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

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 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 multi-party 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 ≈ 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 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 $1.62s$ 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 [JW18] then asymptotically reduced the bandwidth of [FJKW15], but still reports $\approx$ 0.7s CPU time per access on a 2^{30}-element array.

Although 3-party SC-ORAM improves over 2-party SC-ORAMs, a $0.7s$ 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 0.1s 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.

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 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. $\mathcal{A}$ 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 $\mathcal{A}$ 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 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 $\approx 1s$ 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 [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 $\mathcal{A}$ 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 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 2.13ms 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 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 ≈ 0.7s 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 4.8 minutes and 20GB communication. In Section 6.2, we discuss a natural optimization that would reduce communication to 8GB.

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 (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 [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* `read`(i) *or* `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), (5) multi-server ORAMs in the client-server setting, and (6) 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 and 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 seconds per access and communicates ≈ 5MBs 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 $1.62s$ 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. [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 ≈ 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.

[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 [LO13b] 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 $\approx$ 16MB. In comparison, our work communicates $\approx 0.2KB$ (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 $\mathcal{A}$.

Multi-Server Client-Server ORAM. [LO13a] 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 [AFN+17] reduced the asymptotic communication bandwidth, but did not improve round complexity. [GKW18] 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 protocol.

[CKN+18] 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. [KM19] 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 [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, 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 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. 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 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 $\mathbf{D}_{\pi_{n+q}} = \mathbf{D}_{\mathbf{pos}_i}$ *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 $[\![\mathbf{D}_{\mathbf{pos}_{n+q}}]\!]$. 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 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 prove it correct and secure in Section 6.3.

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` *sets* $\boldsymbol{D} := \boldsymbol{d}$ *to initialize the working array* $\boldsymbol{D}$ *with the input array* $\boldsymbol{d}$*. 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 Figure 1 for the* `init` *and* `access` *functionalities.*

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*$(\boldsymbol{d})_{n,k,l}$ *is defined by first invoking* Π`-init` *in Figure 2 and then up to* k *invocations to* Π`-access` *in Figure 3.*

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

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

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

`Π-init`. `Π-init` sets up working data structures used to `access` $\mathbf{d}$ (see `init` in Figure 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 working 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 $q := 0$, which 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.

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 $\mathbf{D}_i$.

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 $\mathcal{B}$ cannot learn 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 latter for $op =$ write. $\mathcal{B}$ first retrieves $[\![\mathbf{D}_{\mathbf{pos}_i}]\!]_{\mathcal{B}}$ for his read share. From the Π-access input, he already holds $[\![x]\!]_{\mathcal{B}}$ 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 $\mathcal{A}$.

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

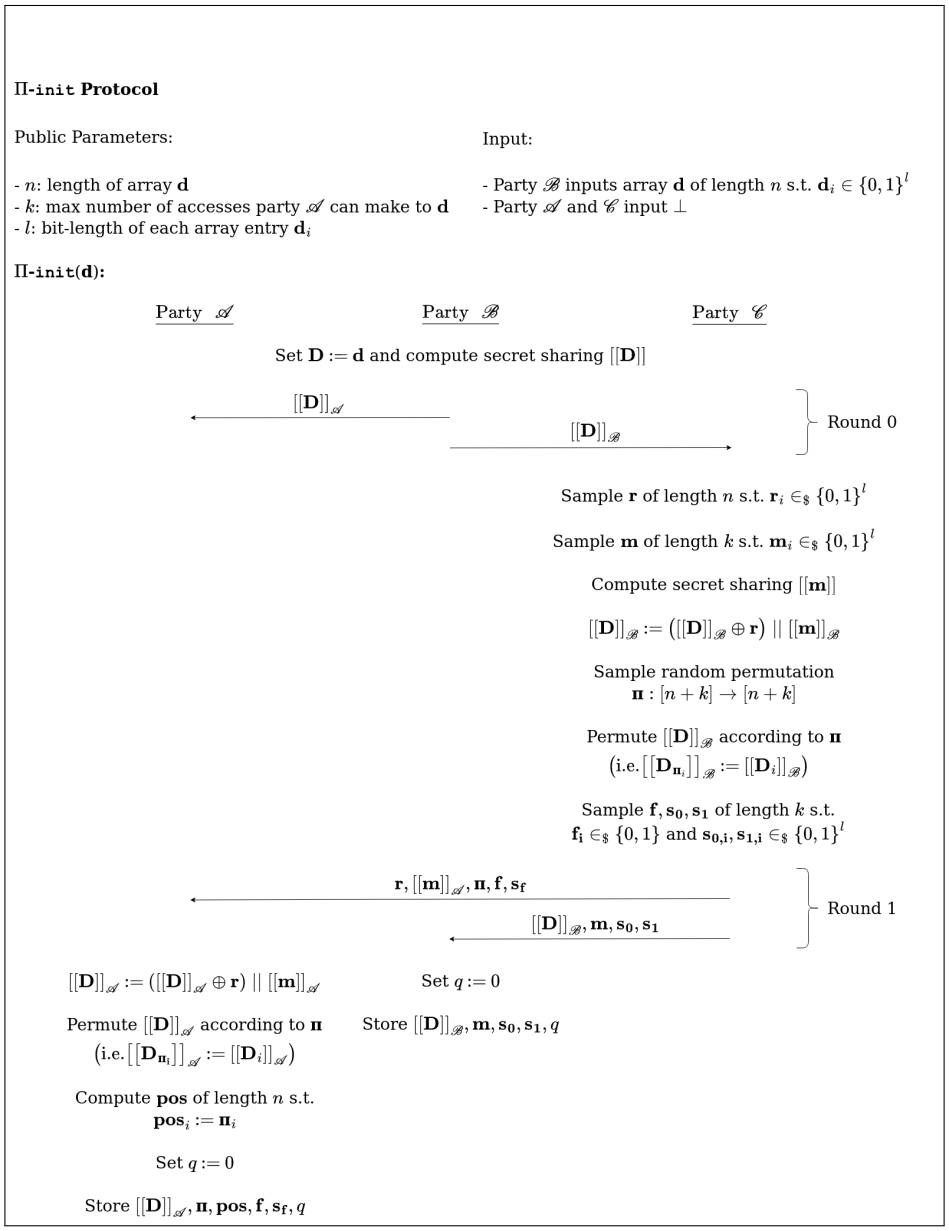

Fig. 2. Π-init is a subroutine of Π-SOCS-ORAM.

$\mathcal{A}$ 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 $[\![out]\!]$.

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 $\mathcal{D}$ be a space of all arrays with n l-bit entries. For all arrays $\boldsymbol{d} \in \mathcal{D}$ and for all sequences of up to $k+1$ instructions starting with* `init`*/* `Π-init` *(the remaining instructions* `access`*/* `Π-access` *not necessarily known a priori):*

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

Proof. We prove Π-SOCS-ORAM correct by showing that `Π-init` computes a valid XOR sharing of the input array $\mathbf{d}$ that `Π-access` can use to read a correct sharing if $op =$ `read` and to write a correct sharing if $op =$ `write`.

`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 entry of $\mathbf{D}$. Next, $[\![\mathbf{D}]\!]$ is permuted according to the *same* random permutation π. Hence, entries in both shares are shifted to a same new position. Therefore, $\mathbf{D}_i$ before permutation equals $\mathbf{D}_{\pi_i}$ after the entries are permuted.

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

So far, this is proved only for the *first time any index is retrieved*. After each access, the retrieved entry (or a new entry if $op =$ `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 entries. What we need to show is that we maintain correct $[\![\mathbf{D}_i]\!]$ if $op =$ `read` and write $[\![x]\!]$ if $op =$ `write`.

Let q represent the access number and $\mathbf{pos}$ the position map that tracks the location of each array entry (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 $[\![x]\!]$. I.e., he computes $\mathbf{m}_q \oplus [\![\mathbf{D}_{\mathbf{pos}_i}]\!]_{\mathcal{B}}$

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 $\mathbf{pos}(i) := \pi_{n+q}$ // 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.

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.

Note that in the real execution $\mathcal{B}$ does not have *op*, and hence does not know which message to send. We now show that our technique sends the correct message to $\mathcal{A}$. 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 $\mathbf{f}_q$. 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 remove.

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 $[\![out]\!]_\mathcal{A}$. $\mathcal{S}_\mathcal{A}$ now simulates $\mathcal{A}$'s view:

- Consider Π-`init`. $\mathcal{A}$ receives a uniform share $[\![\mathbf{D}]\!]_\mathcal{A}$ from $\mathcal{B}$. $\mathcal{A}$ also receives a 5-part message from $\mathcal{C}$: $\mathbf{r}, [\![\mathbf{m}]\!]_\mathcal{A}, \pi, \mathbf{f}, \mathbf{s}_\mathbf{f}$. Note that all are indistinguishable from random bits. $\mathbf{r}$ and $\mathbf{f}$ are uniformly sampled arrays; depending on $\mathbf{f}$, $\mathbf{s}_\mathbf{f}$ has entries from uniformly drawn $\mathbf{s_0}$ or $\mathbf{s_1}$; $[\![\mathbf{m}]\!]_\mathcal{A}$ is a uniform share of $\mathbf{m}$ and π is a random permutation. $\mathbf{r}, \pi, \mathbf{f}, \mathbf{s}_\mathbf{f}$ can be simulated by uniformly drawing bits; $[\![\mathbf{D}]\!]_\mathcal{A}$ and $[\![\mathbf{m}]\!]_\mathcal{A}$ are more complex to simulate as they must result in $[\![\mathbf{D}]\!]_\mathcal{A}$ that is consistent with Π-SOCS-ORAM's output. To simulate them, we first discuss Π-`access`.

– Consider Π-`access`. Note that $\mathcal{S}_\mathcal{A}$ gets a list of indices as input. Thus, $\mathcal{S}_\mathcal{A}$ knows which entries of $[\![\mathbf{D}]\!]_\mathcal{A}$ (combined with $[\![\mathbf{m}]\!]_\mathcal{A}$) 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 drawn uniformly and is unknown to $\mathcal{A}$. Additionally, ms_0 is masked by $\mathbf{s}_{\mathbf{b},q}$, ms_1 by $\mathbf{s}_{\overline{\mathbf{b}},q}$, which were both generated uniformly at random and $\mathcal{A}$ knows *exactly one* of them. Note that both parts look uniform to $\mathcal{A}$ because even after removing one of the second pair of masks, both parts remain masked with two different uniformly sampled values. Thus, $\mathcal{S}_\mathcal{A}$ draws them uniformly.

  Hence, we observe that all messages received during the full protocol are indistinguishable from random. We now go through the access indices one by one and simulate $[\![\mathbf{D}]\!]_\mathcal{A}$ and $[\![\mathbf{m}]\!]_\mathcal{A}$:
  - 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 $\mathbf{r}$ 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 computation:
    * If yes, check if the *nearest* `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`, set $[\![\mathbf{m}]\!]_\mathcal{A}$ such that it results in the right output share 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`, ensure $[\![\mathbf{m}]\!]_\mathcal{A} \oplus [\![x]\!]_\mathcal{A} \oplus ms_1 \oplus \mathbf{s}_{\mathbf{f},q}$ add to the right output share.
    * Otherwise, set the appropriate value in $[\![\mathbf{m}]\!]_\mathcal{A}$ to uniformly drawn values.
  - Repeat this process until $\mathcal{S}_\mathcal{A}$ gets through all accesses while taking care that the next access may be in $[\![\mathbf{m}]\!]_\mathcal{A}$ (and hence already simulated) rather than $[\![\mathbf{D}]\!]_\mathcal{A}$.
  - Uniformly draw all the entries not accessed in $[\![\mathbf{D}]\!]_\mathcal{A}$ and $[\![\mathbf{m}]\!]_\mathcal{A}$.

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 $[\![out]\!]_\mathcal{B}$. $\mathcal{S}_\mathcal{B}$ now simulates $\mathcal{B}$'s view:

– Consider Π-`init`. $\mathcal{B}$ receives 4 arrays from $\mathcal{C}$: $\mathbf{m}, \mathbf{s_0}, \mathbf{s_1}, [\![\mathbf{D}]\!]_\mathcal{B}$. The first three are uniformly sampled by $\mathcal{C}$, and hence are trivially simulatable by drawing uniform array entries.

  The last array is a modified $[\![\mathbf{D}]\!]_\mathcal{B}$ that $\mathcal{B}$ initially sent to $\mathcal{C}$. $\mathcal{C}$ masked each entry of $[\![\mathbf{D}]\!]_\mathcal{B}$ with $\mathbf{r}$ that he uniformly sampled and $\mathcal{B}$ never learns. He then extended $[\![\mathbf{D}]\!]_\mathcal{B}$ with a uniform share of $\mathbf{m}$ that is also not known to $\mathcal{B}$. Although each entry of $[\![\mathbf{D}]\!]_\mathcal{B}$ was now masked with a uniformly drawn mask or set to a uniform value, $\mathcal{B}$ could use his initial share of $[\![\mathbf{D}]\!]_\mathcal{B}$ to recover the masks and the shelter entries (and retrieved indices during access). Thus, $\mathcal{A}$

permutes $[\![\mathbf{D}]\!]_{\mathcal{B}}$ according to a random permutation π before sending it to $\mathcal{B}$. $[\![\mathbf{D}]\!]_{\mathcal{B}}$ is now indistinguishable from random.

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 was 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 $\mathbf{pos}_i$ indices were 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 was XORed with uniformly drawn $\mathbf{f}_q$. Hence, $\mathcal{S}_{\mathcal{B}}$ simulates b by drawing random bit.

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

- Consider Π-init. $\mathcal{C}$ receives a uniform secret-share of the input array $\mathbf{d}$ from $\mathcal{B}$. $\mathcal{B}$ generates this share by sampling uniform bits, and hence $\mathcal{S}_{\mathcal{C}}$ simulates it with uniform bits.
- Consider Π-access. $\mathcal{C}$ is not involved in Π-access. Hence, $\mathcal{C}$ need 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 ($\perp$).

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 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 @ 3.50GHz and 64GB 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 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 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 20GB (our plaintext array is 4GB). For all runs of the initialization algorithm, our implementation matches exactly the number of bits incurred by our algorithm.

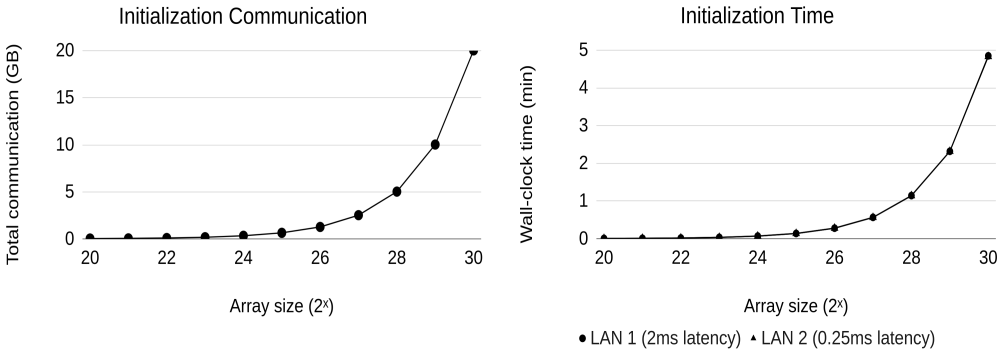

Fig. 4. `Π-init` performance. We fix the number of accesses to 2^{20} and plot the following metrics as functions of the *binary logarithm of the array size*: the overall communication (left) and the wall-clock time to complete the protocol on LAN 1 and LAN 2 (right). Note that the plots for LAN 1 and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

– **Wall-clock time.** For a large 2^{30}-entry array and for 2^{20} accesses, initialization runs for ≈ 4.8 minutes[2]. For a small 2^{20}-entry array with the same number of accesses, initialization takes ≈ 0.5 second. 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 0.019ms per access on localhost for all runs and for all tested array sizes. Communication is 13B[3] 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 5 plots the total communication and the wall-clock time in each network setting.

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

[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 ($1B$).

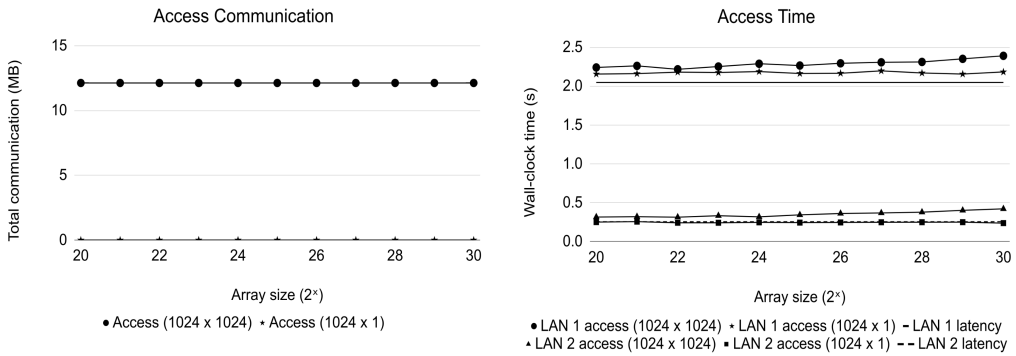

Fig. 5. `Π-access` performance. We consider two parameter regimes for the number of accesses: (1024×1024) and 1024×1. Then we plot the following metrics as functions of the *binary logarithm of the array size*: the overall communication (left) and the wall-clock time to complete the protocol on LAN 1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

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 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 can be packaged with other bits in the batched setting.
- **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 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.

[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`).

References

[AFN+17] Ittai Abraham, Christopher W. Fletcher, Kartik Nayak, Benny Pinkas, and Ling Ren. Asymptotically tight bounds for composing ORAM with PIR. In Serge Fehr, editor, *PKC 2017, Part I*, volume 10174 of *LNCS*, pages 91–120. Springer, Heidelberg, March 2017.

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

[CCMS19] T.-H. Hubert Chan, Kai-Min Chung, Bruce M. Maggs, and Elaine Shi. Foundations of differentially oblivious algorithms. In Timothy M. Chan, editor, *30th SODA*, pages 2448–2467. ACM-SIAM, January 2019.

[CKGS95] Benny Chor, Eyal Kushilevitz, Oded Goldreich, and Madhu Sudan. Private information retrieval. *FOCS*, 1995.

[CKN+18] T.-H. Hubert Chan, Jonathan Katz, Kartik Nayak, Antigoni Polychroniadou, and Elaine Shi. More is less: Perfectly secure oblivious algorithms in the multi-server setting. In Thomas Peyrin and Steven Galbraith, editors, *ASIACRYPT 2018, Part III*, volume 11274 of *LNCS*, pages 158–188. Springer, Heidelberg, December 2018.

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

[FJKW15] Sky Faber, Stanislaw Jarecki, Sotirios Kentros, and Boyang Wei. Three-party ORAM for secure computation. In Tetsu Iwata and Jung Hee Cheon, editors, *ASIACRYPT 2015, Part I*, volume 9452 of *LNCS*, pages 360–385. Springer, Heidelberg, November / December 2015.

[FNO22] Brett Hemenway Falk, Daniel Noble, and Rafail Ostrovsky. 3-party distributed oram from oblivious set membership. *SN*, 2022.

[GIKM98] Yael Gertner, Yuval Ishai, Eyal Kushilevitz, and Tal Malkin. Protecting data privacy in private information retrieval schemes. In *30th ACM STOC*, pages 151–160. ACM Press, May 1998.

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

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

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

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

[JW18] Stanislaw Jarecki and Boyang Wei. 3PC ORAM with low latency, low bandwidth, and fast batch retrieval. In Bart Preneel and Frederik Vercauteren,

editors, *ACNS 18*, volume 10892 of *LNCS*, pages 360–378. Springer, Heidelberg, July 2018.

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

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

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

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

[PKV+14] Vasilis Pappas, Fernando Krell, Binh Vo, Vladimir Kolesnikov, Tal Malkin, Seung Geol Choi, Wesley George, Angelos D. Keromytis, and Steve Bellovin. Blind seer: A scalable private DBMS. In *2014 IEEE Symposium on Security and Privacy*, pages 359–374. IEEE Computer Society Press, May 2014.

[PS06] Phillip Porras and Vitaly Shmatikov. Large-scale collection and sanitization of network security data: risks and challenges. *NSPW*, 2006.

[SCSL11] Elaine Shi, T.-H. Hubert Chan, Emil Stefanov, and Mingfei Li. Oblivious RAM with $O((\log N)^3)$ worst-case cost. In Dong Hoon Lee and Xiaoyun Wang, editors, *ASIACRYPT 2011*, volume 7073 of *LNCS*, pages 197–214. Springer, Heidelberg, December 2011.

[SSS12] Emil Stefanov, Elaine Shi, and Dawn Xiaodong Song. Towards practical oblivious RAM. In *NDSS 2012*. The Internet Society, February 2012.

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

[WHC+14] Xiao Shaun Wang, Yan Huang, T.-H. Hubert Chan, abhi shelat, and Elaine Shi. SCORAM: Oblivious RAM for secure computation. In Gail-Joon Ahn, Moti Yung, and Ninghui Li, editors, *ACM CCS 2014*, pages 191–202. ACM Press, November 2014.

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

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