Scaling ORAM for Secure Computation

Jack Doerner
Northeastern University
j@ckdoerner.net

abhi shelat
Northeastern University
abhi@neu.edu

ABSTRACT

We design and implement a Distributed Oblivious Random Access Memory (DORAM) data structure that is optimized for use in two-party secure computation protocols. We improve upon the access time of previous constructions by a factor of up to ten, their memory overhead by a factor of one hundred or more, and their initialization time by a factor of thousands. We are able to instantiate ORAMs that hold 2^{34} bytes, and perform operations on them in seconds, which was not previously feasible with any implemented scheme.

Unlike prior ORAM constructions based on hierarchical hashing [21], permutation [21], or trees [40], our Distributed ORAM is derived from the new Function Secret Sharing scheme introduced by Boyle, Gilboa and Ishai [11, 12]. This significantly reduces the amount of secure computation required to implement an ORAM access, albeit at the cost of $O(n)$ efficient *local* memory operations.

We implement our construction and find that, despite its poor $O(n)$ asymptotic complexity, it still outperforms the fastest previously known constructions, Circuit ORAM [43] and Square-root ORAM [56], for datasets that are 32 KiB or larger, and outperforms prior work on applications such as *stable matching* [16] or *binary search* [25] by factors of two to ten.

1 INTRODUCTION

In spite of the substantial improvements to the efficiency of two-party secure computation protocols, they still encounter major obstacles when evaluating many types of functions. In particular, functions that make *data-dependent* accesses to memory remain difficult cases. A data-dependent memory access is an access to an element within an array, at an index i that is computed from some secret input. A secure computation protocol must guarantee that no information about its inputs is leaked to either party, even via intermediate computations, and thus it must be able to execute such memory accesses without leaking any bits of i.

Data-dependent memory accesses are common even in textbook algorithms; they are required by, for example, binary search, most graph algorithms, sparse matrix methods, greedy algorithms, and dynamic programming algorithms. More generally, they are required by any program that is written in the RAM model of computation. Any attempt to evaluate such an algorithm in a secure context upon a large dataset certainly requires an efficient data-dependent memory access mechanism.

The simplest solution to this problem is the *linear scan* technique, which hides the index of an accessed element by touching every element in the memory and using multiplexers to ensure that only the desired element is actually read or written. This effectively ensures data-obliviousness, but it requires an expensive secure computation involving $O(n)$ gates for each individual memory access. With accesses incurring overhead linear in the size of the entire memory, scanning is impractical for all but the smallest amounts of data.

Another solution is *Oblivious Random Access Memory* (ORAM). Intuitively, ORAM is a technique to transform a memory access to a *secret* index i into a sequence of memory accesses that can be revealed to an adversary, the indices of which appear independent of i. ORAM was first proposed by Goldreich and Ostrovsky in their seminal paper [21], which studied the general context of client-server memory outsourcing. In this setting, a client wishes to perform a computation on a database of size n, which is held by some untrusted server, but does not want the server to learn the semantic pattern of accesses to the database. Goldreich and Ostrovsky proposed two schemes to solve this problem, the second of which requires that the client perform $O(\text{polylog}\, n)$ accesses to the database for every access in the client's original program. In the subsequent two decades, ORAM techniques have been widely studied [7, 13, 14, 18, 22–24, 30, 34–36, 39, 41, 46–48] with the goals of reducing the communication overhead between the client and server, reducing the amount of memory required of the client, and reducing the server's overall memory overhead. State of the art approaches to ORAM design limit the overhead in all of these measures to $O(\log^c n)$ where $c \leq 3$.

ORAM can be applied to the domain of secure computation by implementing ORAM client operations as secure functions, while the mutually-untrusting computation parties share the role of the ORAM server. This arrangement was proposed by Ostrovsky and Shoup [35], who used it to show that secure computations need not take time linear in the size of their input. It was later taken up by Gordon *et al.* [25]. Subsequently, the development of secure-computation-specific ORAMs began.

Wang *et al.* [44] observed that memory and communication overhead, the metrics for which ORAM had traditionally been optimized, were inappropriate for the context of secure computation. They proposed that *circuit complexity* is a more relevant measure, and described a heuristic ORAM based on this idea. Subsequently, Wang *et al.* [43] proposed Circuit ORAM, which offers asymptotically strong parameters for a data-structure with small circuit complexity.

Zahur *et al.* [56] observed that by relaxing asymptotic bounds, it is possible to produce a scheme that has a smaller concrete circuit size. They described a modification of the original Goldreich-Ostrovsky Square-root ORAM that is asymptotically inferior to Circuit ORAM, but outperforms it for data sizes up to 4 MiB.

Although they represent a dramatic improvement over initial efforts, the ORAM constructions of Gordon *et al.*, Zahur *et al.*, and Wang *et al.* suffer drawbacks. For instance, they are all recursively structured. That is, accessing the top level ORAM data structure for n elements requires recursively accessing another ORAM data structure of size $n/8$ elements, and so on, each layer adding a communication round. As a result, each semantic access requires accessing $O(\log n)$ different ORAM layers, incurring $O(\log n)$ rounds of communication and latency.

These constructions also have high concrete memory overhead, due in part to their recursive nature and to the fact that they store *wire labels* for each bit of their memory, each wire label being at least 80 times larger than the data it represents. All prior research efforts of which we are aware report on concrete experiments that involve at most 2^{20} elements. In our own experiments, we confirm that the constructions they describe cannot handle more elements in a reasonable amount of time and space.[1]

The last, and possibly most significant problem is initialization. In many cases, an ORAM must be filled with some initial data before it can be used. Circuit ORAM requires an individual write into each element, a process that is extremely expensive: we observed it to require more than 3000 seconds for a moderately-sized memory of 2^{15} elements.[2] Zahur *et al.*'s Square-root ORAM is asymptotically similar, but uses a permutation network [42] instead of individual writes to achieve a constant-factor improvement of roughly 100. Nevertheless, even for moderately-sized memories, initialization is a significant cost.

These bottlenecks limit the use of secure computation protocols mostly to data-independent algorithms (e.g. AES [37], edit distance [45], or linear regression [33]) or RAM programs that exploit specific algorithmic properties to restrict their access patterns (e.g. BFS [6], Dijkstra's algorithm [28], or stable matching [16]).

1.1 Contributions

We propose a new data structure that addresses the drawbacks discussed previously, and we demonstrate the first concrete secure computation memory implementation that is capable of hosting data at the scale of many gigabytes. Our scheme has faster access times than all prior constructions for memories that are larger than 32 KiB, and, as it does not have any recursive components, each access requires only three rounds in principle. Unlike prior ORAMs, our data structure supports read and write operations independently, and can perform read operations substantially faster. Instead of storing wire labels, we store either XOR-shares or encryptions of the data, and thereby reduce the memory overhead to a small constant. Additionally, we have a linear-time method to fill our structure with initial data that requires no secure computation. As a result, an instance with 2^{20} 4-byte elements can be initialized in 166 milliseconds, roughly 4000 times faster than the best prior initialization technique from Zahur *et al.*'s Square-root ORAM [56]. We show that our advantages hold not only in microbenchmarks, but also in previously-published application contexts such as binary search and stable matching.

In contrast to most prior secure computation ORAM research, we consider the *Distributed* ORAM model [32], and derive our scheme from two-server Private Information Retrieval (PIR) techniques. In PIR, a client wishes to retrieve an element A^i at index i in database A, copies of which are held by two servers. The client issues a query $q_1(i)$ to server 1 and query $q_2(i)$ to server 2, and the servers respond with short messages m_1 and m_2 respectively, which the client can use to reconstruct A^i. PIR schemes must satisfy two properties: the total communication between client and servers must be sub-linear in n, and the query $q_p(i)$ in isolation must reveal no information about i.

Gilboa and Ishai [17] and Boyle, Gilboa, and Ishai [11] recently presented a surprisingly efficient PIR construction that is based on the notion of a *function secret sharing* (FSS) scheme for a *distributed point function* (DPF). Their construction offers properties new to PIR which make it well-suited for use in an ORAM for secure computation. In particular, it produces a query message of size $O(\log n)$, as opposed to the size of $O(n^{1/3})$ required by many PIR schemes [51], and it requires only a cryptographic pseudo-random generator, whereas other PIR schemes with logarithmic query size require public key cryptography. We discuss the specifics of this primitive in Section 2. In our construction, the parties to the secure computation, Alice and Bob, also act as the two servers in the PIR scheme, and secure computation performs the role of the client. Owing to the efficiency of FSS, our ORAM requires a very small secure computation in comparison to prior ORAM designs (up to one hundred times smaller for the memory sizes that we explore).

The second novel property offered by Boyle *et al.*'s PIR scheme is support for "PIR-writing", which we use to implement ORAM write operations, in combination with a standard *stash* data structure that retains updated elements until they can be reintegrated into the ORAM's main memory. The secure computation needed to implement the stash has an amortized computation and communication complexity of $O(\sqrt{n})$ per access; however, as demonstrated by Zahur *et al.* [56], even schemes with a complexity of $O(\sqrt{n \log^3 n})$ can outperform poly-logarithmic schemes in practice. Our stash reintegration procedure is related to our initialization procedure, and similarly requires linear time with no secure computation.

The *theoretical* disadvantage of our PIR-derived ORAM stems from the fact that the servers in a PIR scheme (i.e., Alice and Bob, in our case) must perform $O(n)$ local computation. This is an unavoidable property of any PIR system. However, unlike the $O(n)$ secure computation required by a traditional linear scan, this computation is simple, highly parallelizable, and enjoys widespread hardware-acceleration support. In practice, secure computation protocols are typically bottlenecked by network or single-core CPU performance and utilize a very small portion of the total computational power and memory bandwidth available with modern hardware; thus, the approach of replacing secure computation with asymptotically-worse local computation can yield significant performance improvements. Despite the poor theoretical complexity of our scheme, we show via a concrete implementation that it outperforms all prior ORAMs, even for large datasets.

Due to the heavy influence of the FSS scheme and the fact that the computation parties make local linear scans of the memory for each operation, we call our ORAM construction Function-secret-sharing Linear ORAM, or *Floram*.

As with most prior ORAM research, our implementation is in the honest-but-curious adversarial setting. We conjecture that our scheme can be hardened more easily than others due to its simplicity, but we leave that question for future work.

Organization. The remainder of the paper is organized as follows: In Section 2, we review definitions of techniques we use, including

[1] Wang *et al.* [43] report on an instance of Circuit ORAM storing 2^{30} 4-byte elements using an older implementation of Circuit ORAM that stores its data as XOR-shares instead of wire labels, but they do not report concrete performance figures for that size. In this paper we evaluate the faster implementation reported by Zahur *et al.* [56]; with this implementation, an instance of Circuit ORAM larger than 64 MiB exhausts the 122 GiB of memory in each of our two test machines.

[2] See Figure 9d

ORAM and the recently developed technique of Function Secret Sharing. In Section 3 we construct simple single-function ORAMs based upon FSS, and analyze their properties, and in Section 4 we combine and extend these constructions to yield a fully functional ORAM. In Section 5 we present a technique for outsourcing the FSS computation that yields a significant practical speed increase over a naïve implementation, and in Section 6 we describe a few additional optimizations. Finally, in Section 7, we describe an implementation of our scheme and evaluate its performance. In the Appendices we give formal definitions and security proofs.

2 BACKGROUND

Secure Multi-party Computation. The field of Secure Multi-Party Computation (MPC) studies mechanisms by which a group of individuals, each individual i having some secret input x_i, can evaluate a function $y = f(x_1, x_2, \ldots)$ jointly, in such a way that no party i learns anything other than what is revealed by the output y and their private input x_i. Specifically, party i must neither learn any x_j for all $j \neq i$, nor any intermediate value derived from x_j during the evaluation of f. A special case of MPC is Two-Party Computation (2PC), in which only two parties, Alice and Bob, participate. Though many variations of MPC have been developed in its thirty-plus year history, and it is likely possible to adapt our work to suit a significant subset of them, this paper focuses on Yao's Garbled Circuits [52, 53].

Yao's Garbled Circuits conforms to the *honest-but-curious* or *semi-honest* security model, in which Alice and Bob are trusted to follow the protocol instructions, but are curious adversaries who may attempt to learn each others' secrets by analyzing protocol transcripts. Outside observers may also analyze protocol transcripts, but must learn nothing in so doing. Selective security for Yao's Garbled Circuits in this model has been proven by Lindell and Pinkas [31], and adaptive security for circuits in NC1 by Jafargholi and Wichs [27]. We provide a standard security definition in Appendix A.1.

Oblivious RAM. ORAM [21] is a data structure that provides the familiar semantics of random access memory, but translates the logical access instructions it receives into sequences of physical accesses in such a way that no adversary can recover the logical accesses by observing the physical access patterns. An ORAM must support the functions $\mathsf{Read}(i)$ and $\mathsf{Write}(i, v)$, which perform semantic reads and writes to locations specified by a private index i. An ORAM may also support functions $\mathsf{Apply}(f, i, v)$, which applies some function privately to a single location, and $\mathsf{Init}(V)$, which fills the ORAM with data from the array V.

As traditionally defined, an ORAM must satisfy the security property that, for any two sequences of logical accesses of the same length, transcripts of the physical accesses produced must be indistinguishable. We concern ourselves with a variant, *Distributed* Oblivious RAM (DORAM) [32], which considers the context wherein the underlying memory is split among multiple parties, and which satisfies a slightly weaker security property: for any two sequences of logical accesses of the same length, transcripts of the physical accesses performed by any *single* party must be indistinguishable. Intuitively, no party may learn anything about the semantic memory by observing their own share of the physical memory. We provide formal definitions for DORAM in Appendix A.2.

ORAMs are traditionally considered to have some manner of secure CPU that transforms semantic memory accesses into physical ones. In the setting of MPC, the CPU is typically implemented as a multiparty protocol. Thus, in some sense, all ORAMs become DORAMs when applied to MPC: the constructions as wholes can be only as secure as the MPC protocols that implement their CPUs, and no protocol can be secure when all participants are corrupt. For simplicity, we refer to our scheme as an ORAM, except where the distinction is important.

Function Secret Sharing. Secret Sharing [38] allows a *dealer* to divide a secret value into m shares, one for each of m parties, such that none of the parties can individually gain any insight into the secret value, yet all m shares, as a group, contain enough information to reconstruct it. Recently, Gilboa and Ishai [17] observed that it is possible to secret-share a point function using shares with sizes sublinear in the size of the function's domain; they call this concept a *Distributed Point Function* (DPF). Boyle *et al.* [11, 12] subsequently improved upon this work and described how to construct a two-server PIR scheme using a DPF. We begin by formally defining a Function Secret Sharing Scheme for two parties.

Definition 2.1 (Point Function). A point function is a function $f_{\alpha,\beta} : [1, n] \to G$ such that

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

Definition 2.2 (Function Secret Sharing Scheme for Point Functions [11, 17]). A two-party function secret sharing scheme is a pair of Probabilistic Polynomial Time algorithms $(\mathsf{Gen}, \mathsf{Eval})$ of the following form

(1) $\mathsf{Gen}(1^\lambda, (\alpha, \beta))$ is a key generation algorithm, which on input 1^λ (a security parameter), and a description of a point function function $f_{\alpha,\beta}$, outputs a tuple of keys $(k_a^{\mathsf{FSS}}, k_b^{\mathsf{FSS}})$.
(2) $\mathsf{Eval}(k_p^{\mathsf{FSS}}, x)$ is an evaluation algorithm, which on input k_p^{FSS} (party key share for party $p \in \{a, b\}$), and evaluation point $x \in [1, n]$, outputs a group element $y_p^x \in G$ and a bit $t_p^x \in \{0, 1\}$ such that $y_p^x = f_p(x)$ (party p's share of $f(x)$) and t_p^x is a share of 0 if $f(x) = 0$, or a share of 1 otherwise.

Definition 2.3 (Security for an FSS Scheme for Point Functions). A two-party FSS for point functions is secure if

(1) (Correctness) For all point functions $f_{\alpha,\beta}$, and for every $x \in [1, n]$ in the domain of $f_{\alpha,\beta}$

$$(k_a^{\mathsf{FSS}}, k_b^{\mathsf{FSS}}) \leftarrow \mathsf{Gen}(1^\lambda, (\alpha, \beta)) \implies$$
$$\Pr\left[\mathsf{Eval}(k_a^{\mathsf{FSS}}, x) - \mathsf{Eval}(k_b^{\mathsf{FSS}}, x) = f(x)\right] = 1$$

(2) (Privacy) For every corrupted party p (either a or b), and every sequence of point function descriptions $f_1, f_2, \ldots$, there exists a simulator Sim such that:

$$\left\{k_p^{\mathsf{FSS}} : (k_a^{\mathsf{FSS}}, k_b^{\mathsf{FSS}}) \leftarrow \mathsf{Gen}(1^\lambda, f_\lambda)\right\}_{\lambda \in \mathbb{N}} \stackrel{c}{\equiv} \left\{\mathsf{Sim}(p, 1^\lambda)\right\}_{\lambda \in \mathbb{N}}$$

In other words, the simulator can produce a share (without knowing the function) that is indistinguishable from the real share for

the function. Thus, the function share leaks nothing about $f_{\alpha,\beta}$ other than its domain and the group that contains its range.

We summarize the FSS construction of a distributed point function $f_{\alpha,\beta}$ from Boyle *et al.* [11, 12] in Figure 1. The $\mathsf{Gen}(1^\lambda, (\alpha, \beta))$ method produces shares $k_a^{\mathsf{FSS}}, k_b^{\mathsf{FSS}}$ of the point function $f_{\alpha,\beta}$. These shares consist of one private seed each (s_a, t_a and s_b, t_b respectively), and the rest of the information in the share is the same for both parties. The FSS scheme follows a tree-based PRF construction, wherein each node of the tree is associated with a seed, and a pseudo-random generator (PRG) is used to double the seed into two seeds, one for the left child, and one for the right. At each level j of the tree, Alice and Bob will have exactly the same seed for all nodes except for the node along the path from the root to the leaf α. At this node, Alice and Bob have different seeds, s_a^{j,α_j} and s_b^{j,α_j} respectively, and thus the expansion of their seeds result in different seeds for the children of this node at level $j+1$, $s_a^{j+1,0}, s_a^{j+1,1}$ and $s_b^{j+1,0}, s_b^{j+1,1}$. The scheme provides a correction word σ^j and two advice bits, $\tau^{j,0}$ and $\tau^{j,1}$, for each level. σ^j is conditionally applied to both child seeds of a node according to $t^j = \mathsf{Lsb}(s_p^{j,\alpha_j}) \oplus t^{j-1} \cdot \tau^{j,\alpha_j}$. This modifies the child seeds such that afterward, Alice and Bob share the same seed for all nodes except for the node along the path to leaf α. That is, of the two children of each node along the path to leaf α, for which Alice and Bob's seeds differ, one is "deactivated" (i.e. Alice and Bob's seeds at that position are made identical), and the other is not. This correction is performed in such a way that neither party can determine which branch has been deactivated.

A Private Information Retrieval (PIR) system is a mechanism by which a client may retrieve an item from a database replicated among some number of servers, without revealing to any server which item was retrieved. Though similar to ORAMs, PIR systems are notably distinct: they typically do not concern themselves with writing or with hiding the contents of the memory from the servers, they do not require any initialization or allow reorganization of the database, and they do not incur memory overheads for the client or servers. On the other hand, PIR schemes take for granted that servers must perform $O(n)$ work for each access, whereas ORAM literature has hitherto focused on providing sublinear-in-n computation complexity. When combined with memory encryption, a PIR scheme may be thought of as an Oblivious Read-only Memory (OROM), and we show how to construct such a primitive from FSS in Section 3.

```
1  function Gen(1^λ, α = α_m ... α_2 α_1, β):
2    s'^0_a, s'^0_b ←$ {0,1}^λ    // pick random seeds
3    t^0_a, t^0_b ← a random xor−share of 1
4    for j ∈ [1, m]:
5      {(s^{j,0}_p ‖ s^{j,1}_p)}_{p∈{a,b}} ← {Prg(s'^{j-1}_p)}_{p∈{a,b}}
6      σ^j ← s^{j,ᾱ_j}_a ⊕ s^{j,ᾱ_j}_b    // xor off-path children
7      τ^{j,0} ← Lsb(s^{j,0}_a) ⊕ Lsb(s^{j,0}_b) ⊕ α_j ⊕ 1
8      τ^{j,1} ← Lsb(s^{j,1}_a) ⊕ Lsb(s^{j,1}_b) ⊕ α_j
9      {s'^j_p}_{p∈{a,b}} ← {s^{j,α_j}_p ⊕ t^{j-1}_p · σ^j}_{p∈{a,b}}
10     {t^j_p}_{p∈{a,b}} ← {Lsb(s^{α_j}_p) ⊕ t^{j-1}_p · τ^{j,α_j}}_{p∈{a,b}}
11   γ ← s^m_a ⊕ s^m_b ⊕ β
12   k^FSS_a ← (s'^0_a, t^0_a, {σ^j, τ^{j,0}, τ^{j,1}}_{j∈[1,m]}, γ)
13   k^FSS_b ← (s'^0_b, t^0_b, {σ^j, τ^{j,0}, τ^{j,1}}_{j∈[1,m]}, γ)
14   return k^FSS_a, k^FSS_b
15
16 function Eval(k^FSS_p, x = x_m ... x_2 x_1)
17   // Parse key k^FSS_p as (s^0_p, t^0_p, {σ^j, τ^{j,0}, τ^{j,1}}_{j∈[1,m]}, γ)
18   for j ∈ [1, m]:
19     (s^{j,0} ‖ s^{j,1}) ← Prg(s'^{j-1})
20     s'^j ← s^{j,x_j} ⊕ t^{j-1} · σ^j
21     t^j ← Lsb(s^{j,x_j}) ⊕ t^{j-1} · τ^{j,x_j}
22   y ← s'^m ⊕ t^m · γ
23   return y, t^m
```

Figure 1: Pseudocode for the Function Secret Sharing scheme. Our design follows Boyle *et al.* [11, 12].

3 SINGLE-FUNCTION MEMORY

We begin by explaining how to construct write-only and read-only random access memories from the FSS scheme described in Section 2. The constructions presented here may be independently useful in scenarios wherein simultaneous read and write capabilities are not needed; we combine them into a full ORAM in Section 4.

Oblivious Write-Only Memory. We first construct an Oblivious Write-Only Memory (OWOM), based on the folkloric technique of *PIR-writing*. Both parties hold a local XOR-share of each memory location; in order to write to a location i (this index being given as private data within the MPC protocol), the secure computation must determine the difference, v^Δ, between the value already stored there and the value to be written. It must then use the FSS scheme to construct a distributed point function that evaluates to 0 everywhere except location i, whereat the DPF evaluates to v^Δ. Alice and Bob individually evaluate their shares of the DPF, and add these shares into the memory-shares that they hold. Because they are adding shares of zero at all locations other than i, those values remain unchanged. At index i, they add shares of the difference between the old and new values to shares of the old value, producing shares of the value that was to be written.

More precisely, we represent the value at memory location i as W^i, and party p's share as W_p^i, where $W^i = W_a^i \oplus W_b^i$. To write value W'^i into the memory, the secure computation calculates $v^\Delta = W^i \oplus W'^i$ and then $(k_a^{\mathsf{FSS}}, k_b^{\mathsf{FSS}}) \leftarrow \mathsf{Gen}(1^\lambda, (i, v^\Delta))$, delivering k_a^{FSS} to Alice and k_b^{FSS} to Bob, who use these keys to derive $(y_p^x, t_p^x) \leftarrow \mathsf{Eval}(k_p^{\mathsf{FSS}}, x)$ for all $x \in [1, n]$. For the purpose of writing, the parties will ignore t_p^x and use the main DPF output y_p^x, which they XOR into the underlying memory to perform the write, $W_p'^x \leftarrow W_p^x \oplus y_p^x$.

Because write operations are performed by *cumulatively* XORing adjustment values with each W^i, it is necessary to write the difference between the old and new values, rather than writing the new value directly. In absence of any mechanism for reading (or otherwise determining which values are currently stored), this limits our OWOM to use only in write-only, write-once situations.

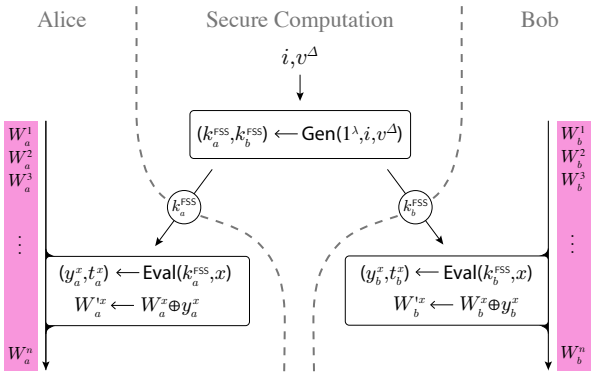


Figure 2: Diagram of Oblivious Write-only Memory. To perform a write, the secure computation generates shares of a DPF, k_a^{FSS} and k_b^{FSS}, which are distributed to Alice and Bob. Alice and Bob each evaluate the DPF at every value $x \in [1, n]$ and XOR the result into their respective corresponding shares of the OWOM memory.

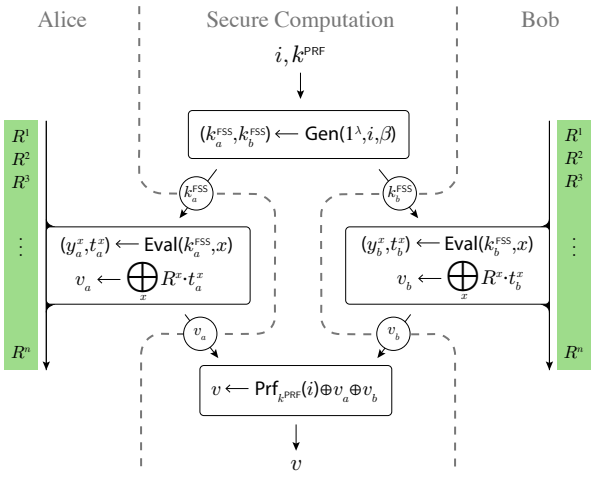


Figure 3: Diagram of Oblivious Read-Only Memory. To perform a read, the secure computation generates shares of a DPF, k_a^{FSS} and k_b^{FSS}, which are distributed to Alice and Bob. Alice and Bob each evaluate a normalized version of the DPF at every value $x \in [1, n]$, calculate the dot product of the normalized DPF with their respective copies of the OROM memory, and feed the result back into the secure computation to compute the value v at location i.

However, it will become a building block for a full ORAM in the next section. We depict this scheme in Figure 2.

Oblivious Read-Only Memory. We implement read-only memory in a manner similar to classic PIR constructions. Alice and Bob, in their roles as the PIR servers, each hold identical copies of the memory, masked by the output of a pseudo-random function (PRF) using a key k^{PRF} that is known to the secure computation, but not to Alice or Bob individually. To read an element R^i from the memory at a private index i (again, this index is given as private data within the protocol), Alice and Bob engage in a secure computation protocol to calculate $(k_a^{\text{FSS}}, k_b^{\text{FSS}}) \leftarrow \text{Gen}(1^{\lambda}, (i, \beta))$. Each party receives a k_p^{FSS} and uses it to calculate $(y_p^x, t_p^x) \leftarrow \text{Eval}(k_p^{\text{FSS}}, x)$ for all $x \in [1, n]$. Although the DPF y_p^x may have an arbitrary range β, for the purpose of reading, it is necessary that they hold a DPF of magnitude 1. Thus, the parties will use the final advice bits, t_p^x, which essentially represent the same DPF normalized to $\{0, 1\}$. Both parties compute $v_p = \bigoplus_x t_p^x \cdot R^x$. According to the properties of our FSS scheme, since $t_a^x = t_b^x$ for all $x \neq i$, it follows that $v_a \oplus v_b = R^i$. Finally, Alice and Bob use a secure computation to evaluate $\text{Prf}_{k^{\text{PRF}}}(i) \oplus R^i$, effectively importing the semantic value of interest into the secure computation. We depict this scheme in Figure 3.

Though this scheme permits an unlimited number of reads, it cannot be written. Each party stores a PRF-masked copy (i.e. an encryption) of the data rather than a secret share: were any single memory location to be changed by a write, the access pattern would be revealed; on the other hand, if *all* memory locations were changed during a write, the semantic values of those not being updated must be destroyed.

Complexity Analysis. For both schemes, the secure FSS component (which forms the bulk of the secure computation) is identical. The computation of $\text{Gen}(1^{\lambda}, (\alpha, \beta))$ requires $4 \log_2(n)$ evaluations of the PRG function, along with some basic boolean operations. It must be seeded with random data of length $O(\lambda)$, and it produces an output of size $O(\lambda \log n)$ where λ is the security parameter. This output can be revealed to the computation parties all at once, or incrementally, in $\log n$ chunks of λ bits, one for each layer of the FSS scheme. In the former case, the secure component incurs a memory complexity of $O(\lambda \log n)$ and $O(1)$ communication rounds. In the latter case, the secure component incurs a memory complexity of $O(\lambda)$, and no additional rounds, as the secure computation does not need to wait for replies. In either case, the communication and computation complexities are $O(\lambda \log n)$.

Subsequently, a local computation is required to construct the DPF, $(y_p^x, t_p^x) \leftarrow \text{Eval}(k_p^{\text{FSS}}, x)$ for all $x \in [1, n]$. If all n FSS evaluations are combined into a single operation, then the FSS tree can be constructed in its entirety only once, requiring $O(n)$ PRG calls. In the case of a write, each of the n elements in the output DPF's domain must be XORed into the corresponding memory location; in the case of a read, the dot product of the DPF and the memory must be taken instead. In either case, this incurs $O(n)$ memory accesses. All of the operations performed by the local FSS evaluation and the application of the output DPF are highly parallelizable. We make extensive use of this fact in our concrete implementation, and in Section 7 we show experimentally that the local component does not become a significant burden until the amount of data stored is on the order of hundreds of megabytes.

4 READING AND WRITING

We now combine the OWOM and OROM from Section 3 into an ORAM construction. We need a few building blocks in order to make this combination possible, and conjecture that these building blocks are sufficient for the combination of any PIR and PIR-writing schemes into an ORAM, assuming that the schemes themselves are suitable (that is, their access patterns and underlying memory formats are secure).

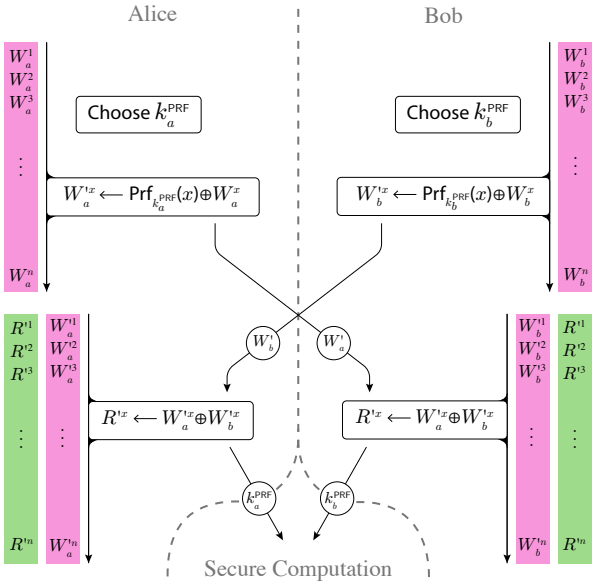


Figure 4: Diagram of the Floram Refresh method. In addition to the operations illustrated here, the secure computation must clear the stash.

At a high level, the construction works as follows: we initialize both an OROM and an OWOM with the same data, and create a linear-scan *stash* that stores elements while they are waiting to be returned to the main memory. *Read* operations are performed by inspecting both the stash and the OROM, and returning the most recent data. *Write* operations are performed by first reading the current value at the specified index, using it to calculate the difference necessary to correctly update the OWOM, and finally writing the new value into both the OWOM and the stash. When the stash fills, we perform a refresh operation to convert the OWOM memory into OROM memory, and then clear the stash. The cost of this refresh can be amortized over the refresh period of the construction. Because we use this stash-and-refresh technique, our amortized secure computation complexity becomes $O(\sqrt{n})$.

Refresh Procedure. To refresh our ORAM construction, we need to convert the underlying memory of an OWOM into the underlying memory of an OROM. The former stores its data as XOR-shares, while the latter uses a masked copy of the data as the underlying format. We can avoid incurring any secure computation overhead at all if, instead of masking the OROM memory only once, using a key known only to the secure computation, we mask it first with a key known only to Alice, and then with a key known only to Bob. To convert the OWOM into an OROM, Alice and Bob mask their local OWOM memory shares using two PRFs with individual secret keys, k_a^{PRF} and k_b^{PRF}.

$$W'_p \leftarrow \left\{ W'^x_p \leftarrow \mathsf{Prf}_{k_p^{\mathsf{PRF}}}(x) \oplus W^x_p \right\}_{x \in [1,n]}$$

They each transmit their masked OWOM memory share to the other party, and both parties calculate

$$R' \leftarrow \left\{ R'^x \leftarrow W'^x_a \oplus W'^x_b \right\}_{x \in [1,n]}$$

Finally, each party feeds their key k_p^{PRF} into the secure computation, so that the OROM memory can be unmasked via $v \leftarrow \mathsf{Prf}_{k_a^{\mathsf{PRF}}}(x) \oplus \mathsf{Prf}_{k_b^{\mathsf{PRF}}}(x) \oplus R^x$. This refresh procedure is illustrated in Figure 4. Unlike previous Square-root ORAM constructions [21, 56], our refresh procedure does not require access to the stash. Instead, we simply clear it. Our stash serves only the purpose of allowing updated elements to be accessed multiple times between refreshes.

Semi-private Access. It may be the case that some algorithms call for both private (i.e. data-dependent) and data independent accesses to the same memory. Ostrovsky and Shoup refer to the latter type of accesses as *semi-private* [35]. To our knowledge, it has heretofore been necessary to implement *all* accesses as fully private accesses in such a scenario, or to perform costly import and export operations upon the entire ORAM. Floram, however, allows for a secondary, semi-private access mechanism, which has a significantly reduced asymptotic and practical cost. Unlike all other ORAMs of which we are aware, Floram stores each memory element at the physical address corresponding to its semantic index. Thus, to read the element at the publicly known semantic index i, the two parties feed their OWOM memory shares W_a^i and W_b^i into the secure computation, which computes the value W^i in $O(1)$ complexity (and potentially using only free gates [29]). Semi-private writes must additionally append to the stash.

Private Read Access. Read operations that are *publicly known* to be read operations can also be performed without invoking the full-access mechanism: neither a write to the stash nor a write to the OWOM is required. Because no write to the stash is required, ORAM reads do not contribute to the refresh period.

Full Private Access. A full private access accepts some arbitrary oblivious function f and applies it to a single element within the ORAM. f takes an ORAM element and some auxiliary input v^f, and produces a new element and some auxiliary output y^f. We use this general-purpose mechanism to implement ORAM writes via simple f_{write} that returns v^f as the output element. To perform a full access, our scheme first retrieves the desired element from the OROM, then scans the stash to determine whether a newer version of the same element exists. f is then applied to it. Finally, the result is stored using an OWOM operation and appended to the stash. Because the OROM and OWOM access the same element, they can share a single FSS evaluation. This process is illustrated in Figure 5.

Initialization. The initialization of our ORAM can be performed efficiently using the mechanism for refreshing that we described earlier. That is, assuming that the parties begin with some secret sharing of the data values with which the ORAM is to be filled, they may initialize it by copying those shares into the OWOM's memory and performing a refresh. If the ORAM is hosted by a Yao's Garbled Circuits protocol, then the point-and-permute technique of Beaver *et al.* [4] can be used to encode XOR shares of the data within the protocol's wire labels, effectively making the generation of shares a free action. Furthermore, because this technique encodes the XOR sharing of each data bit only in the final bit of a much larger wire-label, it is actually a significant constant factor *faster* to initialize our ORAM than it is to perform a single linear scan on the same data. To our knowledge, this property is unique among all known ORAMs.

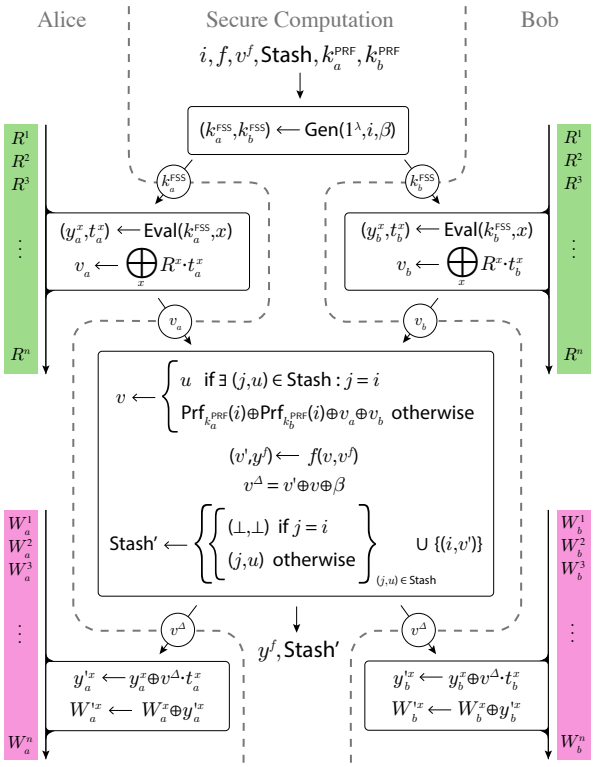


Figure 5: Diagram of the Floram Access method. Note that β is randomly chosen on each access.

Complexity Analysis. If we briefly set aside the stash, the complexities of our scheme for full access to private indices closely follow the complexities of the individual components described in Section 3. That is, each access requires a single FSS Gen execution within the secure context, incurring $O(\log n)$ communication and secure computation, followed by the evaluation of the DPF at all points in its domain, incurring $O(n)$ local computation by both parties. This is in turn followed by a memory scan for the ROM component, adding a further $O(n)$ local computation, an unmasking within the secure computation context, which accounts for $O(1)$ communication and secure computation complexity, and a local memory scan for the WOM component, which incurs a further $O(n)$ local computation. Thus, still ignoring the stash, a standard access operation incurs $O(\log n)$ secure computation and communication overall, as well as $O(n)$ local computation.

The stash must be traversed on each access, and its length depends upon the refresh period of the ORAM. The refresh operation requires a simple masking (i.e. encryption), transmission, and element-wise XOR of n memory elements by each of the two parties, without any secure computation. Thus the total cost of a refresh is $O(n)$ in terms of local computation and communication. This is optimally amortized over $O(\sqrt{n})$ accesses, and thus the cost of each access must include the cost of scanning $O(\sqrt{n})$ elements in the stash. The optimal constant can be determined by the relative costs of secure and local scans. Our concrete implementation uses a stash of size $\sqrt{n}/8$. A summary of these costs, along with comparisons to other ORAM schemes, is provided in Table 1.

The asymptotic complexity of our initialization procedure is $O(n)$ in terms of local computation, memory, and communication. Like the refresh procedure on which it is based, it requires no secure computation at all. This is optimal, at least from a complexity standpoint. Furthermore, as we shall see in Section 7, the practical costs of our initialization procedure are so low that it is actually faster in practice than a simple `memcpy` over the same data.

Comparison to other ORAM schemes. Our ORAM scheme stands in contrast to those that have preceded it in a number of respects, as summarized in Table 1. Here we discuss their implications. We focus primarily on the secure component of our scheme (which cannot be parallelized), and explore the practical consequences of the local component in Section 7. Although our ORAM uses a simple stash that incurs square-root overhead, it does not use recursive position maps or permutations required by Zahur *et al.*'s construction [56], nor does it need the sorting and binary searching required by the classic Goldreich and Ostrovsky construction [21]. Consequently, its optimal stash size is much smaller. Moreover, our scheme can be refreshed more efficiently than that of Zahur *et al.*, and much more efficiently than classic Square-root ORAM, which requires $O(n)$ encryptions within the secure context as well as an oblivious sort for each refresh operation. In previous Square-root ORAM constructions, stash scan and amortized refresh operations accounted for the vast majority of per-access cost; in having provided asymptotic improvements to both (as well as significant constant cost improvements), we have made our new ORAM far more suitable than its predecessors for handling large data sizes. On the other hand, our ORAM requires $O(\log n)$ calls to a PRG within the secure context for each access. Because these PRG calls are expensive, our ORAM is less suitable than that of Zahur *et al.* for small data sizes. In Section 5, we describe a method for reducing the number of secure PRG calls to $O(1)$ at the cost of incurring $O(\log n)$ communication rounds. This significantly improves our performance for small values of n, but for *very* small values, the construction of Zahur *et al.* remains more efficient in practice.

A comparison to Circuit ORAM (and other tree-based ORAMs) is somewhat less straightforward. Our ORAM enjoys an initialization procedure many orders of magnitude more efficient; however, in terms of access complexity, Circuit ORAM remains ahead. Nonetheless, as we shall discuss in Section 7, reduction in constant costs renders our scheme far more efficient in practice. Boyle *et al.* [10] propose a parallelization method for tree-based ORAMs, from which it is possible to derive an initialization procedure that uses permutations in place of individual writes. With this mechanism, Circuit ORAMs could achieve initialization performance similar to that of Zahur *et al.*'s construction, at best.[3] Although the local component of our ORAM is highly parallelizable, no equivalent parallelization scheme for our secure component is possible.

Finally, it is worthwhile to acknowledge the distinctions between our scheme and the recent work of Abraham *et al.* [2], which also combined ORAM with PIR. Like Floram, their scheme is properly a *Distributed* ORAM, but in contrast, their scheme uses PIR to

[3]This mechanism has not yet been implemented, so we cannot currently provide concrete data to support this claim.

	Access				Initialization			
	Floram	Florom	Square-root	Circuit	Floram	Florom	Square-root	Circuit
Secure Comp.	$O(\sqrt{n})$	$O(\log n)$	$O(\sqrt{n \log^3 n})$	$O(\log^3 n)$	–	–	$O(n \log^2 n)$	$O(n \log^3 n)$
Local Comp.	$O(n)$	$O(n)$	$O(\sqrt{n \log n})$	$O(1)$	$O(n)$	$O(n)$	$O(n \log n)$	$O(1)$
Communication	$O(\sqrt{n})$	$O(\log n)$	$O(\sqrt{n \log^3 n})$	$O(\log^3 n)$	$O(n)$	$O(n)$	$O(n \log^2 n)$	$O(n \log^3 n)$
Rounds	$O(1)$	$O(1)$	$O(\log n)$	$O(\log n)$	$O(1)$	$O(1)$	$O(\log n)$	$O(n \log n)$

Table 1: Access and Initialization Complexities. Complexities include amortized refresh operations where relevant. *Florom* refers an instantiation of Floram with a stash size of zero (i.e. one which has recently been refreshed); due to the fact that only writes increase the stash size, refreshes can be forced before long sequences of reads to achieve these complexities.

retrieve single elements along the branches of a larger recursive tree ORAM. Consequently, it shares more with Circuit ORAM and Onion ORAM [15] than it does with our scheme. They optimize for communication overhead, and their scheme achieves a communication complexity of $O(\log n)$ per access, which we can match only when no writes are performed. Furthermore, it is likely that PIR-server computation is significantly less burdensome in their scheme, since their PIR requires no PRG and is evaluated over only $O(\log n)$ elements. On the other hand, they primarily consider the outsourcing model, and do not account for costs in an MPC context. We find it likely[4] that these would be similar to Circuit ORAM.

Security Analysis. To argue that our scheme is semi-honest secureunder the definition of security given in Appendix A.2, we must present a simulator that produces a party's view of an ORAM operation (without receiving any information about other parties' private inputs) that is indistinguishable from the same party's view of the real ORAM operation. Simulators for access and initialization, along with proofs of computational indistinguishability, are presented in Appendix B.1. Informally, the security of our scheme follows from the security properties of the MPC technique chosen to host the construction and the security of the FSS scheme, which guarantees that the neither the FSS key share nor the output leaks any information about the associated point function, other than its domain and range. The underlying memory itself reveals nothing about its contents due to its mechanism of representation: each party views an OROM memory that is masked by the output of a PRF for which they key is not known, as well as an information-theoretically secure secret-share of an OWOM memory

PRG and PRF. Among several options for the PRG, we have chosen AES-128 [1]. Significant research effort has been put toward optimizing the boolean-circuit representation of AES [8, 50], and these optimizations have naturally been adapted for the context of secure computation [26]. Specifically, we use the AES S-box circuit of Boyar and Peralta [9], which requires less than 5000 non-free gates per block, and we accelerate local AES evaluations using Intel's AES-NI instruction set. In order to avoid the cost of repeated key expansion, we assume that AES satisfies the ideal cipher property and use the Davies-Meyer construction [49], with independent keys for left and right expansions in the FSS tree. We use AES in counter mode as the PRF that masks the OROM.

[4] As we have no implementation of their scheme (MPC-oriented or otherwise), we cannot perform a practical evaluation.

5 CONSTANT SECURE PRG EVALUATIONS

The costliest single component of our scheme is the repeated evaluation of the PRG function within the secure computation of the FSS Gen algorithm. In this section, we present an optimization that can be used to achieve a significant constant-factor speed improvement relative to a naïve implementation by *outsourcing* the evaluations of the PRG in the FSS Gen algorithm to Alice and Bob. That is, instead of Alice and Bob performing a *single* secure computation which uses $O(\log n)$ PRG expansions to compute their shares of the FSS key (line 5 in Figure 1), we instead divide Gen into $m = \log_2 n$ iterative computations that compute the FSS key one part at a time. Surprisingly, we can divide the computation in a manner that requires no PRG evaluations inside the secure computation, and that also maintains the security properties of the original.[5] Specifically, we devise an equivalent method of computing the value σ^j (line 6 in Figure 1) that does not require the PRG to be evaluated in a secure computation. Hereafter, we refer to this as the *Constant PRG* or CPRG optimization.

Thus far, our FSS notation has only identified seeds s_p^{j,α_j} that are on the path from the root to the leaf α in the FSS evaluation tree. We now introduce notation to identify *all* of the nodes in the evaluation tree. Let $S_p^{j,\ell}$ denote the ℓ^{th} node from the left at level j of player p's FSS evaluation tree, where $p \in \{a, b\}$, $j \in [1, m]$, and $\ell \in [0, 2^j)$. Thus, seed s_a^{j,α_j} can also be identified as node S_a^{j,α_j^*} where α_j^* is the integer with the binary representation $\alpha_j \ldots \alpha_2 \alpha_1$.

Next, we observe that the FSS construction guarantees that at any level j, $S_a^{j,\ell} = S_b^{j,\ell}$ for all $\ell \neq \alpha^*$ (that is, for all nodes *except* the one along the path to leaf α), and $S_a^{j,\alpha_j^*} \neq S_b^{j,\alpha_j^*}$. It follows that all of the PRG expansions of the nodes at level j, i.e., the *uncorrected children* at level $j+1$, are equal except for the two children of the node along the path to α. Finally, consider the sum of the PRG expansions of $S_p^{j,\ell}$ for $\ell \in [0, 2^j)$:

$$\left(z_p^{j+1,0} \middle\| z_p^{j+1,1}\right) = \bigoplus_{\ell \in [0,2^j)} \text{Prg}\left(S_p^{j,\ell}\right)$$

[5] i.e., we will still be able to simulate the view of Alice or Bob given only the output of the function. Notice that we would not be able to simulate the view if our protocol simply asked Alice and Bob to evaluate line 5 in Figure 1.

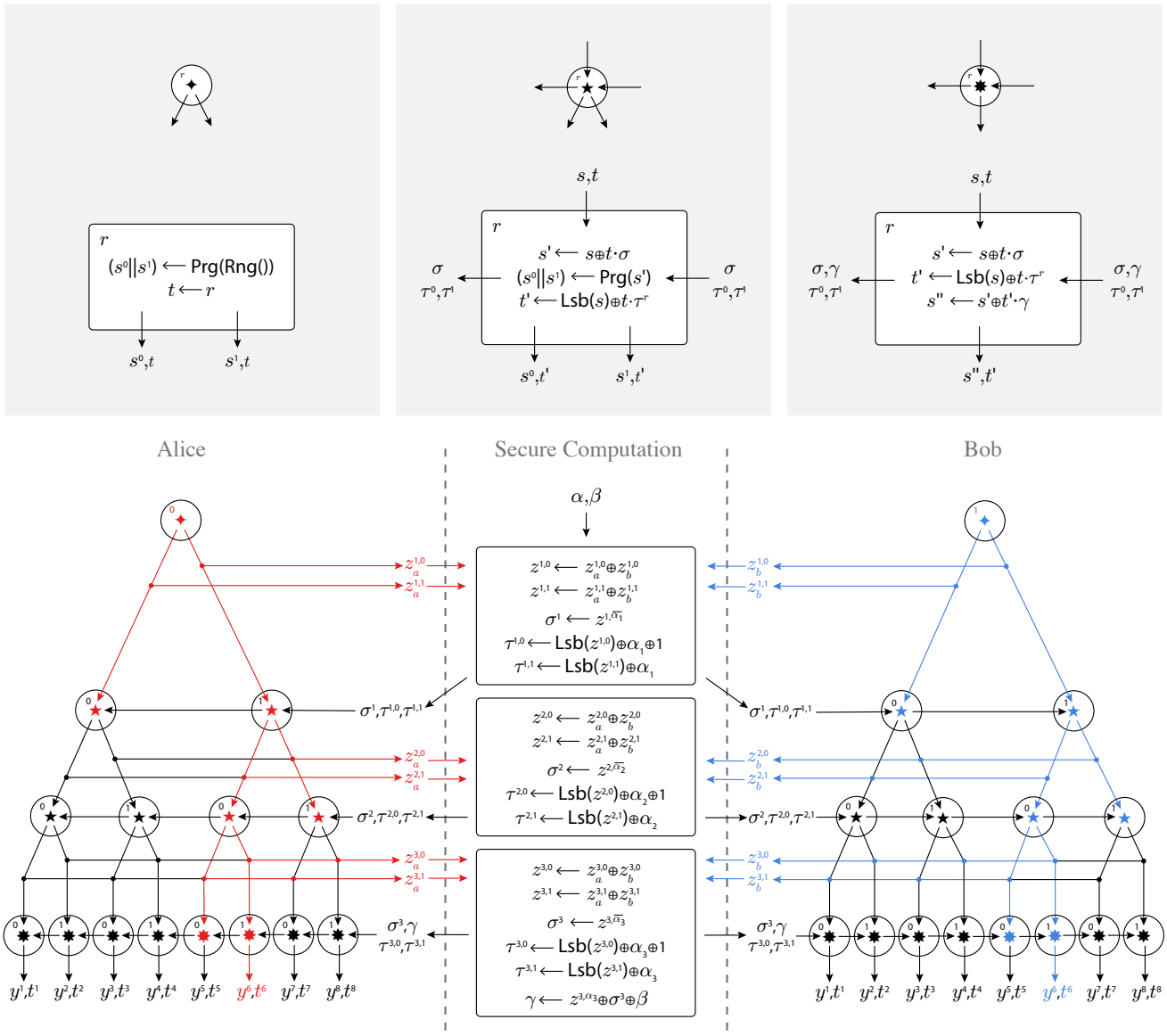
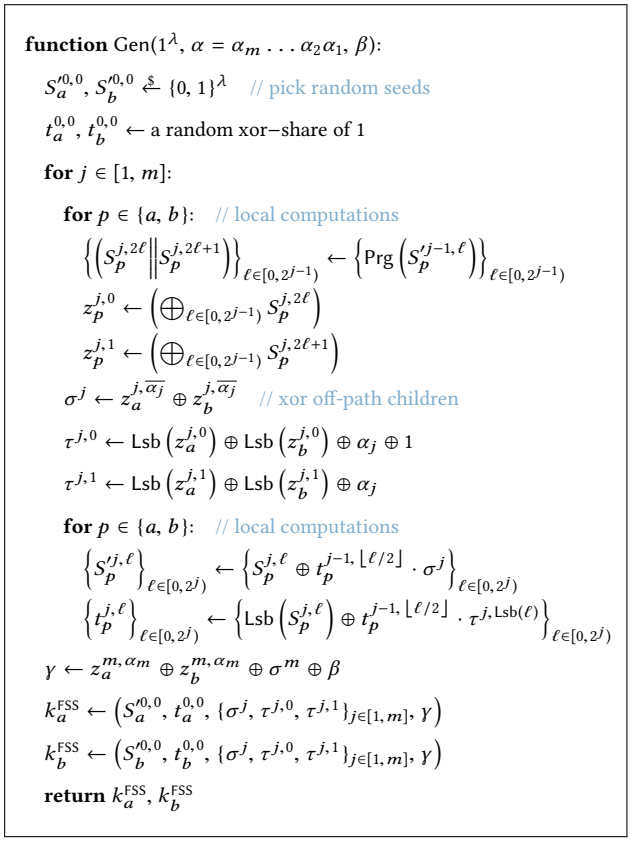


Figure 6: Diagram of the modified Gen/Eval algorithm used by the CPRG optimization. Variables and processes for which Alice and Bob's views are identical are rendered in black. Variables and processes for which Alice and Bob's views differ are rendered in red for Alice and blue for Bob. In this example, $n = 8$, $m = 3$, and α is a three-bit number with value 6.

From the above, we have:

$$z_a^{j,0} \oplus z_b^{j,0} = s_a^{j,0} \oplus s_b^{j,0}$$
$$z_a^{j,1} \oplus z_b^{j,1} = s_a^{j,1} \oplus s_b^{j,1}$$
$$\sigma^j = z_a^{j,\overline{\alpha_j}} \oplus z_b^{j,\overline{\alpha_j}}$$

Thus, we instruct Alice and Bob to locally compute $z_p^{j,0}$ and $z_p^{j,1}$ by accumulating the XOR of all left children and all right children at each level. These two values are submitted to a secure computation, which selects the correct sum using bit $\overline{\alpha_j}$, computes the next advice words $(\sigma^j, \tau^{j,0}, \tau^{j,1})$ and returns them to both parties. Both parties can then apply these values (per lines 9–10 in Figure 1) to generate the corrected seeds for all nodes at the next level, and then continue the process until level m. Revised pseudocode is presented in Figure 7. Although we model this function as returning a pair of key values $(k_a^{\mathsf{FSS}}, k_b^{\mathsf{FSS}})$, note that most components of each party's key are revealed to them over the course of the function, and furthermore, that both parties will have had to perform most of the work of evaluating $\mathsf{Eval}(k_p^{\mathsf{FSS}}, x)$ for all $x \in [1, n]$ in order to calculate $(z_p^{j,0}, z_p^{j,1})$. Consequently, in practice, the CPRG-optimized Gen algorithm returns only those key components that have not already been revealed, and Alice and Bob evaluate Eval simultaneously with the evaluation of Gen. This process is illustrated in Figure 6.



function $\mathsf{Gen}(1^\lambda, \alpha = \alpha_m \ldots \alpha_2\alpha_1, \beta)$:

$S_a'^{0,0}, S_b'^{0,0} \xleftarrow{\$} \{0,1\}^\lambda$ // pick random seeds

$t_a^{0,0}, t_b^{0,0} \leftarrow$ a random xor−share of 1

for $j \in [1, m]$:

for $p \in \{a, b\}$: // local computations

$\left\{\left(S_p^{j,2\ell} \middle\| S_p^{j,2\ell+1}\right)\right\}_{\ell\in[0,2^{j-1})} \leftarrow \left\{\mathsf{Prg}\left(S_p'^{j-1,\ell}\right)\right\}_{\ell\in[0,2^{j-1})}$

$z_p^{j,0} \leftarrow \left(\bigoplus_{\ell\in[0,2^{j-1})} S_p^{j,2\ell}\right)$

$z_p^{j,1} \leftarrow \left(\bigoplus_{\ell\in[0,2^{j-1})} S_p^{j,2\ell+1}\right)$

$\sigma^j \leftarrow z_a^{j,\overline{\alpha_j}} \oplus z_b^{j,\overline{\alpha_j}}$ // xor off-path children

$\tau^{j,0} \leftarrow \mathsf{Lsb}\left(z_a^{j,0}\right) \oplus \mathsf{Lsb}\left(z_b^{j,0}\right) \oplus \alpha_j \oplus 1$

$\tau^{j,1} \leftarrow \mathsf{Lsb}\left(z_a^{j,1}\right) \oplus \mathsf{Lsb}\left(z_b^{j,1}\right) \oplus \alpha_j$

for $p \in \{a, b\}$: // local computations

$\left\{S_p'^{j,\ell}\right\}_{\ell\in[0,2^j)} \leftarrow \left\{S_p^{j,\ell} \oplus t_p^{j-1,\lfloor\ell/2\rfloor} \cdot \sigma^j\right\}_{\ell\in[0,2^j)}$

$\left\{t_p^{j,\ell}\right\}_{\ell\in[0,2^j)} \leftarrow \left\{\mathsf{Lsb}\left(S_p^{j,\ell}\right) \oplus t_p^{j-1,\lfloor\ell/2\rfloor} \cdot \tau^{j,\mathsf{Lsb}(\ell)}\right\}_{\ell\in[0,2^j)}$

$\gamma \leftarrow z_a^{m,\alpha_m} \oplus z_b^{m,\alpha_m} \oplus \sigma^m \oplus \beta$

$k_a^{\mathsf{FSS}} \leftarrow \left(S_a'^{0,0}, t_a^{0,0}, \{\sigma^j, \tau^{j,0}, \tau^{j,1}\}_{j\in[1,m]}, \gamma\right)$

$k_b^{\mathsf{FSS}} \leftarrow \left(S_b'^{0,0}, t_b^{0,0}, \{\sigma^j, \tau^{j,0}, \tau^{j,1}\}_{j\in[1,m]}, \gamma\right)$

return $k_a^{\mathsf{FSS}}, k_b^{\mathsf{FSS}}$

Figure 7: Pseudocode for the Constant PRG optimization applied to the FSS Gen **method. This optimization is discussed in Section 5.**

Security Analysis. Relative to the original Gen algorithm, nothing additional is revealed to either party, i.e., the output of the CPRG-optimized Gen is exactly the same, and the view of each party can be easily simulated with the final key. The only difference is that the advice strings included in the output key are revealed one by one. In the honest-but-curious setting that we consider here, the adversary has no additional power when receiving outputs in this manner.

Efficiency Analysis. The CPRG optimization requires no calls to the PRG function within the secure evaluation of Gen, and only two calls to the PRF to unmask the value retrieved from the OROM. We still perform $O(\log n)$ differencing and advice bit generation steps, but these require only a handful of gates each. On the other hand, our local stage now requires a reduction to be performed over all of the blocks in each layer of the FSS Eval algorithm. Consequently, this variant is significantly more efficient for small and medium sized memories, where secure computation dominates total runtime, but slightly less efficient for memories on the scale of gigabytes, as shown by our evaluations in Section 7.

6 TECHNIQUES AND OPTIMIZATIONS

In this section we present a few additional optimizations that we employ to improve the practical performance of Floram.

6.1 Tree Trimming

During private read operations (that is, accesses wherein the index i is private but the operation is publicly known to be a read), the scheme as previously described generates a full FSS tree with one leaf per ORAM element, but uses only the DPF t and never the DPF y. As an optimization, we can truncate the last $\log_2(\log_2(|G|))$ levels of the FSS tree, split each leaf into individual bits, and set $\beta = 2^{(i \mod \log_2(|G|))}$ such that the bits formed from y are equivalent to the bits t would otherwise have held. As can be seen in Figure 8, in both the standard FSS and CPRG cases these last levels (seven in our implementation) are by far the most expensive.

In the standard FSS case, we may save some additional time by trimming the root of the tree. The first five iterations of the loop in the FSS Gen algorithm expand a single seed into 32. In our implementation (*without* the CPRG optimization), these five loops account for roughly 100,000 non-free gates in the secure computation. As an optimization, we eliminate them, and instead collect enough random coins from each party to generate 32 seeds directly, and include all of them in the output keys. This increases the input size of the secure computation that evaluates Gen, but the savings are nonetheless substantial. The Eval method is similarly changed to index the correct starting seed from the 32 in the key.

6.2 Multithreading and Scheduling

We interleave several steps of our ORAM for efficiency. First, as the secure computation produces the output of Gen, we use separate threads to begin the local Eval steps. This interleaving incurs no additional round trips and does not increase communications costs, and thus it can only improve timing. Second, the stash scan does not depend on the FSS construction or the OROM and can be performed simultaneously with the final layer of the FSS Eval and the OROM memory scan. In the case of the CPRG optimization, it can also be interleaved with the secure FSS Gen function. Together, these optimizations allow non-dominant components of our ORAM scheme to effectively disappear behind dominant components, an effect that is illustrated in the concrete benchmarks that we present in Section 7. Using the benchmarking setup described in Section 7, and an instrumented version of our code-base, we recorded a detailed wall-clock profile, to illustrate both the temporal layout of our scheduling strategy as it appears in practice, and the relative costs of Floram's various parts. We recorded this profile both for standard Floram, and for the CPRG variant, for ORAMs of 2^{20} and 2^{30} 4-byte elements. The results are presented as a diagram in Figure 8.

7 EVALUATION

Experimental Setup. We implemented and benchmarked Floram, using Obliv-C [54], a C derivate that compiles and executes Yao's Garbled Circuits protocols [52] with many protocol-level optimizations [4, 5, 26, 29, 55]. Additionally, we made use of Obliv-C-based Square-root and Circuit ORAM implementations that were provided by the original authors of those works and are identical to the ones reported on previously by Zahur *et al.* [56].

We created two variants of our ORAM, one using the basic construction described in Section 4, and the other using the CPRG method from Section 5. Both variants have optimized scheduling, as described in Section 6.2. Our concrete implementation uses a 128

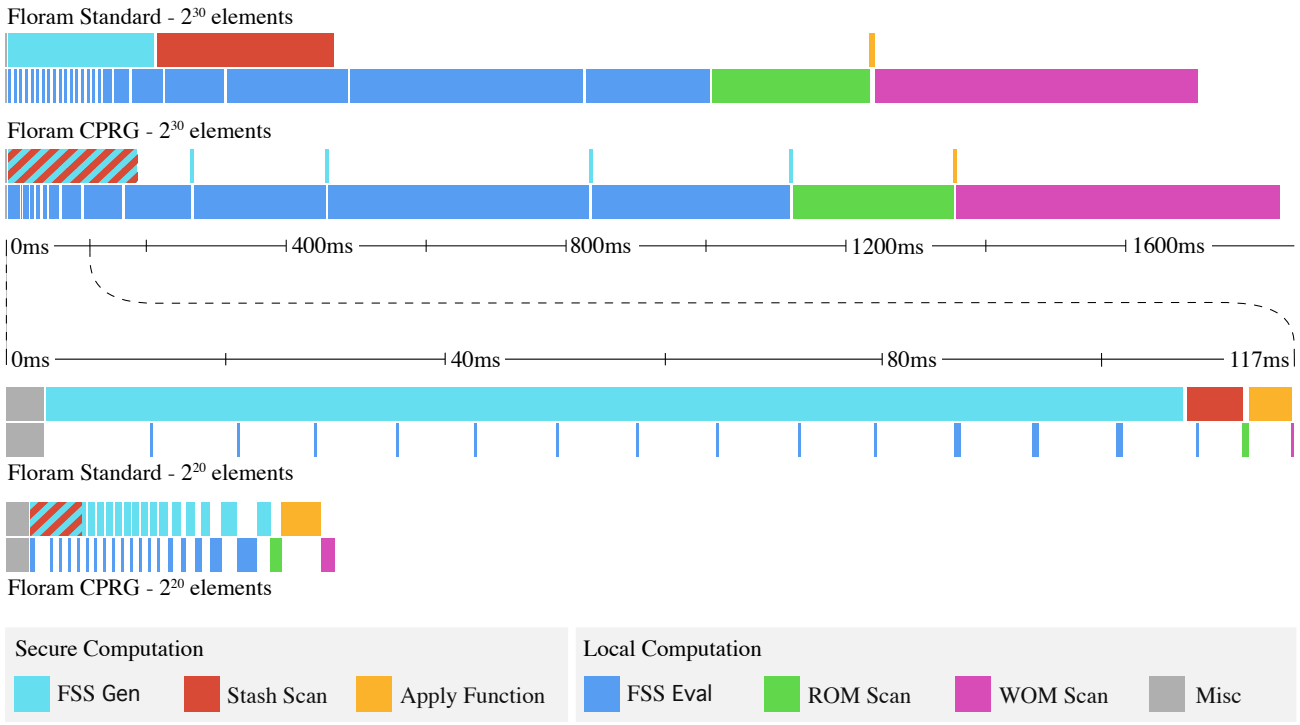


Figure 8: Scheduling diagram for an ORAM access operation. This illustrates the way in which we interleave the various operations of our ORAM. The x-axis represents time, in milliseconds, and the y-axes represent the divide between secure computation, and local computation. Times are averages from a number of samples that is greater than 100 and a multiple of the refresh period. Elements are 4 bytes. Cross-hatching indicates regions wherein two components are scheduled to run simultaneously, and may preempt one another. The misc category includes time spent allocating and copying memory, managing threads, and performing other local setup tasks.

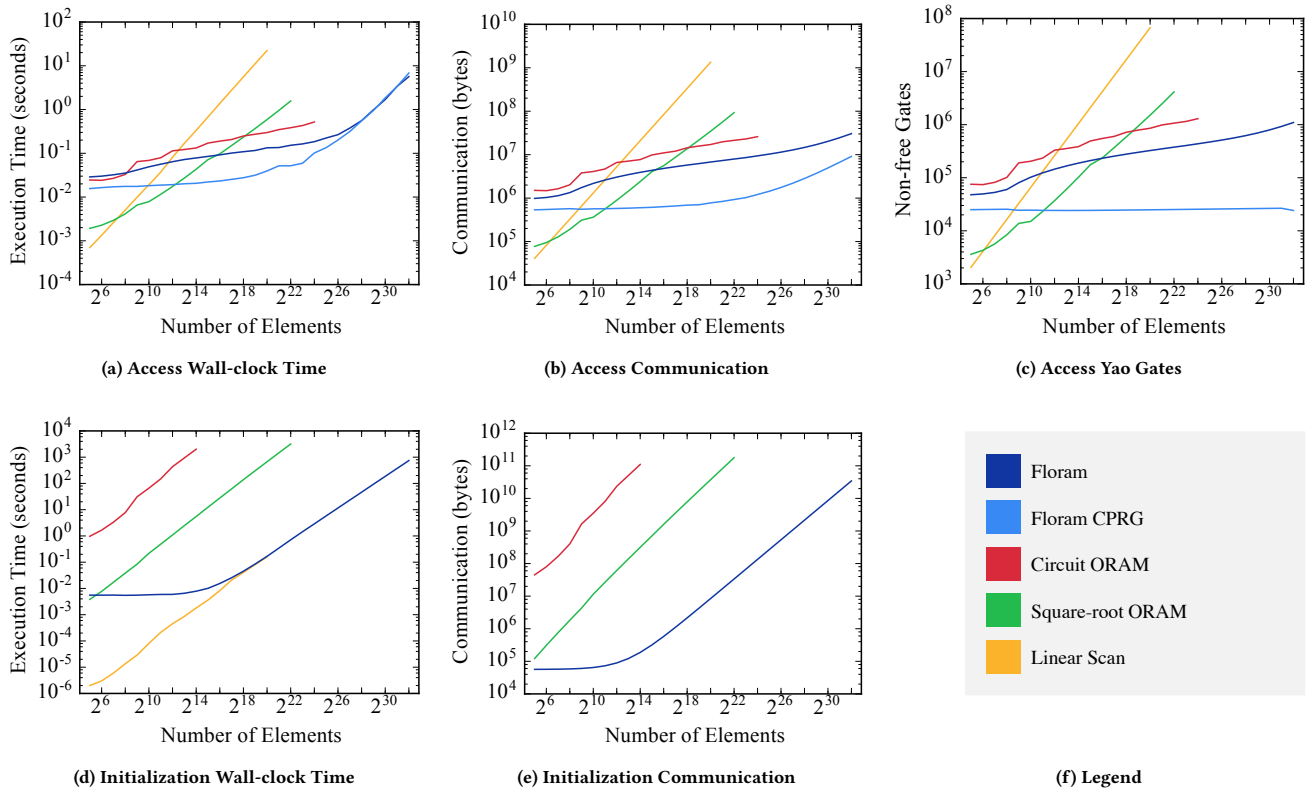
bit block size, this being the block size of AES-128, our chosen PRG function. For ORAMs with element sizes smaller than 128 bits, we pack multiple elements into a single block and linearly scan them. For ORAMS with element sizes greater than 128 bits, we perform an additional expansion and correction stage after the last layer of the FSS in order to enlarge the blocks to the correct length.

Our benchmarks were performed under Ubuntu 16.04 with Linux kernel 4.4.0 64-bit, running on a pair of identical Amazon EC2 `R4.4xlarge` instances. All code was compiled using `gcc` version 5.4.0, with the `-O3` flag enabled, OpenMP was used to manage multithreading and SIMD operations, and local AES computations were implemented using Intel's AES-NI instructions. Each machine had 122GB of DDR4 memory and eight physical cores partitioned from an Intel Xeon E5-2686 v4 CPU clocked at 2.3 GHz, each core being capable of executing two threads. We measured the bandwidth between our two instances to be roughly four gigabits per second. In order to ensure that the secure computation would be bandwidth-bound, as we would expect it to be in real-world conditions, we artificially restricted the bandwidth to 500 megabits per second, using the linux tool `tc`.

Multithreading. Our two Floram implementations make extensive use of multithreading for their local components, but we have not attempted to multithread their secure components, nor have we multithreaded the other ORAMs against which we make comparisons. Multithreading a secure computation does not reduce the total communication between parties, and thus in bandwidth-bound environments provides no advantage. Neither Square-root nor Circuit ORAM performs significant local computation, and so they cannot benefit significantly from local parallelism.

7.1 Full ORAM Microbenchmarks

Full Access. We performed single-access microbenchmarks for Floram, as well as Floram with the CPRG optimization discussed in Section 5. For the purpose of comparison, we also performed benchmarks for the Square-root ORAM of Zahur *et al.* [56], Circuit ORAM [43], and linear scan. For all ORAMs, we used an element sizes of 4 bytes. For linear scan, we varied the number of ORAM elements between 2^5 and 2^{20}, and for Square-root ORAM, between 2^5 and 2^{22}. In both cases, this is far past the range in which those schemes are competitive. For Circuit ORAM, we performed benchmarks with up to 2^{24} 4-byte elements, corresponding to 64 MiB of data; beyond this the ORAM's physical size was so large that it could not be instantiated on our machine. We benchmarked Floram with sizes up to 2^{32} 4-byte elements, corresponding to 16 GiB of data; these were the largest instances that our machine could handle. We recorded the wall-clock times for both parties, the number of



(a) Access Wall-clock Time (b) Access Communication (c) Access Yao Gates

(d) Initialization Wall-clock Time (e) Initialization Communication (f) Legend

Figure 9: Microbenchmark Results. Access figures are averages from at least 100 samples; for refreshing ORAMs, the sample count was a multiple of the refresh period. Initialization figures are averages from 30 samples. For all benchmarks, elements were 4 bytes in size.

bytes transmitted, and the number of non-free Yao gates executed. Our results are reported in Figures 9a, 9b, and 9c, respectively.

As we expected, the wall-clock time of our scheme exhibits a piecewise behavior. Up to roughly 2^{25} 4-byte elements, secure computation (specifically, the FSS Gen algorithm) dominates the total access time, and thus the time grows with $O(\log n)$—noticeably more slowly than any other ORAM. In this region, as expected, the CPRG optimization leads to a significant concrete performance gain, amounting to roughly a four-fold improvement. Beyond 2^{25} elements, local computation becomes the dominant factor, and thus the wall-clock time grows with $O(n)$ and the standard FSS scheme becomes more efficient. We estimate that the break-even point with Circuit ORAM lies at 2^{30} elements.

Initialization. We also performed initialization benchmarks. That is, beginning with an array of data, we evaluated each construction's native mechanism for importing that data into a fresh ORAM instance. As before, we varied the number of elements for linear scan between 2^5 and 2^{20}, and for Square-root ORAM between 2^5 and 2^{22}. Circuit ORAM has the slowest initialization process by several orders of magnitude, and so we benchmarked only up to 2^{14} elements, after which continuing was impractical. Both variants of Floram share the same initialization procedure, and we tested instances up to the largest size that our machines supported: 2^{32} 4-byte elements, or 16 GiB of data in total. Results for wall-clock time and total communication are reported in Figures 9d and 9e respectively; gate counts are not reported, as our ORAM requires no gates to initialize.

As we expected, our ORAM has a clear asymptotic advantage over other schemes in terms of initialization. Moreover, at 2^{22} elements, it has a 4500-fold concrete performance advantage over Square-root ORAM, the fastest previously known construction in this respect. In fact, in the context of garbled circuits, our construction even initializes somewhat faster than a linear scan, which requires only a simple `memcpy` by each party. Thus, so long as a single access in our scheme is faster than a single linear scan, the efficiency break-even point between the two is exactly *one* access. This is far better than other schemes, which require $\Omega(\log n)$ accesses in order to reach their break-even points.

Thread-restricted Microbenchmarks. Although our ORAM is bound by secure computation at small sizes, for very large instances, the local component becomes the dominant factor. Here we analyze its performance when a varying number of threads are used, in order to assess the performance of our algorithms in contexts where a high level of parallelism may not be available. We collected samples for each combination of ORAMs of 2^{10}, 2^{15}, 2^{20}, 2^{25}, and 2^{30} 4-byte elements, and 1, 2, 4, 8, and 16 threads. The results are plotted in Figure 10.

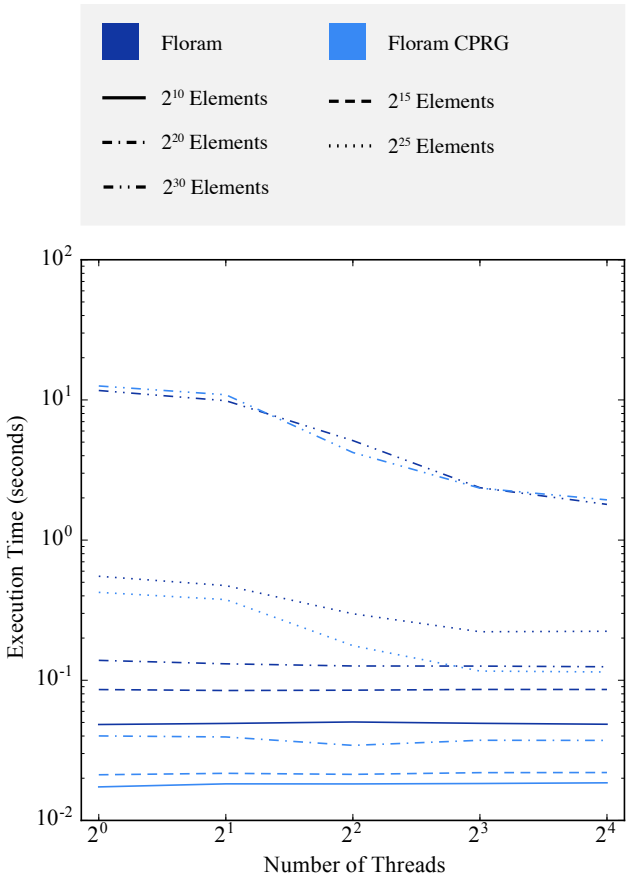


Figure 10: Thread-limited Access Wall-clock Time. Sample counts are multiples of the refresh period. Elements are 4 bytes.

At small ORAM sizes, where the entire computation might fit into the CPU cache, it is unsurprisingly the case that additional threads decrease performance. It is not until the linear component of our ORAM's complexity becomes dominant that parallelism makes a significant difference. Note that at 2^{25} elements and greater, the execution time decreases nearly linearly with threadcount, for threadcounts of eight and fewer. As our benchmark machines have only eight physical CPU cores, using more than eight threads offers little to no advantage.

7.2 Applications

In order to assess the performance of our ORAM construction in realistic scenarios, we implemented two secure applications, and benchmarked them with each of the ORAMs considered previously.

Binary Search. In order to highlight the ways in which the novel properties of our ORAM differentiate it from previous ORAM constructions, we begin with a simple binary search benchmark. The use of ORAM for performing binary searches was first considered by Gordon *et al.* [25], who reported that searching a database of 2^{20} 64-byte elements required roughly 1000 seconds.[6] Our ORAM benchmark procedure is derived from that used by Square-root ORAM [56]: first, the data is loaded from secure computation into an ORAM, and then a number of searches are performed (each requiring $\log_2 n$ semantic accesses to complete). In this context, linear scan has a special advantage: because it touches each element in the memory, it requires only a single semantic access to perform a search. As a consequence of this property, ORAM has thus far yielded little improvement over the trivial solution for the problem of searching.

n	s	Linear	Circuit	Square-root	Floram	CPRG
2^{15}	1	2.80	5192.4	12.87	0.79	**0.37**
	2^5	89.75	5284.2	37.24	23.73	**11.15**
	2^{10}	2872.1	8126.8	1210.0	758.89	**358.0**
2^{20}	1	89.52	–	690.99	2.04	**0.99**
	2^5	2864.5	–	800.23	56.94	**21.94**
	2^{10}	91,663.	–	12,736.	1826.5	**697.65**
2^{25}	1	2864.5	–	–	14.37	**11.55**

Table 2: Binary Search Benchmark Results. We measured the wall-clock time required for s searches through n 16-byte data elements, including initialization. Figures are averages in seconds from 30 samples for databases of 2^{15} elements, or 3 samples for larger databases. Linear scan figures are estimated from results in Section 7.1.

We executed instances of this benchmark upon databases of 2^{15} and 2^{20} 16-byte elements, with 1, 2^5, and 2^{10} searches being performed. In addition, we benchmarked single searches of databases of 2^{25} elements under Floram (due to exhaustion of memory, it was not possible to instantiate Square-root or Circuit ORAMs of this size). We do not include in our benchmark the cost of sorting the data, which is unnecessary for the linear scan solution. Sorting can be performed with a Batcher Mergesort [3] in $O(n \log^2 n)$, with practical costs being lower than the that of instantiating any of the tested ORAMs, other than Floram. Results are reported in Table 2.

Floram has the fastest access and initialization procedures at these sizes, and so, not surprisingly, it is the fastest among the ORAMs regardless of the number of searches performed. What is surprising, however, is that it is significantly faster than linear scan, *even when only a single search is performed.* To our knowledge, such a thing is not possible under any other ORAM scheme, at any data size. Our scheme achieves this due to the fact that, considering initialization and a single access, only two full scans of XOR shares are required, whereas in the context of Yao's Garbled Circuits a linear scan requires iterating over wire labels that are at least eighty times larger than the equivalent secret-shared representation.

Stable Matching. Many previous research efforts have sought to optimize the secure evaluation of the Gale-Shapley algorithm for stable matching. Recently, Doerner *et al.* [16] developed algorithmic

[6] Though we show significant improvement upon this number, our construction is not directly comparable to theirs, due to differences in the underlying protocol and benchmarking hardware.

	Square-root	Floram CPRG
Wall-clock Time (Hours)	28.98	**15.78**
Billions of Non-free Gates	226.87	**143.29**

Table 3: Roth-Peranson Benchmark Results. Our wall-clock time result for Square-root ORAM differs from that presented by Doerner *et al.* [16]; this is due to differences in benchmarking environments used.

improvements which yielded a significant increase in asymptotic and concrete performance, allowing them to execute a secure stable matching using the related Roth-Peranson algorithm on the scale of the stable matching performed annually by the National Resident Matching Program (NRMP) to match graduating doctors to medical residencies in the United States. This algorithm requires $O(nr)$ ORAM accesses in n, the number of doctors, and r, the number of hospitals for which the doctors are allowed to submit rankings, to a comparatively small ORAM of size $O(m)$ in m, the number of hospitals (in practice, around 5000 for NRMP-scale matchings). Nonetheless, in terms of gates, the NRMP matching is one of the largest secure computations ever reported. In other words, this is a benchmark for which Floram's initialization advantage matters very little. The parameters of the benchmark were derived by Doerner *et al.* from the 2016 NRMP Statistical Report; specifically: 35,476 residents submitting up to 15 rankings each, and 4836 hospitals submitting up to 120 rankings each, and having at most 12 open positions. Individual preferences were generated at random. We collected one sample each for Square-root ORAM and Floram CPRG, and, following Doerner *et al.*, we did not collect any data for Circuit ORAM or linear scan, which would not be competitive. The results are shown in Table 3, and demonstrate a factor of 1.83 improvement over prior work for a very small ORAM used in a real application.

7.3 Notes on Scalability

The title of this document is "Scaling ORAM for Secure Computation", and so it is fitting that we should comment upon the limits of scaling, and how well we believe our implementation has fared relative to the theoretical possibilities. At 2^{32} four-byte elements, we measured our scheme to require 6.3 seconds to complete an access, on average. During this time, it reads the underlying memories of both the WOM and the ROM, and writes the WOM. In the course of the FSS Eval algorithm, it both reads and writes an amount of data equal to twice the size of the WOM or ROM memory. The stash is negligible in size by comparison. Thus, the amount of data transferred to and from memory inside each local machine is $2^{32} \cdot 4 \cdot 7$ bytes in total, or 120.3 gigabytes, at 152.8 gigabits per second. For comparison, a single DDR4-2400 memory controller has a maximum bandwidth of 153.6 gigabits per second. We do not know exactly how resources are apportioned among EC2 instances, but we do know that we are renting eight of the 18 physical cores in a single CPU, and that those 18 physical cores share four memory controllers. If partitioning were perfectly fair, we would expect our instance to have access to slightly less than two memory controllers' worth of bandwidth. Thus, we conjecture that we are within roughly a factor of two of the best possible performance on our test hardware. This is not bad, considering that the parallelization and scheduling of our implementation are not hand-tuned, and we have taken no pains to ensure proper affinity between CPU and memory.

At large sizes, local CPU and memory bandwidths are the definitive bottlenecks for our scheme. These are easily increased: in modern systems each additional processor has its own set of memory controllers. Furthermore, our algorithm is parallel in such a way that it can be run on a cluster with little performance loss: only $\log(n)$ synchronizations per access would be required, and each synchronization involves the transfer of a small, constant amount of data. We suggest that further scaling and performance improvement can be accomplished by the addition of computing hardware, which is typically cheap relative to the cost of additional bandwidth, as would be incurred were our scheme network bound.

ACKNOWLEDGMENT

The authors would like to thank Yuval Ishai for his suggestion to truncate the FSS tree during reads. The authors would also like to thank the authors of the Square-root ORAM paper [56], and especially Samee Zahur, for his insight and technical expertise. This research was supported by NSF Grants TWC-1664445 and TWC-1646671.

CODE AVAILABILITY

Complete reference implementations of the constructions described in this paper along with implementations of Square-root and Circuit ORAM sharing a common interface are available under the 3-clause BSD license from https://gitlab.com/neucrypt/floram.

REFERENCES

[1] 2001. Advanced Encryption Standard. (2001).
[2] Ittai Abraham, Christopher W. Fletcher, Kartik Nayak, Benny Pinkas, and Ling Ren. 2017. Asymptotically Tight Bounds for Composing ORAM with PIR. In *PKC*.
[3] Ken Batcher. 1968. Sorting Networks and Their Applications. In *Spring Joint Computer Conference*.
[4] Donald Beaver, Silvio Micali, and Phillip Rogaway. 1990. The Round Complexity of Secure Protocols. In *ACM STOC*.
[5] Mihir Bellare, Viet Tung Hoang, Sriram Keelveedhi, and Phillip Rogaway. 2013. Efficient Garbling from a Fixed-Key Blockcipher. In *IEEE S&P*.
[6] Marina Blanton, Aaron Steele, and Mehrdad Alisagari. 2013. Data-oblivious Graph Algorithms for Secure Computation and Outsourcing. In *ACM Asia CCS*.
[7] Dan Boneh, David Mazieres, and Raluca Ada Popa. 2011. Remote Oblivious Storage: Making Oblivious RAM practical. http://dspace.mit.edu/bitstream/handle/1721.1/62006/MIT-CSAIL-TR-2011-018.pdf. (2011).
[8] Joan Boyar and René Peralta. 2010. A New Combinational Logic Minimization Technique with Applications to Cryptology. In *Lecture Notes in Computer Science*.
[9] Joan Boyar and René Peralta. 2012. *A Small Depth-16 Circuit for the AES S-Box*.
[10] Elette Boyle, Kai-Min Chung, and Rafael Pass. 2016. Oblivious Parallel RAM and Applications. In *TCC*.
[11] Elette Boyle, Niv Gilboa, and Yuval Ishai. 2015. Function Secret Sharing. In *EUROCRYPT*.
[12] Elette Boyle, Niv Gilboa, and Yuval Ishai. 2016. Function Secret Sharing: Improvements and Extensions. In *ACM CCS*.
[13] Kai-Min Chung, Zhenming Liu, and Rafael Pass. 2013. Statistically-secure ORAM with $\tilde{O}(\log^2 n)$ Overhead. *arXiv preprint arXiv:1307.3699* (2013).
[14] Ivan Damgård, Sigurd Meldgaard, and Jesper Buus Nielsen. 2011. Perfectly Secure Oblivious RAM without Random Oracles. In *TCC*.
[15] Srinivas Devadas, Marten van Dijk, Christopher W. Fletcher, Ling Ren, Elaine Shi, and Daniel Wichs. 2016. Onion ORAM: A Constant Bandwidth Blowup Oblivious RAM. In *TCC*.
[16] Jack Doerner, David Evans, and abhi shelat. 2016. Secure Stable Matching at Scale. In *ACM CCS*.

[17] Niv Gilboa and Yuval Ishai. 2014. *Distributed Point Functions and Their Applications.*
[18] Oded Goldreich. 1987. Towards a theory of software protection and simulation by oblivious RAMs. In *ACM STOC.*
[19] Oded Goldreich. 2004. *Foundations of Cryptography: Volume 2, Basic Applications.* Cambridge University Press, New York, NY, USA.
[20] Oded Goldreich, Shafi Goldwasser, and Silvio Micali. 1986. How to Construct Random Functions. *J. ACM* 33, 4 (Aug. 1986).
[21] Oded Goldreich and Rafail Ostrovsky. 1996. Software Protection and Simulation on Oblivious RAMs. *Journal of the ACM* 43, 3 (1996).
[22] Michael T. Goodrich and Michael Mitzenmacher. 2011. Privacy-Preserving Access of Outsourced Data via Oblivious RAM Simulation. In *ICALP.*
[23] Michael T. Goodrich, Michael Mitzenmacher, Olga Ohrimenko, and Roberto Tamassia. 2011. Oblivious RAM Simulation with Efficient Worst-Case Access Overhead. In *ACM CCSW.*
[24] Michael T. Goodrich, Michael Mitzenmacher, Olga Ohrimenko, and Roberto Tamassia. 2012. Privacy-preserving group data access via stateless oblivious RAM simulation. In *ACM-SIAM SODA.*
[25] Dov Gordon, Jonathan Katz, Vladimir Kolesnikov, Fernando Krell, Tal Malkin, Mariana Raykova, and Yevgeniy Vahlis. 2012. Secure Two-party Computation in Sublinear (Amortized) Time. In *ACM CCS.*
[26] Yan Huang, David Evans, Jonathan Katz, and Lior Malka. 2011. Faster Secure Two-party Computation Using Garbled Circuits. In *USENIX Security Symposium.*
[27] Zahra Jafargholi and Daniel Wichs. 2016. *Adaptive Security of Yao's Garbled Circuits.*
[28] Marcel Keller and Peter Scholl. 2014. *Efficient, Oblivious Data Structures for MPC.*
[29] Vladimir Kolesnikov and Thomas Schneider. 2008. Improved Garbled Circuit: Free XOR Gates and Applications. In *ICALP.*
[30] Eyal Kushilevitz, Steve Lu, and Rafail Ostrovsky. 2012. On the (In)security of Hash-based Oblivious RAM and a New Balancing Scheme. In *ACM-SIAM SODA.*
[31] Yehuda Lindell and Benny Pinkas. 2009. A Proof of Security of Yao's Protocol for Two-Party Computation. *Journal of Cryptology* 22, 2 (2009).
[32] Steve Lu and Rafail Ostrovsky. 2013. *Distributed Oblivious RAM for Secure Two-Party Computation.*
[33] Valeria Nikolaenko, Udi Weinsberg, Stratis Ioannidis, Marc Joye, Dan Boneh, and Nina Taft. 2013. Privacy-Preserving Ridge Regression on Hundreds of Millions of Records. In *IEEE S&P.*
[34] Rafail Ostrovsky. 1990. Efficient computation on oblivious RAMs. In *ACM STOC.*
[35] Rafail Ostrovsky and Victor Shoup. 1997. Private Information Storage (Extended Abstract). In *ACM STOC.*
[36] Benny Pinkas and Tzachy Reinman. 2010. Oblivious RAM revisited. In *CRYPTO.*
[37] Benny Pinkas, Thomas Schneider, Nigel P. Smart, and Stephen C. Williams. 2009. Secure Two-Party Computation Is Practical. In *ASIACRYPT.*
[38] Adi Shamir. 1979. How to Share a Secret. *Commun. ACM* 22, 11 (Nov. 1979).
[39] Elaine Shi, T.-H. Hubert Chan, Emil Stefanov, and Mingfei Li. 2011. Oblivious RAM with $O((\log N)^3)$ Worst-Case Cost. In *ASIACRYPT.*
[40] Emil Stefanov, Marten Van Dijk, Elaine Shi, Christopher Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. 2013. Path ORAM: an Extremely Simple Oblivious RAM Protocol. In *ACM CCS.*
[41] Emil Stefanov, Marten van Dijk, Elaine Shi, Christopher Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. 2013. Path ORAM: an Extremely Simple Oblivious RAM Protocol. In *ACM CCS.*
[42] Abraham Waksman. 1968. A Permutation Network. *Journal of the ACM* 15, 1 (Jan. 1968).
[43] Xiao Wang, Hubert Chan, and Elaine Shi. 2015. Circuit ORAM: On Tightness of the Goldreich-Ostrovsky Lower Bound. In *ACM CCS.*
[44] Xiao Wang, Yan Huang, Hubert Chan, Abhi Shelat, and Elaine Shi. 2014. SCORAM: Oblivious RAM for Secure Computation. In *ACM CCS.*
[45] Xiao Wang, Yan Huang, Yongan Zhao, Haixu Tang, XiaoFeng Wang, and Diyue Bu. 2015. Efficient Genome-Wide, Privacy-Preserving Similar Patient Query Based on Private Edit Distance. In *ACM CCS.*
[46] Peter Williams and Radu Sion. 2008. Usable PIR. In *NDSS.*
[47] Peter Williams and Radu Sion. 2012. Round-Optimal Access Privacy on Outsourced Storage. In *ACM CCS.*
[48] Peter Williams, Radu Sion, and Bogdan Carbunar. 2008. Building castles out of mud: Practical access pattern privacy and correctness on untrusted storage. In *ACM CCS.*
[49] R. S. Winternitz. 1984. A Secure One-Way Hash Function Built from DES. In *IEEE S&P.*
[50] Johannes Wolkerstorfer, Elisabeth Oswald, and Mario Lamberger. 2002. An ASIC Implementation of the AES SBoxes. In *RSA Conference on Topics in Cryptology.*
[51] David Woodruff and Sergey Yekhanin. 2005. A Geometric Approach to Information-Theoretic Private Information Retrieval. In *Proceedings of the 20th Annual IEEE Conference on Computational Complexity.*
[52] Andrew Chi-Chih Yao. 1982. Protocols for Secure Computations. In *IEEE FOCS.*
[53] Andrew Chi-Chih Yao. 1986. How to Generate and Exchange Secrets (Extended Abstract). In *IEEE FOCS.*
[54] Samee Zahur and David Evans. 2015. Obliv-C: A Lightweight Compiler for Data-Oblivious Computation. Cryptology ePrint Archive, Report 2015/1153. http://oblivc.org. (2015).
[55] Samee Zahur, Mike Rosulek, and David Evans. 2015. Two Halves Make a Whole: Reducing Data Transfer in Garbled Circuits Using Half Gates. In *EUROCRYPT.*
[56] Samee Zahur, Xiao Wang, Mariana Raykova, Adrià Gascón, Jack Doerner, David Evans, and Jonathan Katz. 2016. Revisiting Square Root ORAM: Efficient Random Access in Multi-Party Computation. In *IEEE S&P.*

A DEFINITIONS

A.1 Security

We first recall the semi-honest security model in which we claim our scheme is secure.

Definition A.1 (Semi-honest Security [19, 31]). Let $\mathcal{F} = (\mathcal{F}_a, \mathcal{F}_b)$ be a probabilistic polynomial time functionality, and let π be a two party protocol for computing $\mathcal{F}$ such that party A supplies input x_a and receives output $\mathcal{F}_a(x_a, x_b)$, while party B supplies input x_b and receives output $\mathcal{F}_b(x_a, x_b)$, with $|x_a| = |x_b|$. π is considered secure in the presence of static semi-honest adversaries if there exist probabilistic polynomial-time simulators Sim_a and Sim_b such that

$$\left\{\left(\mathsf{Sim}_p\left(1^\lambda, x_p, \mathcal{F}_p\left(x_a, x_b\right)\right), \mathcal{F}\left(1^\lambda, x_a, x_b\right)\right)\right\}_{\lambda\in\mathbb{N}, x_a, x_b\in\{0,1\}^*}$$
$$\stackrel{c}{\equiv} \left\{\left(\mathsf{View}_p^\pi\left(1^\lambda, x_a, x_b\right), \mathsf{Output}^\pi\left(1^\lambda, x_a, x_b\right)\right)\right\}_{\lambda\in\mathbb{N}, x_a, x_b\in\{0,1\}^*}$$

for $p \in \{a, b\}$ where $\mathsf{View}_p^\pi(x_a, x_b) = (x_p, r_p, m_p^1, \ldots, m_p^t)$ is party p's view of the computation, with r_p denoting party p's internal random tape and m_p^j denoting the j^{th} message that party p received; and where $\mathsf{Output}^\pi(1^\lambda, x_a, x_b)$ denotes the union of the outputs of all parties; and where $\stackrel{c}{\equiv}$ denotes computational indistinguishability with security parameter λ. That is, a protocol π is secure in the semi-honest setting if the full view of a party can be simulated by a probabilistic polynomial time algorithm given only a record of that party's input and output. Note we assume that all protocols and functionalities have access to the security parameter λ, and that computational indistinguishability is relative to this parameter. In proofs, we omit λ from our notation.

A.2 Distributed ORAM

We deviate from the standard formulation of ORAM in order to align the security model of our scheme with the security model of multiparty computation (Definition A.1). We assume that the ORAM's storage, like the protocols that implement its access an initialization methods, is split among multiple parties, and we guarantee security only against the corruption of some subset of those parties. In contrast, the standard ORAM definition [21] considers a context wherein there exists a single trusted CPU and a single untrusted memory, and assumes that an adversary has a full view of all memory accesses, but no insight into the CPU. Our variant of the ORAM definition is known as Distributed ORAM; it was originally proposed by Lu and Ostrovsky [32], and our definitions expound theirs.

Definition A.2 (Random Access Memory). For every $n, m \in \mathbb{N}$, a random access memory $\mathsf{RAM}_{n,m}$ is a functionality that associates an m-bit value with each unique integer index in $[1, n]$ and can recall this value when queried with the index. Indexes are by default

associated with values of 0^m. A $\mathsf{RAM}_{n,m}$ receives instructions of the form (o, i, v), where $o \in \{\text{read}, \text{write}\}$ is an operation specifier, $i \in [1, n]$ is an index, and $v \in \{0,1\}^m$ is a value. Additionally, a $\mathsf{RAM}_{n,m}$ may receive an initialization instruction of the form (init, V), where $V \in \{0,1\}^{n \times m}$ is an array of values. Upon receiving an instruction (o, i, v), a $\mathsf{RAM}_{n,m}$ must behave as follows:

(1) if $o = \text{read}$, then $\mathsf{RAM}_{n,m}$ immediately recalls and returns the value associated with index i, and ignores v.
(2) if $o = \text{write}$, then $\mathsf{RAM}_{n,m}$ remembers value v and associates it with index i, forgetting any previous associations that index i may have had, and returns nothing.

Upon Receiving an initialization instruction (init, V), $\mathsf{RAM}_{n,m}$ immediately forgets all associations it has previously made, and associates the values in V with their corresponding indices.

Note. Any structure that implements the write operation can implement the init operation as a sequence of writes. However, our construction has a dedicated initialization function which requires its own analysis. Therefore, we include init in our definition.

Definition A.3 (Distributed Random Access Memory). For every $n, m \in \mathbb{N}$, a Distributed Random Access Memory $\mathsf{DRAM}_{n,m}$ is a protocol evaluated among two parties which correctly implements the $\mathsf{RAM}_{n,m}$ functionality. An implementation of $\mathsf{DRAM}_{n,m}$ may require that each party $p \in \{a, b\}$ implements a private, local instance M_p of the $\mathsf{RAM}_{\text{poly}(n),\text{poly}(m)}$ functionality. For each instruction it receives, a $\mathsf{DRAM}_{n,m}$ may issue to each of its local memories a number of instructions bounded by $\text{poly}(n)$. We assume that instructions issued to and replies received from the M_p of a non-corrupt party p are observable only by p. A $\mathsf{DRAM}_{n,m}$ may additionally have access to a random tape.

Note. For simplicity, we define DRAM for two parties and observe that it can be extended to many parties.

Definition A.4 (Access Patterns and Epochs). For any memory M that implements $\mathsf{RAM}_{n,m}$, an access pattern is a sequence $\{x^j\}_{j \in [1,\ell]}$ of length ℓ, such that x^j corresponds to the j^{th} instruction received by M. An epoch is an access pattern $X = \{x^j\}_{j \in [1,\ell]}$ such that x^1 is an initialization instruction (init, V) and all subsequent instructions are either read or write instructions. We use $\Xi_{n,m,\lambda}$ to denote the set of all valid epochs for a $\mathsf{RAM}_{n,m}$ with lengths in $O(\text{poly}(\lambda))$. A sequence of epochs is constructed by deriving the initialization vector for epoch j from the final state in epoch $j-1$. Thus, a sequence of epochs has only one initialization vector. We use $\mathcal{X}$ to represent a sequence of epochs, and $\Xi^*_{n,m,\lambda}$ to denote the set of all such sequences with total lengths in $O(\text{poly}(\lambda))$.

Note. It is necessary to introduce the concept of epochs due to the existence of an initialization instruction. While we expect an ORAM to hide indices accessed and whether accesses are reads or writes, we cannot expect it to hide which instructions are initialization instructions. Consequently, in subsequent definitions, we will reason over sequences of epochs, each of which has exactly one initialization. While most ORAM schemes that require refreshing use fixed epoch lengths, this is seldom necessary, and in fact Floram can vary its refresh period to achieve greater practical efficiency. Consequently, we allow for arbitrary epoch lengths in our definitions and proofs.

Definition A.5 (Distributed Oblivious Random Access Memory). For every $n, m, \lambda \in \mathbb{N}$, a suite of multi-party protocols D is a $\mathsf{DORAM}_{n,m,\lambda}$ if it implements $\mathsf{DRAM}_{n,m}$ and there exists a simulator Sim^D_p for $p \in \{a, b\}$ such that for security parameter λ:

$$\left\{\mathsf{Sim}^D_p\left(1^\lambda, \left\{1^{|X|}\right\}_{X \in \mathcal{X}}, V_p\right)\right\}_{\mathcal{X} \in \Xi^*_{n,m,\lambda}} \stackrel{c}{\equiv} \left\{\mathsf{View}^D_p(1^\lambda, \mathcal{X})\right\}_{\mathcal{X} \in \Xi^*_{n,m,\lambda}}$$

That is, the view of party p over a sequence of epochs can be simulated given only the lengths of those epochs and p's share of the initialization vector V associated with the first epoch. Note that V_p is an array of $n \times m$ bits.

Discussion. Although ORAM is sometimes taken as an acronym for Oblivious Random Access Memory, Goldreich and Ostrovsky use it to stand for Oblivious Random Access *Machine*, and their model includes a trusted CPU capable of arbitrary computation in a data-oblivious fashion. Although our definitions do not explicitly call upon universal computation, they nonetheless imply a similar conclusion. Specifically, our definitions, in combination with MPC protocols, imply the ability to securely compute circuits with "memory gates"; that is, gates capable of storing and retrieving data in a black-box fashion while maintaining data-obliviousness. From such circuits, it is possible to construct CPUs that can execute secure instructions in a familiar way.

B PROOFS OF SECURITY

In this section, we prove that the standard Floram construction is a secure DORAM under Definition A.5. To do this, we prove the protocol security of our initialization and access methods under Definition A.1, and then compose these proofs to show security over the course of an epoch. Given security over an epoch, a standard hybrid proof can show security over a sequence of epochs. We do not consider semi-private access, data export, or any other nonstandard capabilities of our construction, nor do we consider any of the optimizations we have presented throughout this work. Nonetheless, we have no reason to suspect that they are insecure.

Mapping definitions to concrete schemes. We have defined DORAM to implement three different methods: read, write, and init, but Floram only actually implements init and a generic access method, which applies an arbitrary function f to the target element. If f_{read} and f_{write} are combined into a single circuit or constructed in such a way that they can be simulated by a single simulator, then accesses that perform reads will be indistinguishable from accesses that perform writes, as required.

B.1 Proof of Security for Access

Notation and Real-world View. Before we present our proof, we specify a convenient notation describing the same access algorithm given in Section 4. We refer to the functionality implemented by the algorithm as $\mathcal{F}_A$, and the protocol as π_A. Party p's share of the output of the functionality $\mathcal{F}_A$ is $\mathcal{F}_{Ap}$. Party p's input to the algorithm is denoted by Input^A_p, and p's output of a protocol execution using that input is denoted $\mathsf{Output}^{\pi_A}_p(\mathsf{Input}^A)$, while a complete transcript of the protocol execution for party p is denoted

by $\text{View}_p^{\pi_A}(\text{Input}^A)$. The access protocol can be decomposed into a four step process, $(C_1, \mathcal{L}_1, C_2, \mathcal{L}_2)$, where C_1 and C_2 are circuits evaluated by some MPC protocol (we use Yao's Garbled Circuits), and $\mathcal{L}_1$ and $\mathcal{L}_2$ are party-local computations. These circuits receive some of their input values as secret-shares, and party p's secret share of value x is denoted x_p. We omit special notation for share-creation and reconstruction operations, leaving them implicit. We use $x \xleftarrow{\$} X$ to signify the uniform random choice of element x from the set X, $\overset{c}{\equiv}$ to signify computational indistinguishability, and $\overset{s}{\equiv}$ to signify statistical indistinguishability.

The first circuit, C_1, implements the FSS Gen algorithm. C_1 receives shares of the target index i, as well as shares of a uniformly randomly chosen value β, such that $\beta_p \in \{0,1\}^\lambda$. To Alice, C_1 returns the FSS key k_a^{FSS}, and to Bob, k_b^{FSS} (these keys may be thought of as a sharing of the joint FSS key, k^{FSS}). Formally:

$$\begin{aligned}\text{Input}_p^{C_1} &= \left(i_p, \beta_p\right)\\ \text{Output}_p^{\pi_{C_1}}\left(\text{Input}^{C_1}\right) &= \left(k_p^{\text{FSS}}\right)\end{aligned}$$

Subsequent to C_1, each party p executes a local computation, $\mathcal{L}_1$, which takes as input k_p^{FSS} and also some local state R (the ROM memory), and produces v_p.

The second circuit, C_2, implements the stash scan, function application, and FSS leaf adjustment procedures. This circuit receives shares from both parties of i, β, and the stash state Stash. From Alice, it receives as input $v_a, k_a^{\text{FSS}}, k_a^{\text{PRF}}$, and from Bob, $v_b, k_b^{\text{FSS}}, k_b^{\text{PRF}}$. A description of f, the function to be applied, is baked into the circuit C_2. As output, the circuit returns v^Δ to both parties. In addition, it returns shares of the updated stash state Stash$'$. f may receive some auxiliary input v^f as shares, and may produce some auxiliary output y^f as shares. Formally:

$$\begin{aligned}\text{Input}_p^{C_2} &= \left(i_p, v_p, v_p^f, \beta_p, \text{Stash}_p, k_p^{\text{FSS}}, k_p^{\text{PRF}}\right)\\ \text{Output}_p^{\pi_{C_2}}\left(\text{Input}^{C_2}\right) &= \left(v^\Delta, y_p^f, \text{Stash}'_p\right)\end{aligned}$$

Subsequent to C_2, each party p executes a local computation, $\mathcal{L}_2$, which takes as input k_p^{FSS}, v^Δ, and some local state, W_p (a share of the WOM memory), and returns some updated local state, W'_p.

The sequence $(C_1, \mathcal{L}_1, C_2, \mathcal{L}_2)$ composes the access protocol, as illustrated in Figure 11. Party p's view of an access is equal to the union of p's internal random tape r_p, its inputs, outputs, and the messages it receives. Using $\text{Msgs}_p^{\pi_C}(\text{Input}^C)$ to denote the messages received during evaluation of circuit C via protocol π_C (excepting the input and output), this give us:

$$\begin{aligned}\text{Input}_p^A &= \begin{pmatrix} i_p, R, f, v_p^f,\\ \text{Stash}_p, k_p^{\text{PRF}}, W_p\end{pmatrix}\\ \text{Input}^A &= \text{Input}_a^A \cup \text{Input}_b^A\\ \text{Output}_p^{\pi_A}\left(\text{Input}^A\right) &= \left(y_p^f, \text{Stash}'_p, W'_p\right)\end{aligned}$$

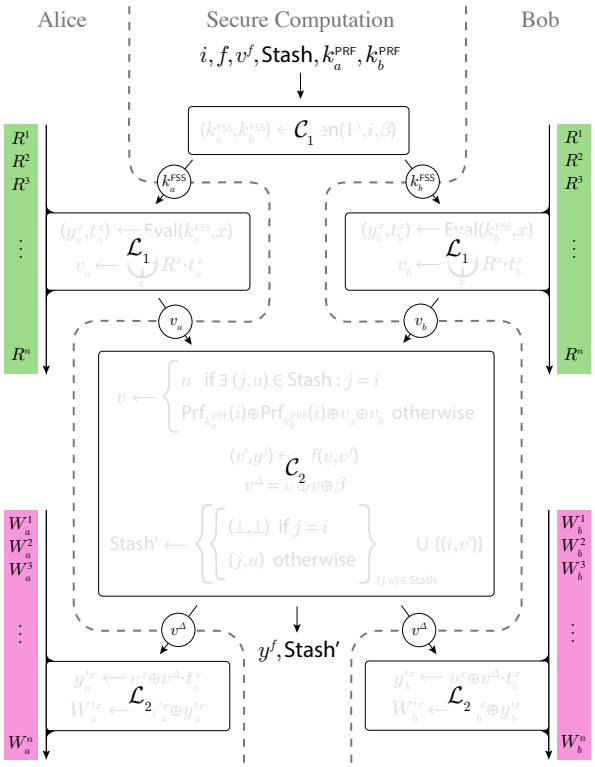


Figure 11: Diagram of the Floram Access method, illustrating the correspondence between the view described here and the algorithm presented in Section 4.

$$\begin{aligned}\text{View}_p^{\pi_A}\left(\text{Input}^A\right) &= \begin{pmatrix} r_p, \text{View}_p^{\pi_{C_1}}\left(\text{Input}^{C_1}\right), R,\\ \text{View}_p^{\pi_{C_2}}\left(\text{Input}^{C_2}\right), W_p, W'_p\end{pmatrix}\\ &= \begin{pmatrix} r_p, \text{Input}_p^A, \beta_p, \text{Msgs}_p^{\pi_{C_1}}\left(\text{Input}^{C_1}\right),\\ k_p^{\text{FSS}}, \text{Msgs}_p^{\pi_{C_2}}\left(\text{Input}^{C_2}\right), v^\Delta, y_p^f,\\ \text{Stash}'_p, W'_p\end{pmatrix}\end{aligned}$$

Valid Inputs. An input to the access protocol, Input^A, is said to be *valid* if and only if $i \in [1,n]$, $|R| = |W| = n$, k_a^{PRF} and k_b^{PRF} are the two keys for the PRFs that were used to mask R, and the stash contains only those elements which differ between R and W when R is unmasked:

$$(j,u) \in \text{Stash} \iff \left(u = W^j\right) \wedge \left(W^j \neq \text{Prf}_{k_a^{\text{PRF}}} \oplus \text{Prf}_{k_b^{\text{PRF}}} \oplus R^j\right)$$

We denote the set of all valid inputs for A for an ORAM of n elements of size m as $\text{Dom}_{n,m}^A$.

Lemma B.1 (Correctness for π_A). *If* (Gen, Eval) *is a secure FSS scheme for DPFs, π_{C_1} and π_{C_2} are secure multiparty computation protocols for C_1 and C_2 respectively, and* Prf *is a Pseudo-random Function Family, then:*

$$\begin{aligned}&\left\{\mathcal{F}_A\left(\text{Input}^A\right)\right\}_{\text{Input}^A \in \text{Dom}_{n,m}^A}\\ &\overset{s}{\equiv} \left\{\text{Output}^{\pi_A}\left(\text{Input}^A\right)\right\}_{\text{Input}^A \in \text{Dom}_{n,m}^A}\end{aligned} \tag{1}$$

function $\mathcal{F}_A\left(\mathsf{Input}^A\right)$:

// Parse Input^A as $\left(\begin{array}{l} i, R, f, v^f, \\ \mathsf{Stash}, k_a^{\mathsf{PRF}}, k_b^{\mathsf{PRF}}, W \end{array}\right)$

$$v \leftarrow \begin{cases} u & \text{if } \exists\,(j,u) \in \mathsf{Stash} : j = i \\ \mathsf{Prf}_{k_a^{\mathsf{PRF}}}(i) \oplus \mathsf{Prf}_{k_b^{\mathsf{PRF}}}(i) \oplus R^i & \text{otherwise} \end{cases}$$

$$\left(v', y^f\right) \leftarrow f\left(v, v^f\right)$$

$$W' \leftarrow \left\{ \begin{cases} v' & \text{if } j = i \\ W^j & \text{otherwise} \end{cases} \right\}_{j \in [1,n]}$$

$$\mathsf{Stash}' \leftarrow \left\{ \begin{cases} (\perp, \perp) & \text{if } j = i \\ (j,u) & \text{otherwise} \end{cases} \right\}_{(j,u)\in\mathsf{Stash}} \cup \left\{(i, v')\right\}$$

return $\left(y^f, \mathsf{Stash}', W'\right)$

function $\mathcal{F}_{Ap}\left(\mathsf{Input}^A\right)$:

$\left(y^f, \mathsf{Stash}', W'\right) \leftarrow \mathcal{F}_A\left(\mathsf{Input}^A\right)$

return $\left(y_p^f, \mathsf{Stash}'_p, W'_p\right)$ // generate secret shares

Figure 12: Pseudocode for the ideal functionality of the access protocol π_A.

Proof. We specify the ideal functionality for π_A in Figure 12. First we consider the circuits C_1 and C_2, implemented by MPC protocols π_{C_1} and π_{C_2} respectively. If these protocols are secure under Definition A.1 then $\mathsf{Output}^{\pi_{C_1}}\left(\mathsf{Input}^{C_1}\right) \overset{c}{\equiv} C_1\left(\mathsf{Input}^{C_1}\right)$ and $\mathsf{Output}^{\pi_{C_2}}\left(\mathsf{Input}^{C_2}\right) \overset{c}{\equiv} C_2\left(\mathsf{Input}^{C_1}\right)$. Because we consider only the outputs and not the full views, the lengths of the elements in these ensembles remain fixed, even as the security parameter increases. If they are computationally indistinguishable then it follows that as the security parameter increases they must also become statistically close (i.e. correct with very high probability).

The remainder of the correctness proof follows by inspection. As shown in Figures 5 and 11, the functionality of C_1 is the FSS Gen algorithm. Because (Gen, Eval) is a correct FSS scheme per Definition 2.3, the output of the Eval function for party p will be p's share of a pair of point functions y and t with values β and 1 respectively at index i. Algorithm $\mathcal{L}_1$ implements the dot product of t with R, and so v_a and v_b are shares of R^i.

The functionality of C_2 is given in the appropriate sections of Figures 5 and 11. Input^A is assumed to be valid, which implies that R was twice-masked by Prf, and it follows that either

$$W^i = \mathsf{Prf}_{k_a^{\mathsf{PRF}}}(i) \oplus \mathsf{Prf}_{k_b^{\mathsf{PRF}}}(i) \oplus R^i$$

or $(i, W^i) \in \mathsf{Stash}$. Either way, we have $v = W^i$, the correct value. Note that the stash read, function application, and stash write steps are specified identically between $\mathcal{F}_A$ and C_2, and consequently

$$\begin{aligned} &\left\{\mathcal{F}_A\left(\mathsf{Input}^A\right) : \left(y^f, \mathsf{Stash}'\right)\right\}_{\mathsf{Input}^A \in \mathsf{Dom}^A_{n,m}} \\ &\overset{s}{\equiv} \left\{\mathsf{Output}^{\pi_A}\left(\mathsf{Input}^A\right) : \left(y^f, \mathsf{Stash}'\right)\right\}_{\mathsf{Input}^A \in \mathsf{Dom}^A_{n,m}} \end{aligned} \tag{2}$$

Here we reason about only two of the three elements in Output^{π_A}; we must still reason about W'. Recall that $v = W^i$, and that each party has shares of two point functions: y and t with values β and 1 at index i respectively. C_2 calculates $v^\Delta = W^i \oplus v' \oplus \beta$. By inspection of $\mathcal{L}_2$, we see that both parties XOR v^Δ into their shares of the point function y, conditioned element-wise on t, yielding shares of a new point function y' such that for $j \in [1, n]$:

$$y'^j = \begin{cases} \beta \oplus v^\Delta = \beta \oplus \beta \oplus v' \oplus W^i = v' \oplus W^i & \text{if } j = i \\ 0 \oplus v^\Delta \oplus v^\Delta = 0 & \text{otherwise} \end{cases}$$

The parties then combine their shares of y' with their shares of W to yield W' as specified by $\mathcal{F}_A$. For $j \in [1, n]$:

$$W'^j = \begin{cases} W^j \oplus y'^j = W^j \oplus W^i \oplus v' = v' & \text{if } j = i \\ W^j \oplus 0 = W^j & \text{otherwise} \end{cases} \tag{3}$$

By the conjunction of Equations 2 and 3 we have Equation 1, and thus Lemma B.1 holds. □

Lemma B.2 (Security for π_A). *If* (Gen, Eval) *is a secure FSS scheme for DPFs, π_{C_1} and π_{C_2} are secure multiparty computation protocols for C_1 and C_2 respectively, and* Prf *is a Pseudo-random Function Family, then for each party $p \in \{a, b\}$ there exists a simulator Sim_p^A such that:*

$$\begin{aligned} &\left\{\left(\mathsf{Sim}_p^A\left(\mathsf{Input}_p^A, \mathcal{F}_{Ap}\left(\mathsf{Input}^A\right)\right), \mathcal{F}_A\left(\mathsf{Input}^A\right)\right)\right\}_{\mathsf{Input}^A \in \mathsf{Dom}^A_{n,m}} \\ &\overset{c}{\equiv} \left\{\left(\mathsf{View}_p^{\pi_A}\left(\mathsf{Input}^A\right), \mathsf{output}^{\pi_A}(\mathsf{Input}^A)\right)\right\}_{\mathsf{Input}^A \in \mathsf{Dom}^A_{n,m}} \end{aligned}$$

Proof. If (Gen, Eval) is a secure FSS scheme for DPFs, then by Definition 2.3 there must exist some simulator, $\mathsf{Sim}^{\mathsf{FSS}}$, for FSS keys. Similarly, if π_{C_1} and π_{C_2} are secure multiparty computation protocols, then by Definition A.1 for $p \in \{a, b\}$ there must exist simulators $\mathsf{Sim}_p^{C_1}$ and $\mathsf{Sim}_p^{C_2}$ for those protocols. We begin by specifying a simulator for π_A, Sim_p^A, which has access to $\mathsf{Sim}^{\mathsf{FSS}}$, $\mathsf{Sim}_p^{C_1}$, and $\mathsf{Sim}_p^{C_2}$, as well as $\mathcal{L}_1$ and the inverse of $\mathcal{L}_2$ with respect to v^Δ:

$$v^\Delta = \mathcal{L}_2^{-1}\left(W_p, W'_p\right) = \max\left(\left\{W_p^x \oplus W_p'^x\right\}_{x \in [1,n]}\right)$$

Simulator Sim_p^A is given party p's share of the inputs for π_A, along with the output of $\mathcal{F}_A$, upon which it performs the procedure given in Figure 13. Roughly speaking, it uses $\mathsf{Sim}^{\mathsf{FSS}}$ along with $\mathcal{L}_1$ and $\mathcal{L}_2^{-1}$ to simulate the inputs and outputs for C_1 and C_2 given the known inputs and outputs for π_A, and then passes these to $\mathsf{Sim}_p^{C_1}$ and $\mathsf{Sim}_p^{C_2}$. Our proof proceeds via a series of hybrid views.

First Hybrid. Our first hybrid, $\mathcal{H}_1$, is identical to the real-world view, except that subsequent to the evaluation of circuit C_1, we discard the messages produced by the real circuit and replace them with

$$\mathsf{Sim}_p^{C_1}\left(\mathsf{Input}_p^{C_1}, \mathsf{Output}_p^{\pi_{C_1}}\right) : \mathsf{Msgs}_{\mathsf{Sim}}^{C_1}$$

function $\mathsf{Sim}_p^{\pi_A}\left(\mathsf{Input}_p^A, \mathcal{F}_{Ap}\left(\mathsf{Input}^A\right)\right)$:

// Parse Input_p^A as $\left(i_p, R, f, v_p^f, \mathsf{Stash}_p, k_p^{\mathsf{PRF}}, W_p\right)$

// Parse $\mathcal{F}_{Ap}\left(\mathsf{Input}^A\right)$ as $\left(y_p^f, \mathsf{Stash}'_p, W'_p\right)$

$k_{\mathsf{Sim}}^{\mathsf{FSS}} \leftarrow \mathsf{Sim}^{\mathsf{FSS}}\left(p, 1^\lambda\right)$

$\beta_{\mathsf{Sim}} \xleftarrow{\$} \{0,1\}^\lambda$

$\mathsf{Msgs}_{\mathsf{Sim}}^{C_1} \leftarrow \mathsf{Sim}_p^{C_1}\left(i_p, \beta_{\mathsf{Sim}}, k_{\mathsf{Sim}}^{\mathsf{FSS}}\right)$

$v_{\mathsf{Sim}} \leftarrow \mathcal{L}_1\left(k_{\mathsf{Sim}}^{\mathsf{FSS}}, R\right)$

$v_{\mathsf{Sim}}^\Delta \leftarrow \mathcal{L}_2^{-1}\left(W_p, W'_p\right)$

$\mathsf{Msgs}_{\mathsf{Sim}}^{C_2} \leftarrow \mathsf{Sim}_p^{C_2}\left(i_p, k_{\mathsf{Sim}}^{\mathsf{FSS}}, \beta_{\mathsf{Sim}}, v_{\mathsf{Sim}}, f, v_p^f, \mathsf{Stash}_p, k_p^{\mathsf{PRF}}, v_{\mathsf{Sim}}^\Delta, y_p^f, \mathsf{Stash}'_p\right)$

return $\left(r_{\mathsf{Sim}}, i_p, \beta_{\mathsf{Sim}}, \mathsf{Msgs}_{\mathsf{Sim}}^{C_1}, k_{\mathsf{Sim}}^{\mathsf{FSS}}, R, f, v_p^f, \mathsf{Stash}_p, k_p^{\mathsf{PRF}}, \mathsf{Msgs}_{\mathsf{Sim}}^{C_2}, v_{\mathsf{Sim}}^\Delta, y_p^f, \mathsf{Stash}'_p, W_p, W'_p\right)$

Figure 13: Pseudocode for a simulator for the access protocol π_A.

Consequently, the view produced by $\mathcal{H}_1$ for party p is identical to p's view of the real protocol, except where these messages differ.

Suppose there were a probabilistic polynomial time (PPT) distinguisher, D_1, that could distinguish between the ensembles

$$E_p^{\pi_A} = \left\{\left(\mathsf{View}_p^{\pi_A}\left(\mathsf{Input}^A\right), \mathsf{Output}^{\pi_A}\left(\mathsf{Input}^A\right)\right)\right\}$$
$$E_p^{\mathcal{H}_1} = \left\{\left(\mathsf{View}_p^{\mathcal{H}_1}\left(\mathsf{Input}^A\right), \mathsf{Output}^{\mathcal{H}_1}\left(\mathsf{Input}^A\right)\right)\right\}$$

for some input Input^A with some advantage δ_1. We could use D_1 to construct a distinguisher D_2 for the MPC protocol that evaluates C_1. D_2 is given some $\mathsf{View}_p^{D_2}$, produced either by a real evaluation of π_{C_1} or by $\mathsf{Sim}_p^{C_1}$. Additionally, it is given the two-party output of the associated functionality, Output^{D_2}, and some nonuniform auxiliary information, Aux^{D_2} (chosen as a function of $\mathsf{View}_p^{D_2}$, Output^{D_2}, and D_1 to give D_2 the best possible advantage). Furthermore, D_2 has access to the circuit C_2. D_2 performs the procedure specified in Figure 14.

If $\mathsf{View}_p^{D_2}$ was generated by $\mathsf{Sim}_p^{C_1}$, then $E_p^{D_1} = E_p^{\mathcal{H}_1}$, whereas if it was generated by a real evaluation of the circuit C_1, then $E_p^{D_1} = E_p^{\pi_A}$. D_2 makes a single call to D_1, and all of the inputs to D_1 that are not determined by $\mathsf{View}_p^{D_2}$ or Output^{D_2} are given as non-uniform advice to provide the best discriminatory power; thus it must be the case that D_2 has advantage δ_2 such that $\delta_2 = \delta_1$. By Definition A.1, for security parameter λ and all choices of Input^{C_1}:

$$\delta_2 = \left|\begin{array}{r}\Pr\left[D_2\left(\mathsf{View}_p^{\pi_{C_1}}\left(\mathsf{Input}^{C_1}\right), \mathsf{Output}^{\pi_{C_1}}\left(\mathsf{Input}^{C_1}\right)\right) = 1\right] \\ -\Pr\left[D_2\left(\mathsf{Sim}_p^{C_1}\left(\mathsf{Input}_p^{C_1}\right), \mathcal{F}_{C_1}\left(\mathsf{Input}^{C_1}\right)\right) = 1\right]\end{array}\right| < \frac{1}{\mathsf{poly}(\lambda)}$$

function $D_2\left(\mathsf{View}_p^{D_2}, \mathsf{Output}^{D_2}, \mathsf{Aux}^{D_2}\right)$:

// Parse $\mathsf{View}_p^{D_2}$ as $\left(i_p, \beta_p, \mathsf{Msgs}_p^{C_1}, k_p^{\mathsf{FSS}}\right)$

// Parse Output^{D_2} as $\left(k_p^{\mathsf{FSS}}, k_q^{\mathsf{FSS}}\right)$

// Parse Aux^{D_2} as $\left(i_q, \beta_q, R, v_p^f, v_q^f, \mathsf{Stash}_p, \mathsf{Stash}_q, k_p^{\mathsf{PRF}}, k_q^{\mathsf{PRF}}, W_p, W_q\right)$

$v_p \leftarrow \mathcal{L}_1\left(k_p^{\mathsf{FSS}}, R\right)$

$v_q \leftarrow \mathcal{L}_1\left(k_q^{\mathsf{FSS}}, R\right)$

$\mathsf{Input}^{C_2} \leftarrow \left(i_p, i_q, v_p, v_q, v_p^f, v_q^f, \beta_p, \beta_q, \mathsf{Stash}_p, \mathsf{Stash}_q, k_p^{\mathsf{FSS}}, k_q^{\mathsf{FSS}}, k_p^{\mathsf{PRF}}, k_q^{\mathsf{PRF}}\right)$

// Evaluate both parties' portions of the protocol for C_2

$\left(v^\Delta, y_p^f, \mathsf{Stash}'_p\right) \leftarrow \mathsf{Output}_p^{\pi_{C_2}}\left(\mathsf{Input}^{C_2}\right)$

$\left(v^\Delta, y_q^f, \mathsf{Stash}'_q\right) \leftarrow \mathsf{Output}_q^{\pi_{C_2}}\left(\mathsf{Input}^{C_2}\right)$

$W'_p \leftarrow \mathcal{L}_2\left(k_p^{\mathsf{FSS}}, v^\Delta, W_p\right)$

$W'_q \leftarrow \mathcal{L}_2\left(k_q^{\mathsf{FSS}}, v^\Delta, W_q\right)$

$E_p^{D_1} \leftarrow \left(\left(\mathsf{View}_p^{D_2}, \mathsf{View}_p^{\pi_{C_2}}\left(\mathsf{Input}^{C_2}\right), R, W_p, W'_p\right), \left(y^f, \mathsf{Stash}', W'\right)\right)$

return $D_1\left(E_p^{D_1}\right)$

Figure 14: Pseudocode for distinguisher D_2 for MPC protocols. This distinguisher takes nonuniform input Aux^{D_2} and has access to a distinguisher D_1 for the ensembles $E_p^{\mathcal{H}_1}$ and $E_p^{\pi_A}$.

Thus $\delta_1 = \delta_2 < 1/\mathsf{poly}(\lambda)$, and $\mathcal{H}_1$ is computationally indistinguishable from the real view.

Second Hybrid. Our second hybrid, $\mathcal{H}_2$, is identical to the first, except that we omit the evaluation of C_1, and replace its outputs as follows: choose $\beta_{\mathsf{Sim}} \xleftarrow{\$} \{0,1\}^\lambda$ (Note that β_{Sim} has the same distribution as the real value), and then generate the FSS key using a DPF simulator:

$$k_{\mathsf{Sim}}^{\mathsf{FSS}} \leftarrow \mathsf{Sim}^{\mathsf{FSS}}\left(p, 1^\lambda\right)$$

Suppose there were a probabilistic polynomial time (PPT) distinguisher, D_3, that could distinguish between the ensembles

$$E_p^{\mathcal{H}_1} = \left\{\left(\mathsf{View}_p^{\mathcal{H}_1}\left(\mathsf{Input}^A\right), \mathsf{Output}^{\mathcal{H}_1}\left(\mathsf{Input}^A\right)\right)\right\}$$
$$E_p^{\mathcal{H}_2} = \left\{\left(\mathsf{View}_p^{\mathcal{H}_2}\left(\mathsf{Input}^A\right), \mathsf{Output}^{\mathcal{H}_2}\left(\mathsf{Input}^A\right)\right)\right\}$$

for some input Input^A with some advantage δ_3. We could use D_3 to construct a distinguisher D_4 for FSS keys. D_4 has access to the simulator $\mathsf{Sim}_p^{C_1}$, as well as the real circuit C_2. Given some FSS key, k_D^{FSS}, created either by the real FSS Gen algorithm, or by its simulator, $\mathsf{Sim}^{\mathsf{FSS}}$, and some nonuniform auxiliary information, Aux^{D_4}, D_4 performs the procedure given in Figure 15.

If k_D^{FSS} was generated by $\mathsf{Sim}^{\mathsf{FSS}}$, then $E_p^{D_3} = E_p^{\mathcal{H}_2}$, whereas if it was generated by a real instance of the FSS Gen algorithm, then

function $D_4\left(k_D^{\mathsf{FSS}}, \mathsf{Aux}^{D_4}\right)$:

// Parse Aux^{D_4} as $\left(\begin{array}{l} i_p, i_q, \beta_q, k_q^{\mathsf{FSS}}, R, \\ v_p^f, v_q^f, \mathsf{Stash}_p, \mathsf{Stash}_q, \\ k_p^{\mathsf{PRF}}, k_q^{\mathsf{PRF}}, W_p, W_q \end{array}\right)$

$\beta_{\mathsf{Sim}} \xleftarrow{\$} \{0,1\}^\lambda$

$\mathsf{View}_p^{\mathsf{Sim}^{C_1}} \leftarrow \mathsf{Sim}_p^{C_1}\left(i_p, \beta_{\mathsf{Sim}}, k_D^{\mathsf{FSS}}\right)$

$v_p \leftarrow \mathcal{L}_1\left(k_D^{\mathsf{FSS}}, R\right)$

$v_q \leftarrow \mathcal{L}_1\left(k_q^{\mathsf{FSS}}, R\right)$

$\mathsf{Input}^{C_2} \leftarrow \left(\begin{array}{l} i_p, i_q, v_p, v_q, v_p^f, v_q^f, \beta_{\mathsf{Sim}}, \beta_q, \\ \mathsf{Stash}_p, \mathsf{Stash}_q, k_D^{\mathsf{FSS}}, k_q^{\mathsf{FSS}}, k_p^{\mathsf{PRF}}, k_q^{\mathsf{PRF}} \end{array}\right)$

// Evaluate both parties' portions of the protocol for C_2

$\left(v^\Delta, y_p^f, \mathsf{Stash}_p'\right) \leftarrow \mathsf{Output}_p^{\pi_{C_2}}\left(\mathsf{Input}^{C_2}\right)$

$\left(v^\Delta, y_q^f, \mathsf{Stash}_q'\right) \leftarrow \mathsf{Output}_q^{\pi_{C_2}}\left(\mathsf{Input}^{C_2}\right)$

$W_p' \leftarrow \mathcal{L}_2\left(k_D^{\mathsf{FSS}}, v^\Delta, W_p\right)$

$W_q' \leftarrow \mathcal{L}_2\left(k_q^{\mathsf{FSS}}, v^\Delta, W_q\right)$

$E_p^{D_3} \leftarrow \left(\left(\begin{array}{l} \mathsf{View}_p^{\mathsf{Sim}^{C_1}}, \mathsf{View}_p^{\pi_{C_2}}\left(\mathsf{Input}^{C_2}\right), \\ R, W_p, W_p' \end{array}\right), \left(y^f, \mathsf{Stash}', W'\right)\right)$

return $D_3\left(E_p^{D_3}\right)$

Figure 15: Pseudocode for distinguisher D_4 for FSS keys. This distinguisher takes nonuniform input Aux^{D_4} and has access to the distinguisher D_3 for ensembles $E_p^{\mathcal{H}_1}$ and $E_p^{\mathcal{H}_2}$.

$E_p^{D_3} = E_p^{\mathcal{H}_1}$. D_4 makes a single call to D_3, and inputs to D_3 that are not determined by k_D^{FSS} or β_{Sim} are given as non-uniform advice; thus it must be the case that D_4 has advantage δ_4 such that $\delta_4 = \delta_3$. By Definition 2.3, for security parameter λ and all α, β, and p:

$$\delta_4 = \left| \begin{array}{r} \Pr\left[D_4\left(\mathsf{Gen}\left(1^\lambda, f_{\alpha,\beta}\right) : k_p^{\mathsf{FSS}}\right) = 1\right] \\ -\Pr\left[D_4\left(\mathsf{Sim}^{\mathsf{FSS}}\left(p, 1^\lambda\right)\right) = 1\right] \end{array} \right| < \frac{1}{\mathsf{poly}(\lambda)}$$

Thus $\delta_3 = \delta_4 < 1/\mathsf{poly}(\lambda)$, and $\mathcal{H}_2$ is computationally indistinguishable from $\mathcal{H}_1$.

Third Hybrid. The third hybrid, $\mathcal{H}_3$, is identical to the second, except that, subsequent to the evaluation of C_2, we discard its messages and replace them with

$$\mathsf{Sim}_p^{C_2}\left(f, \mathsf{Input}_p^{C_2}, \mathsf{Output}_p^{\pi_{C_2}}\right) : \mathsf{Msgs}_{\mathsf{Sim}}^{C_2}$$

Suppose there were a PPT distinguisher, D_5, that could distinguish between the ensembles

$$E_p^{\mathcal{H}_2} = \left\{\left(\mathsf{View}_p^{\mathcal{H}_2}\left(\mathsf{Input}^A\right), \mathsf{Output}^{\mathcal{H}_2}\left(\mathsf{Input}^A\right)\right)\right\}$$
$$E_p^{\mathcal{H}_3} = \left\{\left(\mathsf{View}_p^{\mathcal{H}_3}\left(\mathsf{Input}^A\right), \mathsf{Output}^{\mathcal{H}_3}\left(\mathsf{Input}^A\right)\right)\right\}$$

function $D_6\left(\mathsf{View}_p^{D_6}, \mathsf{Output}^{D_6}, \mathsf{Aux}^{D_6}\right)$:

// Parse $\mathsf{View}_p^{D_6}$ as $\left(\begin{array}{l} i_p, v_p, v_p^f, \beta_{\mathsf{Sim}}, \mathsf{Stash}_p, k_{\mathsf{Sim}}^{\mathsf{FSS}}, \\ k_p^{\mathsf{PRF}}, \mathsf{Msgs}_p^{C_2}, v^\Delta, y_p^f, \mathsf{Stash}_p' \end{array}\right)$

// Parse Output^{D_6} as $\left(v^\Delta, y^f, \mathsf{Stash}'\right)$

// Parse Aux^{D_6} as $\left(k_q^{\mathsf{FSS}}, R, W_p, W_q\right)$

$\mathsf{View}_p^{\mathsf{Sim}^{C_1}} \leftarrow \mathsf{Sim}_p^{C_1}\left(i_p, \beta_{\mathsf{Sim}}, k_{\mathsf{Sim}}^{\mathsf{FSS}}\right)$

$v_p \leftarrow \mathcal{L}_1\left(k_{\mathsf{Sim}}^{\mathsf{FSS}}, R\right)$

$W_p' \leftarrow \mathcal{L}_2\left(k_{\mathsf{Sim}}^{\mathsf{FSS}}, v^\Delta, W_p\right)$

$W_q' \leftarrow \mathcal{L}_2\left(k_q^{\mathsf{FSS}}, v^\Delta, W_q\right)$

$E_p^{D_5} \leftarrow \left(\left(\begin{array}{l} \mathsf{View}_p^{\mathsf{Sim}^{C_1}}, \mathsf{View}_p^{D_6} \\ R, W_p, W_p' \end{array}\right), \left(y^f, \mathsf{Stash}', W'\right)\right)$

return $D_5\left(E_p^{D_5}\right)$

Figure 16: Pseudocode for distinguisher D_6 for MPC protocols. This distinguisher takes nonuniform input Aux^{D_6} and has access to the distinguisher D_5 for ensembles $E_p^{\mathcal{H}_2}$ and $E_p^{\mathcal{H}_3}$.

for some Input^A with some advantage δ_5. We could use D_5 to construct a distinguisher D_6 for the MPC protocol that evaluates C_2. D_6 has access to $\mathsf{Sim}_p^{C_1}$, and as input it is given some view, $\mathsf{View}_p^{D_6}$, which was produced either by a real evaluation of C_2 or by its simulator, $\mathsf{Sim}_p^{C_2}$. Given $\mathsf{View}_p^{D_6}$, the two-party output of the associated functionality, Output^{D_6}, and some non-uniform auxilliary information Aux^{D_6}, D_6 follows the procedure given in Figure 16. Note that the distinguisher does not simulate the FSS key, because a simulation of a key is included in the view to be distinguished.

If $\mathsf{View}_p^{D_6}$ was generated by $\mathsf{Sim}_p^{C_2}$, then $E_p^{D_5} = E_p^{\mathcal{H}_3}$, whereas if it was generated by a real evaluation of the circuit C_2, then $E_p^{D_5} = E_p^{\mathcal{H}_2}$. D_6 makes a single call to D_5, and all of the inputs to D_5 that are not determined by $\mathsf{View}_p^{D_6}$ or Output^{D_6} are chosen non-uniformly to provide the best discriminatory power; thus it must be the case that D_6 has advantage δ_6 such that $\delta_6 = \delta_5$. By Definition A.1, for security parameter λ and all choices of Input^{C_2}:

$$\delta_6 = \left| \begin{array}{r} \Pr\left[D_6\left(\mathsf{View}_p^{\pi_{C_2}}\left(\mathsf{Input}^{C_2}\right), \mathsf{Output}^{\pi_{C_2}}\left(\mathsf{Input}^{C_2}\right)\right) = 1\right] \\ -\Pr\left[D_6\left(\mathsf{Sim}_p^{C_2}\left(\mathsf{Input}_p^{C_2}\right), \mathcal{F}_{C_2}\left(\mathsf{Input}^{C_2}\right)\right) = 1\right] \end{array} \right| < \frac{1}{\mathsf{poly}(\lambda)}$$

Thus $\delta_5 = \delta_6 < 1/\mathsf{poly}(\lambda)$, and $\mathcal{H}_3$ is computationally indistinguishable from $\mathcal{H}_2$.

Fourth Hybrid. Finally, we return to the full simulator, Sim_p^A, which we specified in Figure 13. The simulator is identical to $\mathcal{H}_3$, except that we omit C_2 entirely. The simulator is provided $\mathcal{F}_{Ap}(\mathsf{Input}^A)$ as input, from which it extracts the necessary values of y_p^f, Stash_p', and W_p'. $\mathcal{L}_2^{-1}$ is employed to derive v^Δ (which is an

input to the simulator for C_2) from W_p and W'_p. We conclude that for all parties:

$$\left\{\mathcal{F}_{Ap}\left(\mathsf{Input}^A\right)\right\}_{\mathsf{Input}^A \in \mathsf{Dom}^A_{n,m}}$$
$$= \left\{\mathsf{Output}^{\mathcal{H}_3}\left(\mathsf{Input}^A\right)\right\}_{\mathsf{Input}^A \in \mathsf{Dom}^A_{n,m}} \implies$$
$$E_p^{\mathcal{H}_3} = \left\{\left(\mathsf{Sim}_p^A\left(\mathsf{Input}_p^A\right), \mathcal{F}_A\left(\mathsf{Input}^A\right)\right)\right\}_{\mathsf{Input}^A \in \mathsf{Dom}^A_{n,m}}$$

Thus by transitivity and Lemma B.1, Lemma B.2 holds. □

Corollary. In order to call the access protocol multiple times upon the same data, it is necessary to show that the state it outputs is also a valid input state, input validity being assumed by Lemmas B.1 and B.2. Notice that in the specification of C_2, we remove any existing elements from the stash that have the index i, and append (i, v'). Consequently, we have a corollary:

Corollary B.3. *Assuming that π_A is a secure protocol under Definition A.1, for any valid input,*

$$\mathsf{Input}^A = \left(i, R, f, v^f, \mathsf{Stash}, k_a^{\mathsf{PRF}}, k_b^{\mathsf{PRF}}, W\right)$$

if $(y^f, \mathsf{Stash}', W') \leftarrow \mathsf{Output}^{\pi_A}(\mathsf{Input}^A)$, then for any f' and any $v^{f'}$ that is valid relative to f', and any $i' \in [1, n]$,

$$\mathsf{Input}'^A = \left(i', R, g', v^{f'}, \mathsf{Stash}', k_a^{\mathsf{PRF}}, k_b^{\mathsf{PRF}}, W'\right)$$

is a valid input for π_A.

B.2 Proof of Security for Initialization

Real-world View. The real-world view of party p of the initialization protocol π_I comprises p's random tape r_p, the inputs and outputs of the protocol, and the messages received by p. As specified in Section 4 and illustrated in Figure 4, party p receives only one message, W'_q (where q is p's counterparty), which is a copy of q's local state, W_q, that has been masked by a PRF under an unknown key. Thus we have

$$\mathsf{Input}_p^I = \left(W_p\right)$$
$$\mathsf{Input}^I = \left(W_p, W_q\right)$$
$$\mathsf{Output}_p^{\pi_I}\left(\mathsf{Input}^I\right) = \left(k_p^{\mathsf{PRF}}, R'\right)$$
$$\mathsf{View}_p^{\pi_I}\left(\mathsf{Input}^I\right) = \left(r_p, W_p, k_p^{\mathsf{PRF}}, W'_p, W'_q, R'\right)$$

Lemma B.4 (Correctness for π_I). *For each party $p \in \{a, b\}$*

$$\left\{\mathcal{F}_I\left(\mathsf{Input}^I\right)\right\}_{\mathsf{Input}^I \in \mathsf{Dom}^I_{n,m}}$$
$$= \left\{\mathsf{Output}^{\pi_I}\left(\mathsf{Input}^I\right)\right\}_{\mathsf{Input}^I \in \mathsf{Dom}^I_{n,m}}$$

Proof. The ideal functionality for initialization, $\mathcal{F}_I\left(\mathsf{Input}^I\right)$, is specified in Figure 17. By comparison with the protocol specification given in Figure 4, we observe that in both the ideal functionality and the actual protocol, new PRF keys are chosen uniformly at random. We further observe that R' is calculated identically in both, the only difference being in the associativity of XOR operations. Thus, the output of the real initialization protocol is identical to that of the ideal functionality. □

1 **function** $\mathcal{F}_I\left(\mathsf{Input}^I\right)$:
2 // Parse Input^I as (W)
3 $k_a^{\mathsf{PRF}} \xleftarrow{\$} \{0,1\}^\lambda$
4 $k_b^{\mathsf{PRF}} \xleftarrow{\$} \{0,1\}^\lambda$
5 $R' \leftarrow \left\{\mathsf{Prf}_{k_a^{\mathsf{PRF}}}(i) \oplus \mathsf{Prf}_{k_b^{\mathsf{PRF}}}(j) \oplus W^j\right\}_{j\in[1,n]}$
6 **return** $\left(k_a^{\mathsf{PRF}}, k_b^{\mathsf{PRF}}, R'\right)$
7
8 **function** $\mathcal{F}_{Ip}\left(\mathsf{Input}^I\right)$:
9 $\left(k_a^{\mathsf{PRF}}, k_b^{\mathsf{PRF}}, R'\right) \leftarrow \mathcal{F}_I\left(\mathsf{Input}^I\right)$
10 **return** $\left(k_p^{\mathsf{PRF}}, R'\right)$

Figure 17: Pseudocode for the ideal functionality of the initialization protocol π_I.

1 **function** $\mathsf{Sim}_p^I\left(\mathsf{Input}_p^I, \mathcal{F}_{Ip}\left(\mathsf{Input}^I\right)\right)$:
2 // Parse Input_p^I as $\left(W_p\right)$
3 // Parse $\mathcal{F}_{Ip}\left(\mathsf{Input}^I\right)$ as $\left(k_p^{\mathsf{PRF}}, R'\right)$
4 $W'_p \leftarrow \left\{\mathsf{Prf}_{k_p^{\mathsf{PRF}}}(j) \oplus W_p^j\right\}_{j\in[1,n]}$
5 $W'_q \leftarrow W'_p \oplus R'$
6 **return** $\left(r_{\mathsf{Sim}}, W_p, k_p^{\mathsf{PRF}}, W'_p, W'_q, R'\right)$

Figure 18: Pseudocode for a simulator for the initialization protocol π_I.

Lemma B.5 (Security for π_I). *If* Prf *is a pseudo-random function family, then for each party $p \in \{a, b\}$ there exists of simulator Sim_p^I for π_I such that:*

$$\left\{\left(\mathsf{Sim}_p^I\left(\mathsf{Input}_p^I, \mathcal{F}_{Ip}\left(\mathsf{Input}^I\right)\right), \mathcal{F}_I\left(\mathsf{Input}^I\right)\right)\right\}_{\mathsf{Input}^I \in \mathsf{Dom}^I_{n,m}}$$
$$= \left\{\left(\mathsf{View}_p^{\pi_I}\left(\mathsf{Input}^I\right), \mathsf{Output}^{\pi_I}\left(\mathsf{Input}^I\right)\right)\right\}_{\mathsf{Input}^I \in \mathsf{Dom}^I_{n,m}}$$

Proof. We specify a simulator, Sim_p^I, which receives as inputs both the inputs and outputs of the original protocol and performs the procedure given in Figure 18. The view produced by the simulator and the one generated by the real evaluation are actually identical, and thus by Lemma B.4, Lemma B.5 holds. □

Corollary B.6. *Assuming that π_I is a secure protocol under Definition A.1, for any input $\mathsf{Input}^I = (W)$, if $(k_a^{\mathsf{PRF}}, k_b^{\mathsf{PRF}}, R') \leftarrow \mathsf{Output}^{\pi_I}\left(\mathsf{Input}^I\right)$ and $\mathsf{Stash} \leftarrow \varnothing$ then for any f and any v^f that is valid relative to f, and any $i \in [1, n]$,*

$$\mathsf{Input}^A = \left(i, R', f, v^f, \mathsf{Stash}, k_a^{\mathsf{PRF}}, k_b^{\mathsf{PRF}}, W\right)$$

is a valid input for π_A

function $\mathsf{View}^{\pi_F}\left(\mathsf{Input}^F\right)$:

// Parse Input^F as $\left(W, \left\{\left(f^j, i^j, v^{f^j}\right)\right\}_{j\in[2,\ell]}\right)$

$\left(k_a^{\mathsf{FSS}}, k_b^{\mathsf{FSS}}, R\right) \leftarrow \mathsf{Output}^{\pi_I}(W)$

$\left\{s_p^1\right\}_{p\in\{a,b\}} \leftarrow \left\{\mathsf{View}_p^{\pi_I}(W)\right\}_{p\in\{a,b\}}$

$\mathsf{Stash} \leftarrow \varnothing$

for $j \in [2, \ell]$:

$\mathsf{Input}^{Aj} \leftarrow \left(i_p^j, R, f^j, v^{f^j}, \mathsf{Stash}, k_a^{\mathsf{PRF}}, k_b^{\mathsf{PRF}}, W\right)$

$\left(y^{f^j}, \mathsf{Stash}', W'\right) \leftarrow \mathsf{Output}^{\pi_A}\left(\mathsf{Input}^{Aj}\right)$

$\left\{s_p^j\right\}_{p\in\{a,b\}} \leftarrow \left\{\mathsf{View}_p^{\pi_A}\left(\mathsf{Input}^{Aj}\right)\right\}_{p\in\{a,b\}}$

$W \leftarrow W'$

$\mathsf{Stash} \leftarrow \mathsf{Stash}'$

$\left\{\mathcal{S}_p\right\}_{p\in\{a,b\}} \leftarrow \left\{\left\{s_p^j\right\}_{j\in[1,\ell]}\right\}_{p\in\{a,b\}}$

return $(\mathcal{S}_a, \mathcal{S}_b)$

function $\mathsf{View}_p^{\pi_F}\left(\mathsf{Input}^F\right)$:

$(\mathcal{S}_a, \mathcal{S}_b) \leftarrow \mathsf{View}^{\pi_F}\left(\mathsf{Input}^F\right)$

return $\mathcal{S}_p$

Figure 19: Pseudocode for a party's view of Floram over an epoch.

function $\mathcal{F}_F\left(\mathsf{Input}^F\right)$:

// Parse Input^F as $\left(W, \left\{\left(f^j, i^j, v^{f^j}\right)\right\}_{j\in[2,\ell]}\right)$

$\left(k_a^{\mathsf{FSS}}, k_b^{\mathsf{FSS}}, R\right) \leftarrow \mathcal{F}_I(W)$

$\mathsf{Stash} \leftarrow \varnothing$

for $j \in [2, \ell]$:

$\mathsf{Input}^{Aj} \leftarrow \left(i_p^j, R, f^j, v^{f^j}, \mathsf{Stash}, k_a^{\mathsf{PRF}}, k_b^{\mathsf{PRF}}, W\right)$

$\left(y^{f^j}, \mathsf{Stash}', W'\right) \leftarrow \mathcal{F}_A\left(\mathsf{Input}^{Aj}\right)$

$W \leftarrow W'$

$\mathsf{Stash} \leftarrow \mathsf{Stash}'$

return $\left(W', \left\{y^{f^j}\right\}_{j\in[2,\ell]}\right)$

function $\mathcal{F}_{Fp}\left(\mathsf{Input}^F\right)$:

$\left(W', \left\{y^{f^j}\right\}_{j\in[2,\ell]}\right) \leftarrow \mathcal{F}_F\left(\mathsf{Input}^F\right)$

return $\left(W'_p, \left\{y_p^{f^j}\right\}_{j\in[2,\ell]}\right)$

Figure 20: Pseudocode for ideal functionality of Floram over an epoch.

B.3 Proof of Security for Floram

Real-world view. Finally, we prove the security of Floram under Definition A.5. As we mentioned previously, our construction differs from DRAM as specified in Definition A.3 in that it accepts arbitrary functions as input rather than simple read and write commands. Thus

$$\mathsf{Input}_p^F = \left(V_p, \left\{\left(f^j, i_p^j, v_p^{f^j}\right)\right\}_{j\in[2,\ell]}\right)$$

where ℓ is the length of the epoch. We formally specify a party's view of Floram over an epoch in Figure 19, and we specify the ideal functionality $\mathcal{F}_F$ in Figure 20. Note that the protocol specification π_F for Floram over an epoch is identical, except that it replaces the ideal functionalities $\mathcal{F}_I$ and $\mathcal{F}_A$ with the protocols π_I and π_A respectively.

Theorem B.7 (Security for Floram). *If π_I is a secure initialization protocol and π_A is a secure access protocol and* Prf *is a Pseudorandom Function Family, then for each party $p \in \{a, b\}$, there exists a simulator Sim_p^F such that:*

$$\left\{\mathsf{Sim}_p^F\left(\mathsf{Input}_p^F, \mathcal{F}_{Fp}\left(\mathsf{Input}^F\right)\right)\right\}_{\mathsf{Input}^F\in\mathsf{Dom}_{n,m,\lambda}^F}$$
$$\stackrel{c}{\equiv} \left\{\mathsf{View}_p^{\pi_F}\left(\mathsf{Input}^F\right)\right\}_{\mathsf{Input}^F\in\mathsf{Dom}_{n,m,\lambda}^F}$$

where $\mathsf{Dom}_{n,m,\lambda}^F$ is the set of valid epochs with lengths ℓ in $O(\mathsf{poly}(\lambda))$.

Proof. We specify Sim_p^F in Figure 21. Notice, first, that $k_{\mathsf{Sim}}^{\mathsf{PRF}}$ is drawn from an identical distribution to its counterpart in the real view, as are $\mathsf{Stash}_{\mathsf{Sim}}$, and W_{Sim} for all iterations $j \in [2, \ell]$ (those counterparts being secret shares). The only distinguishing features in the simulated view are the messages exchanged in the course of π_A and π_I, and R_{Sim}, which is drawn uniformly from its domain, whereas in the real view it is the XOR of two PRF outputs with V. Our proof will proceed via a series of hybrids.

First Hybrid. The first hybrid, $\mathcal{H}_4$, is identical to the real view, except that after π_I is evaluated, $\mathsf{Msgs}_p^{\pi_I}$ is discarded and replaced with the output of the associated simulator, $\mathsf{Sim}_p^{\pi_I}$. As the two views differ only insofar as the simulated view of π_I differs from a real one, it follows from Lemma B.5 that the two are computationally indistinguishable.

Second Hybrid. The second hybrid, $\mathcal{H}_5$, is identical to $\mathcal{H}_4$, except that after each execution of π_A, the corresponding instance of $\mathsf{Msgs}_p^{\pi_A}$ is discarded, and $\mathsf{Sim}_p^{\pi_A}$ is called to replace it. $\mathcal{H}_5$ differs from $\mathcal{H}_4$ only insofar as the simulated views of π_A differ from the real ones. Thus, it follows from Lemma B.2 and Corollaries B.3 and B.6 (which provide that initialization and access protocols can be chained) that the two are computationally indistinguishable if the epoch length ℓ is in $O(\mathsf{poly}(\lambda))$.

Third Hybrid. Finally, we return to the full simulation, which is identical to $\mathcal{H}_5$ save for two details: First, π_A is omitted entirely, and shares of the stash and WOM (except for the final WOM state) are chosen uniformly from the appropriate domains, as in Figure 21. As the new values are distributed identically to the old, it is necessarily

function $\mathsf{Sim}_p^F\left(\mathsf{Input}_p^F, \mathcal{F}_{Fp}\left(\mathsf{Input}^F\right)\right)$:

// Parse Input_p^F as $\left(W_p, \left\{\left(f^j, i_p^j, v_p^{f^j}\right)\right\}_{j\in[2,\ell]}\right)$

// Parse $\mathcal{F}_{Fp}\left(\mathsf{Input}^F\right)$ as $\left(W'_p, \left\{y_p^{f^j}\right\}_{j\in[2,\ell]}\right)$

$R_{\mathsf{Sim}} \xleftarrow{\$} \{0,1\}^{n\times m}$

$k_{\mathsf{Sim}}^{\mathsf{PRF}} \xleftarrow{\$} \{0,1\}^{\lambda}$

$s_{\mathsf{Sim}}^1 \leftarrow \mathsf{Sim}_p^I\left(W_p, k_{\mathsf{Sim}}^{\mathsf{PRF}}, R_{\mathsf{Sim}}\right)$

$W_{\mathsf{Sim}} \leftarrow W_p$

for $j \in [2, \ell|]$:

  if $j = \ell$:

    $W'_{\mathsf{Sim}} \leftarrow W'_p$

  else:

    $W'_{\mathsf{Sim}} \xleftarrow{\$} \{0,1\}^{n\times m}$

  $\mathsf{Stash}'_{\mathsf{Sim}} \xleftarrow{\$} \{0,1\}^{(j-2)\times m}$

  $s_{\mathsf{Sim}}^j \leftarrow \mathsf{Sim}_p^A\left(\begin{matrix} i_p, R_{\mathsf{Sim}}, f^j, v_p^{f^j}, \mathsf{Stash}_{\mathsf{Sim}}, k_{\mathsf{Sim}}^{\mathsf{PRF}}, W_{\mathsf{Sim}}, \\ y_p^{f^j}, \mathsf{Stash}'_{\mathsf{Sim}}, W'_{\mathsf{Sim}} \end{matrix}\right)$

  $W_{\mathsf{Sim}} \leftarrow W'_{\mathsf{Sim}}$

  $\mathsf{Stash}_{\mathsf{Sim}} \leftarrow \mathsf{Stash}'_{\mathsf{Sim}}$

$\mathcal{S}_{\mathsf{Sim}} \leftarrow \left\{s_{\mathsf{Sim}}^j\right\}_{j\in[1,\ell]}$

return $\mathcal{S}_{\mathsf{Sim}}$

Figure 21: Pseudocode for a simulator for Floram over an epoch.

the case that they give a distinguisher no advantage. Second, R is replaced by R_{Sim} and the evaluation of π_I is omitted. Suppose there existed a PPT algorithm D_7 that could distinguish between the ensembles

$$E_p^{\mathcal{H}_5} = \left\{\left(\mathsf{View}_p^{\mathcal{H}_5}\left(\mathsf{Input}^F, \mathcal{F}_{Fp}\left(\mathsf{Input}^F\right)\right), \mathsf{Output}^{\mathcal{H}_5}\left(\mathsf{Input}^F\right)\right)\right\}$$

$$E_p^{\mathsf{Sim}^F} = \left\{\left(\mathsf{Sim}_p^F\left(\mathsf{Input}_p^F, \mathcal{F}_{Fp}\left(\mathsf{Input}^F\right)\right), \mathcal{F}_F\left(\mathsf{Input}^F\right)\right)\right\}$$

for some valid epoch input^F with advantage δ_7. We could use D_7 to construct a distinguisher D_8 for PRF outputs, as specified in Figure 22, which accepts as input some value $V_{D_8} \in \{0,1\}^{n\times m}$, such that

$$V_{D_8} = \left\{\mathsf{Prf}_k(i)\right\}_{i\in[1,n]}$$

where $k \xleftarrow{\$} \{0,1\}^{\lambda}$ and $\mathsf{Prf}_{\{0,1\}^{\lambda}} : \{0,1\}^m \to \{0,1\}^m$ is a Pseudorandom Function Family, or

$$V_{D_8} = \left\{x \xleftarrow{\$} \{0,1\}^m\right\}_{i\in[1,n]}$$

In the former case, D_8 constructs an ensemble with a distribution identical to $E_p^{\mathcal{H}_5}$, and in the latter case it constructs an ensemble with a distribution identical to $E_p^{\mathsf{Sim}^F}$. D_8 also receives some nonuniform advice Aux^{D_8}. D_8 implements a *statistical test* for PRFs,

function $D_8\left(V_{D_8}, \mathsf{Aux}^{D_8}\right)$:

// Parse Aux^{D_8} as $\left(k_p^{\mathsf{PRF}}, V, W_p, \left\{\left(f^j, i_p^j, v_p^{f^j}, y_p^{f^j}\right)\right\}_{j\in[2,\ell]}\right)$

$R_{D_8} \leftarrow \left\{V_{D_8}^j \oplus \mathsf{Prf}_{k_p^{\mathsf{PRF}}}(j) \oplus V^j\right\}_{j\in[1,n]}$

$s_{D_8}^1 \leftarrow \mathsf{Sim}_p^I\left(W_p, k_p^{\mathsf{PRF}}, R_{D_8}\right)$

$W_{\mathsf{Sim}} \leftarrow W_p$

for $j \in [2, \ell]$:

  $W'_{\mathsf{Sim}} \xleftarrow{\$} \{0,1\}^{n\times m}$

  $\mathsf{Stash}'_{\mathsf{Sim}} \xleftarrow{\$} \{0,1\}^{(j-2)\times m}$

  $s_{D_8}^j \leftarrow \mathsf{Sim}_p^A\left(\begin{matrix} i_p, R_{D_8}, f^j, v_p^{f^j}, \mathsf{Stash}_{\mathsf{Sim}}, k_p^{\mathsf{PRF}}, W_{\mathsf{Sim}}, \\ y_p^{f^j}, \mathsf{Stash}'_{\mathsf{Sim}}, W'_{\mathsf{Sim}} \end{matrix}\right)$

  $W_{\mathsf{Sim}} \leftarrow W'_{\mathsf{Sim}}$

  $\mathsf{Stash}_{\mathsf{Sim}} \leftarrow \mathsf{Stash}'_{\mathsf{Sim}}$

$E_p^{D_7} \leftarrow \left\{s_{D_8}^j\right\}_{j\in[1,\ell]}$

return $D_7\left(E_p^{D_7}\right)$

Figure 22: Pseudocode for distinguisher D_8 for PRF outputs. This distinguisher takes nonuniform input Aux^{D_8} and has access to the distinguisher D_7 for ensembles $E_p^{\mathcal{H}_5}$ and $E_p^{\mathsf{Sim}^F}$.

which succeeds with advantage $\delta_8 = \delta_7$, and a Family of Pseudorandom Functions must admit the success of no statistical test with advantage greater than $1/\mathsf{poly}(\lambda)$ [20]. In other words, it must be the case that for any $n, \lambda \in \mathbb{N}, k \xleftarrow{\$} \{0,1\}^{\lambda}$,

$$\left\{\mathsf{Prf}_k(i)\right\}_{i\in[1,n]} \overset{c}{\equiv} \left\{x \xleftarrow{\$} \{0,1\}^m\right\}_{i\in[1,n]}$$

Consequently, $\delta_7 = \delta_8 \leq 1/\mathsf{poly}(\lambda)$, and by transitivity, over all valid epochs, the output of Sim_p^F is computationally indistinguishable from a real view of party p's local memory, as required. □

Note. A standard hybrid argument yields indistinguishability over sequences of epochs, as required by Definition A.5. The definition allowed the simulator knowledge only of the lengths of the epochs it was to simulate, whereas in this proof we have given the simulator function descriptions for each access as well as shares of inputs and outputs for those functions. However, if a single circuit is constructed to implement both the read and write functionalities, and all accesses in an epoch make use of this circuit (i.e. the functionality of Floram is reduced to simple read and write operations), and if the inputs and outputs are information-theoretic secret shares, then it is unnecessary to pass this extra information, and the statement in Theorem B.7 collapses to that in Definition A.5.