E., Malkin, T.: Protecting data privacy in private information retrieval schemes. In: 30th ACM STOC, pp. 151–160. ACM Press (1998)

[GKK+12] Gordon, S.D., et al.: Secure two-party computation in sublinear (amortized) time. In: Yu, T., Danezis, G., Gligor, V.D. (eds.) ACM CCS 2012, pp. 513–524. ACM Press (2012)

[GKW18] Gordon, S.D., Katz, J., Wang, X.: Simple and efficient two-server ORAM. In: Peyrin, T., Galbraith, S. (eds.) ASIACRYPT 2018, Part III. LNCS, vol. 11274, pp. 141–157. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-03332-3_6

[GO96] Goldreich, O., Ostrovsky, R.: Software protection and simulation on oblivious RAMs. J. ACM **43**(3), 431–473 (1996)

[HKO22] Heath, D., Kolesnikov, V., Ostrovsky, R.: EpiGRAM: practical garbled RAM. In: Dunkelman, O., Dziembowski, S. (eds.) EUROCRYPT 2022, Part I. LNCS, vol. 13275, pp. 3–33. Springer, Cham (2022). https://doi.org/10.1007/978-3-031-06944-4_1

[JW18] Jarecki, S., Wei, B.: 3PC ORAM with low latency, low bandwidth, and fast batch retrieval. In: Preneel, B., Vercauteren, F. (eds.) ACNS 2018. LNCS, vol. 10892, pp. 360–378. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-93387-0_19

[KM19] Kushilevitz, E., Mour, T.: Sub-logarithmic distributed oblivious RAM with small block size. In: Lin, D., Sako, K. (eds.) PKC 2019, Part I. LNCS, vol. 11442, pp. 3–33. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-17253-4_1

[LO13] Lu, S., Ostrovsky, R.: How to garble RAM programs? In: Johansson, T., Nguyen, P.Q. (eds.) EUROCRYPT 2013. LNCS, vol. 7881, pp. 719–734. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-38348-9_42

[OS97] Ostrovsky, R., Shoup, V.: Private information storage (extended abstract). In: 29th ACM STOC, pp. 294–303. ACM Press (1997)

[PKV+14] Pappas, V., et al.: Blind seer: a scalable private DBMS. In: 2014 IEEE Symposium on Security and Privacy, pp. 359–374. IEEE Computer Society Press (2014)

[PS06] Porras, P., Shmatikov, V.: Large-scale collection and sanitization of network security data: risks and challenges. NSPW (2006)

[SCSL11] Shi, E., Chan, T.-H.H., Stefanov, E., Li, M.: Oblivious RAM with $O((\log N)^3)$ worst-case cost. In: Lee, D.H., Wang, X. (eds.) ASIACRYPT 2011. LNCS, vol. 7073, pp. 197–214. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-25385-0_11

[SSS12] Stefanov, E., Shi, E., Song, D.X.: Towards practical oblivious RAM. In: NDSS 2012. The Internet Society (2012)

[WCS15] Wang, X., Chan, T.-H.H., Shi, E.: Circuit ORAM: on tightness of the Goldreich-Ostrovsky lower bound. In: Ray, I., Li, N., Kruegel, C. (eds.) ACM CCS 2015, pp. 850–861. ACM Press (2015)

[WHC+14] Wang, X.S., Huang, Y., Chan, T.-H.H., Shelat, A., Shi, E.: SCORAM: oblivious RAM for secure computation. In: Ahn, G.J., Yung, M., Li, N. (eds.) ACM CCS 2014, pp. 191–202. ACM Press (2014)

[WMK16] Wang, X., Malozemoff, A.J., Katz, J.: EMP-toolkit: efficient MultiParty computation toolkit (2016). https://github.com/emp-toolkit

[ZWR+16] Zahur, S., et al.: Revisiting square-root ORAM: efficient random access in multi-party computation. In: 2016 IEEE Symposium on Security and Privacy, pp. 218–234. IEEE Computer Society Press (2016)