Large-Scale Secure Computation

Elette Boyle[*]
Technion Israel
eboyle@alum.mit.edu

Kai-Min Chung
Academica Sinica
kmchung@iis.sinica.edu.tw

Rafael Pass[†]
Cornell University
rafael@cs.cornell.edu

May 31, 2014

Abstract

We are interested in secure computation protocols in settings where the number of parties is huge and their data even larger. Assuming the existence of a single-use broadcast channel (per player), we demonstrate statistically secure computation protocols for computing (multiple) arbitrary dynamic RAM programs over parties' inputs, handling $(1/3 - \epsilon)$ fraction static corruptions, while preserving up to polylogarithmic factors the computation and memory complexities of the RAM program. Additionally, our protocol is load balanced and has polylogarithmic communication locality.

[*]The research of the first author has received funding from the European Union's Tenth Framework Programme (FP10/ 2010-2016) under grant agreement no. 259426 ERC-CaC.

[†]Pass is supported in part by a Alfred P. Sloan Fellowship, Microsoft New Faculty Fellowship, NSF Award CNS-1217821, NSF CAREER Award CCF-0746990, NSF Award CCF-1214844, AFOSR YIP Award FA9550-10-1-0093, and DARPA and AFRL under contract FA8750-11-2- 0211. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the US Government.

1 Introduction

The notion of secure multi-party computation (MPC), introduced in the seminal works of Yao [Yao86] and Goldreich, Micali and Wigderson [GMW87], is one of the cornerstones in cryptography. An MPC protocol for computing a function f allows a group of parties to jointly evaluate f over their private inputs, with the property that an adversary who corrupts a subset of the parties does not learn anything beyond the inputs of the corrupted parties and the output of the function f.

An emerging area of potential applications for secure MPC is to address privacy concerns in data aggregation and analysis to match the explosive current growth of the amount of available data. Large data sets, such as medical data, transaction data, the web and web access logs, or network traffic data, are now in abundance. Much of the data is stored or made accessible in a distributed fashion. This necessitated the development of efficient distributed protocols that compute over such data. In order to address the privacy concerns associated with such protocols, cryptographic techniques such as MPC for secure function evaluation where *data items are equated with servers* can be utilized to prevent unnecessary leakage of information.

However, before MPC can be effectively used to address today's challenges, we need protocols whose efficiency and communication requirements scale practically to the modern regime of massive data. When the data set contains tens or hundreds of thousands of users' web traffic patterns or consumer transaction data, it becomes unreasonable to assume any single user can provide memory, computation, or communication resources on the order of the data of *all users*. When the same number of parties are involved, it is infeasible to require each individual to communicate to one another. When the computations to be executed are lightweight, depend on a small subset of inputs, or require small memory, it will be unacceptable to obliterate these savings to achieve security. Instead, we seek secure protocols that tightly preserve the *individual* memory and computation requirements of each party with respect to an *efficient representation* of the function to be evaluated, even as the number of parties and expanse of data explodes. Explicitly, when dealing with secure computations in a large-scale setting, the following desiderata are of paramount importance:

1. *Exploiting Random Access.* A range of important programs executed on large data sets access only small number of dynamically chosen data items, rely on conditional branching, or maintain small memory. (Consider, for instance, performing some analysis of *all* Internet users' web behavior.) We cannot afford to lose these properties—which make the computation of the program practically viable in the first place—in exchange for security. This means that converting a program first into a circuit to enable its secure computation, which immediately removes these gains, will not be a feasible option. We thus here focus on secure computation of *RAM programs.* Of course, to achieve standard security where nothing is revealed on parties' inputs beyond the function output (including, in general, information about *which inputs* were accessed in the computation), it is unavoidable that all inputs must be "touched" during the protocol. This imposes a necessary $\Theta(n)$ computation requirement (in total computation complexity). However, we may hope that, aside from this $\Theta(n)$ additive term, secure evaluation will incur only polylogarithmic overhead in the resources required to perform a (non-secure) evaluation of the functionality as described as a RAM program.

2. *Reusability.* Note that while this additive $\Theta(n)$ one-time cost is necessary to achieve security, there is no need for it to repeated for every function evaluation. That is, we can hope to "reuse" the input processing: enabling computation of a sequence of RAM programs Π_j, while requiring only resources $O(\sum Time(\Pi_j))$, plus a *single* additive term of $\Theta(n)$.

3. *Load balancing.* In the large-scale setting, with a vast number of participating parties, there becomes a great divide between the total required resources, and each party's "fair share" of these resources. In such a setting it is of cruicial importantce to *balance* the load across nodes. No individual party should be burdened with computation, communication, or memory comparable to the entire computation, and certainly not to the number of parties (for instance, in our example of performing some analysis of the data of all Internet users, it is clearly infeasible for a single user to store all Internet users' data). Ideally, each party's requirement will be $1/n$ of the corresponding total values. For example, the per-party memory requirement should be equal only to the size of his own input, together with $1/n$ fraction of the space required to evaluate the program.

4. *Communication Locality.* In an ideal world, each party speaks only with a single trusted party. In a protocol, we may hope to preserve (up to polylogarithmic factors) this *locality* of communication: that is, the number of total parties that each party must send and receive message to during the course of the protocol. Communication locality has received comparatively little attention in MPC literature, but plays an important role in feasibility when the number of parties is huge.

As far as we know, to date, no MPC solution addresses all of these desiderata. Roughly, relevant prior works fall into the following two categories.

A first line of research initiated by Damgård and Ishai [DI06], known as *scalable MPC*, focuses on achieving load balancing while maintaining relatively small global complexities [DI06, DN07, DIK$^+$08, DIK10, DKMS12, ZMS14]. However, a common thread among all these results (as with with nearly all works in secure computation to date) is that the function to be evaluated is assumed to be in the form of a *circuit*, and thus computation complexity of the protocols grow linearly in the circuit size (and while we can always transform a RAM program to a circuit, this transformation inflicts a multiplicative factor of the *total data size.*)

A second line of work focuses on MPC for RAM programs, but essentially all results in this line of work focus on secure *two-party* computation [OS97, GKK$^+$11, LO13, GGH$^+$13]. To the best of our knowledge, there have been only two proposed solutions for secure computation of RAM programs in the multiparty setting: An aside in the work of Damgård, Meldgaard, and Nielsen [DMN11] on Oblivious RAM,[1] and the work of Boyle, Goldwasser, and Tessaro [BGT13], focusing on efficient secure computation of functions that query only a sublinear number of parties' inputs. Neither of these works satisfy our goals either: the Damgård *et al.* [DMN11] protocol requires each party to communicate and maintain memory of size equivalent to *all parties'* inputs. The protocol of Boyle *et al.* [BGT13] makes progress toward removing this requirement, but does not support full-fledged RAM program evaluation (their work only supports random access to the players inputs, but otherwise models the computation as a circuit), it does not provide reusability (it requires a *multiplicative* overhead of $\mathsf{poly}(n)$ for each function to be compted), and is heavily load imbalanced.[2]

[1]An Oblivious RAM (ORAM) compiler converts any RAM program and associated database to a modified pair, such that data access patterns of the compiled program hide information about the underlying data.

[2]Additionally, the protocol of [BGT13] relies on strong cryptographic hardness assumptions.

1.1 Our Results: Scalable, Load-Balanced MPC for Dynamic RAM Programs

We achieve MPC for RAM programs on parties' inputs, with a protocol that preserves the *per-party* memory and computation complexity requirements of the participating parties. Our construction is information theoretically secure against $(1/3 - \epsilon)$ statically scheduled corruptions, within a synchronous communication network.

Theorem 1.1 (Informal – Load-Balanced, Communication-Local MPC for RAM Programs)**.** *For any constant $\epsilon > 0$, there exists an n-party statistically secure (with error negligible in n) protocol for computing any adaptively chosen sequence of N RAM programs Π_j (that may have shared state), handling $(1/3-\epsilon)$ fraction static corruptions making an initial use of a single broadcast per party (of $\mathsf{polylog}(n)$ bits), and with the following complexities (where $|x|, |y|$ denote input and output size[3]):*

- *Memory per party: $\tilde{O}\left(|x| + \max_{j=1}^{N} Space(\Pi_j)/n\right)$.*
- *Computation per party: $\tilde{O}\left(|x| + \sum_{j=1}^{N} Time(\Pi_j)/n + N|y|\right)$.*
- *Round complexity: $\tilde{O}\left(\sum_{j=1}^{N} Time(\Pi_j)\right)$.*

where $Space(\Pi)$ and $Time(\Pi)$ denote the worst-case space and runtime requirements of Π over different inputs. Additionally, our protocol achieves $\mathsf{polylog}(n)$ communication locality, and a strong "on-line" load-balancing guarantee such that at all times *during the protocol, all parties' communication and computation loads vary by at most a multiplicative factor of $\mathsf{polylog}(n)$ (up to a $\mathsf{polylog}(n)$ additive term).*

Note that the initial one-time uses of a broadcast channel can be implemented via a communication-local broadcast protocol, such as the $O(\sqrt{n})$-locality protocol of King et al [KSSV06], or the amortized $\mathsf{polylog}(n)$-locality protocol of [BSGH13]. The resulting final protocol achieves comparable locality at the cost of a *one-time* per-party computation and communication overhead of $O(\sqrt{n})$, or $\mathsf{polylog}(n)$, respectively. We separate the broadcast cost from our protocol complexity measures to emphasize that any (existing or future) broadcast protocol can be directly plugged in, yielding associated desirable properties.

Our results are most closely related to those of Dani *et al.* [DKMS12]: we strengthen these results by a) dealing with RAM programs, b) achieving reusability (without incurring a multiplicative overhead in computation and communication) and c) achieving locality (and perhaps less importantly, we also achieive the stronger "on-line" notion of load-balancing).

We additionally remark that even without considering reusability, communication locality, or the communication/computation load balancing properties achieved by our protocol, our results already yield the first protocol whose *total* communication and computation complexities for securely evaluating a *single* RAM program Π grow as $\mathsf{poly}(n) + \tilde{O}(Time(\Pi))$ while simultaneously requiring only $\tilde{O}(|x| + Space(\Pi)/n)$ memory per party. Indeed, all existing solutions either require converting the program into a circuit representation, or require parties to maintain storage $\Omega(n|x|)$. (Looking ahead, this will be achieved already by the first step of our approach.)

A note on the optimality of our parameters: To put our theorem in context, consider the best parameters one could hope to achieve in our setting. For example, to prevent leaking

[3]For simplicity of exposition and analysis (in particular, when load balancing), we will assume all parties receive the same output. However, our protocol can easily be modified to support different outputs for different parties.

information on parties' inputs, any observable complexities of the protocol (such as runtime and memory storage) must take the worst-case rather than input-specific specific values. To achieve correctness, the parties necessarily must collectively store enough information to recover all inputs x_i, and to correctly evaluate each of the programs Π_j; thus the total storage capacity of all parties combined must be $\Omega(n|x| + \max Space|\Pi_j|)$. In order to obtain security, (in particular, in order to hide whose inputs were used in the computation), all parties' inputs must be touched at some point during the protocol; this means the computation complexity must include an additive term $\Omega(n|x|)$.[4] An optimal load balancing would assign to each party $1/n$ fraction of each total resource requirement at any given time in the protocol. And, to participate in the protocol, each party must without exception speak with at least one other party. Our protocol hence attains optimal computation, memory, load-balancing, and locality with polylogarithmic overhead, together with the requirement of a single-use broadcast of $\mathsf{polylog}(n)$ bits per party.

A note on communication complexity: We have not focused on minimizing the communication complexity of our protocol: As with all generic multi-party computation protocols in the information-theoretic setting, we can only (trivially) bound the communication complexity by the computation complexity. In the computational setting, protocols whose communication complexity is smaller than the complexity of the evaluated function have been constructed by relying on fully homomorphic encryption (FHE) (e.g., [Gen09, AJLA$^+$12, MSS13]). We leave as an interesting open question whether FHE-style techniques can be applied also to our protocol to improve the communication complexity based on computational assumptions.

1.2 Technical Overview

We reach our final result via three intermediate steps:

- First, we show how to achieve secure computation of RAM program functionalities Π_j in a reusable, memory-balanced fashion. The resulting protocol achieves the desired total communication and computation complexities $\tilde{O}(n|x| + \sum_{j=1}^{N} Time(\Pi_j) + nN|y|)$, and evenly distributes the total best-case memory requirement (up to polylogarithmic factors) to $\tilde{O}(|x| + \max_j Space(\Pi_j)/n)$ per party.
- Next, we show how to load balance the communication and computation requirements of this protocol, while preserving the original complexities. Our techniques guarantee balancing of resources not only at the conclusion of the protocol, but also at any given point in time during the protocol execution (up to an additive $\mathsf{polylog}(n)$ term).
- Finally, we show how to achieve communication locality $\mathsf{polylog}(n)$ for our protocol (assuming a one-time broadcast per party), while preserving the properties attained in the previous two steps.

In the following three subsections, we expand upon each of these three steps.

[4]Note that we are only claiming that for general secure computation protocols, and even if we restrict to functionalities that only access a few players' inputs, and only a few bits of their data, essentially all players needs to perform computation at least $\Omega(|x|)$, thus a total computation complexity of $\Omega(n|x|)$. To see this, consider secure computation of a "multi-party PIR" functionality: each player $i > 1$ has an input some "big data" x_i, and player 1 has as input a player index i and an index j into their data x_i. The functionality returns $x_i[j]$ (i.e., the j'th bit of player i's data) to player 1 and nothing to everyone else. We claim that each player $i > 1$ must access every bit of x_i; if not, it learns that that particular bit of its data was not requested, which it cannot learn in an ideal execution of the functionality.

1.2.1 Step 1: Scalable Memory-Balanced, Reusable MPC for RAM Programs

Warmup. Before proceeding to describe our solution, let us first see briefly why a straightforward extension of prior work does not suffice. Recall the framework described by Damgård *et al.* [DMN11] for evaluating a RAM program on parties' inputs: Everyone holds secret shares of all parties' inputs, and the parties jointly execute (using standard MPC) the trusted CPU instructions of the *ORAM-compiled*[5] version of the program. The complexities of this protocol scale well with the RAM complexity, and not circuit complexity, of the program. However, the complexities scale quite poorly with the number of parties n: each party must communicate and maintain information of size equivalent to *all parties'* inputs, and everyone must talk to everyone else for every time step of the RAM program evaluation.

Instead, suppose we modify the protocol by first electing a small $\mathsf{polylog}(n)$-size representative committee of parties, and then only performing the above steps within this committee. This immediately drops the total communication and computation of the protocol to desired levels. However, this approach does not save the subset of elected parties from carrying the huge burden of the entire computation. In particular, each elected party must maintain enormous memory storage, equal to the size of all parties' inputs combined. It is not clear how to distribute this memory burden among all parties without new techniques.

Our Approach. Our per-party memory requirement means that each party can only hold information the size of $\mathsf{polylog}(n)$ parties' inputs. At the same time, our computation requirement means that only $\mathsf{polylog}(n)$ parties can compute (or communicate) for each data access made in the program Π. It is clear that ORAM will serve as a useful tool for hiding access patterns to data; however, to achieve our stringent complexity requirements, the *way* in which ORAM is used will be different from previous works.

***Distributed* ORAM.** At a high level, our protocol will emulate a *distributed* ORAM structure, where the CPU and memory cells in the ORAM are *each associated with parties.* More specifically, to bridge the gap between the "honest" setting of ORAM (where the CPU is trusted with secret information, memory cells are assumed to obliviously store information, and all entities faithfully follow the protocol), and the malicious setting of MPC (where all parties are symmetric, and nearly one third are malicious), each role of the ORAM will be associated to a *committee* of parties. The "CPU" committee will control the evaluation flow of the (ORAM-compiled) program, communicating with the appropriate memory cell committee for each address to be accessed in the (ORAM-compiled) database. To ensure honest behavior, committees will be elected so as to guarantee 2/3 honest members; each operation will be performed via a (small-scale) secure multiparty computation within the corresponding committee (or pair of committees); and secret

The distributed ORAM structure will enable us to evenly spread the memory burden across parties, incurring only $\mathsf{polylog}(n)$ overhead in total memory and computation, and while guaranteeing that the communication patterns between committees (corresponding to data access patterns) do not reveal information on the underlying secret values.

This framework shares a similar flavor to the protocols of [DKMS12, BGJK12], which assign committees to each of the gates of a circuit being evaluated, and to [BGT13], which uses CPU

[5]Recall an Oblivious RAM (ORAM) compiler converts any RAM program and associated database to a modified pair, such that data access patterns of the compiled program hide information about the underlying data.

and input committees to direct program execution and distributedly store parties' inputs. The distributed ORAM idea improves and conceptually simplifies the input storage handling of Boyle *et al.* [BGT13], in which n committees holding the n parties' inputs execute a distributed "oblivious input shuffling" procedure to break the link between which committees are communicating and which inputs are being accessed in the computation. Input shuffling provides only weak guarantees, e.g. requiring that data items remain forever in local memory and cannot be reinserted once they are accessed; further, the (somewhat expensive) shuffling procedure must be repeated for each program to be evaluated. In contrast, we wish to provide efficient support for full-fledged RAM computations in a reusable fashion.

However, several challenges arise along this path.

Electing Committees. We must first show how to efficiently generate a large collection of good committees. To do so, we follow an approach similar to [KLST11]: we define committees C_i by the evaluations of an appropriately chosen function F at the corresponding inputs i. We need that each committee must be at least 2/3 honest, and no party can appear in significantly more than his share of committees; looking ahead, our protocol will require this to hold even for *superpolynomially many* committees C_i. We show that if F is randomly sampled from a $\mathsf{polylog}(n)$-wise independent function family, then these properties will hold with overwhelming probability, while only requiring a description of size $\mathsf{polylog}(n)$. We achieve this sampling within the protocol by first electing a *single* good committee, and then using this committee to sample and communicate a random seed s that defines the remaining committees as $C_i := F_s(i)$.

Additionally, we must be careful to use for the first step a committee election protocol that scales well with n. For example, the seemingly light Feige committee election protocol [Fei99] requires memory $\Omega(n)$ per party, which we cannot afford. We instead make use of the almost-everywhere scalable committee election protocol of King *et al.* [KSSV06], which requires only $\mathsf{polylog}(n)$ memory per party in $\mathsf{polylog}(n)$ rounds of communication, together with the single per-party use of a broadcast channel to reach full agreement.[6]

ORAM with Efficient Parallel Insertion. In order to make use of ORAM guarantees, the parties' inputs must first be inserted into the ORAM-compiled database structure. However, read and write operations in existing constructions of secure ORAM compilers are inherently *sequential*: in particular, to write n inputs into the data structure incurs a cost of $\tilde{\Omega}(n)$ rounds of communication. This unfortunate cost will likely dwarf the benefits the ORAM would provide in a non-asymptotic (non-amortized) setting.

To fix this problem, we present a novel technique for securely and efficiently inserting several items into an ORAM structure in *parallel*, within a multiprocessor setting. Our technique builds upon the ORAM compiler presented by Chung and Pass [CP13], which in turn closely follows the tree-based ORAM of Shi *et al.* [SCSL11].[7] The advantages of this procedure extend beyond the scope of multi-party computation, providing an efficient means for converting preexisting databases into one whose access patterns are hidden. We define and achieve this new property within the

[6]Note that while broadcast will enable all to receive a message, our protocol will only require a small portion of parties to listen to and store information related to the message.

[7]Although we have not verified the details, it would seem that our techniques for this "parallel insertion step" will apply also to other tree-based ORAMs, such as [SCSL11, SvDS$^+$13]; relying on [CP13] merely simplifies the analysis at this point. However, to apply Steps 2 and 3 to the resulting protocol to achieve load balancing and communication locality, it will actually be important for us to here rely on the specific [CP13] ORAM.

standard ORAM setting; then, with this new procedure in hand, we show how to emulate the specified computation steps in the setting of MPC.

Distributed ORAM of *Dynamic Size.* The parties' inputs are not the only values that must be stored in memory: in addition, evaluating a program Π has an associated space requirement. During evaluation, Π may make random accesses to both the parties' inputs and to this (potentially large) work tape. In order to maintain efficient memory balancing of the protocol,[8] while still ensuring that access patterns do not reveal unwanted information, we will also store this work tape data in the ORAM structure.

However, this idea introduces another problem. Existing ORAM constructions assume an a priori size bound on the desired database size. Thus, straightforward implementation of this idea will support only RAM programs with a priori bounded space requirements.

To subvert this problem, we instead maintain *two* instantiations of ORAM data structures: One will hold the parties' inputs (and any state information to be maintained between program executions), and will be of a priori fixed size. The second ORAM will consist solely of work tape data, will be erased ("deallocated") at the conclusion of each program Π execution, and will *grow and shrink* for each program Π that comes along, to match the space requirement of Π. We show how to achieve such dynamic-size distributed ORAM, building again upon the ORAM of Chung and Pass [CP13] together with techniques for implicitly electing *superpolynomially many* good committees with short description. Intuitively, ORAM-compiled databases of different sizes correspond to binary trees of memory cells of different depths. Thus, for any program Π with space requirement $\mathit{Space}(\Pi)$ that comes along, we will "activate" committees in the binary tree ORAM structure up to the necessary depth to support database size $\mathit{Space}(\Pi)$. This process can support *any* polynomial $\mathit{Space}(\Pi)$, as we have defined superpolynomially good committees; at the same time, work tape storage will induce only $\mathsf{polylog}(n)$ memory overhead above the space requirement of Π, since "deactivated" ORAM memory cell committees need not store data.

Further, we can also support evaluation of RAM programs *for whom a size bound is unknown.* In such case, the protocol will proceed in the fashion described above, mimicking the scenario of a superpolynomial-size ORAM database structure. Since the work tape begins empty, and only undergoes a polynomial number of accesses, we show that the total memory (and communication) requirements of the protocol will grow only by $\mathsf{polylog}(n)$ per time step of the program Π.

The Combined Protocol 1. We now bring these new tools together into our desired Step 1 protocol, as given in Figure 1.

1.2.2 Step 2: Load Balancing the Protocol

The protocol achieved in Step 1 already achieves a lot of the properties we desire: a) it deals with RAM programs, b) the *total* computation and memory overheads are small, c) the protocol is reusable, and d) the *memory* used by each player is load-balanced. In this step we show how to balance the computation requirements of the protocol (on the way, we will actually balance also communication). Then finally, in Step 3 we will show how to additionally achieve locality without sacrificing any of the other properties.

[8]Note that in [BGT13] a single committee is burdened with the entire work tape memory storage.

(Informal) Protocol 1: Scalable Memory-Balanced, Reusable MPC for RAM Programs
Inputs: Each party P_i holds secret input x_i
Outputs: Evaluates sequence of RAM programs Π_j on $x_1, \ldots, x_n$

Committee Setup

1. Elect a $\mathsf{polylog}(n)$-size "CPU" committee C, using a low-memory protocol (see Section 2.3).
2. Parties in C collectively sample two random seeds s, s^{Work} to define several (slightly super-polynomially many) committees as $C_\ell := F_s(\ell)$ where $\{F\}$ is a $\mathsf{polylog}(n)$-wise independent function family (similar for s^{Work}). (See Section 3.1).
   Each such committee will play the role of a single memory cell in one of two ORAM structures: one holding parties' inputs (and remnant data), and one holding the computation work tape.

Input Commitment and Initialization

1. In parallel, each party P_i verifiably secret shares his input among a corresponding committee C_i (For the case of large inputs, each input block is shared to a separate committee).
2. Parties' inputs are "inserted" into the Input ORAM structure in parallel. This is done by pairwise communications among the n input committees in $\log n$ sequential rounds, as described in Section 3.2.

Computation (Repeated for each program Π_j)

For each RAM program Π to be computed, "activate" memory cell committees in the work tape ORAM up to depth $\log Space(\Pi)$ (or depth $\log^2 n$, if $Space(\Pi) =$"unbounded"). Consider the ORAM-compiled program Π', composed of a sequence of triples $(\Pi_t, \mathsf{Access}_t, \mathsf{Access}_t^{\mathsf{Work}})$. Progressing for $t = 1, 2, \ldots, Time(\Pi')$, perform the following steps:

1. Π_t: The CPU committee C securely executes subprogram Π_t on the current secret CPU state (which begins empty). The output of this computation is a pair of (public) memory addresses $(\mathsf{addr}, \mathsf{addr}^{\mathsf{Work}})$ corresponding to one committee each from the Input and Work Tape ORAMs, and an updated secret CPU state (in particular, including read/write instructions to be implemented at these addresses).
2. Access_t: Committee C communicates with the committee C_{addr} from the Input ORAM and executes the secret read/write instruction via an MPC. As a result, C_{addr} holds a possibly new memory value, and C holds an updated secret CPU state.
3. $\mathsf{Access}_t^{\mathsf{Work}}$: The previous step is mirrored for the Work Tape ORAM, via an MPC between C and committee $C^{\mathsf{Work}}_{\mathsf{addr}^{\mathsf{Work}}}$.

At the conclusion of $Time(\Pi')$ steps, committee C learns the desired output $y := \Pi(x_1, \ldots, x_n)$.

Output Delivery: Disseminate output y to committees $C_1, \ldots, C_{nK}$ for $K \in \omega(\log n)$ (to cover all parties P_i) via a binary tree structure. I.e., C sends y to C_1, C_2; they each send y to two distinct assigned committees; etc.

Figure 1: Informal description of combined Protocol 1, achieving scalable memory-balanced, reusable MPC for RAM programs.

From a load-balancing perspective, the key problem with the protocol resulting from Step 1 is that some parties (in particular, the parties in the CPU committee) are required to do significantly more work than other parties (in fact, they need to perform the entire program computation). We now address this issue and modify our protocol such that the total computational complexity is preserved, while additionally achieving a strong load balancing property—with high probability, at all times throughout the protocol execution, every party performs close to $1/n$ fraction of current total work, up to an additive $\mathsf{polylog}(n)$ amount of work. This will hold simultaneously for both communication and computational complexity.

Job Passing. Our main idea is a very natural one: Recall that in the protocol from Step 1, each committee is responsible for some task or "job" (e.g., implementing a particular memory cell, or the CPU). Whenever a committee has performed "enough work" for a particular job, it selects a new committee and passes on the job to this new committee. Note that each committee of our protocol has $\mathsf{polylog}(n)$ parties and only needs to hold a small $\mathsf{polylog}(n)$-size (secret) state (either the CPU state or the memory content). Thus, one committee C_1 can pass its job to a new committee C_2 by simply "sending the state" to C_2 using a generic MPC protocol that on input the shares of the state from each party of C_1 output fresh shares of the state for each party of C_2, in total incurring a cost of $\mathsf{polylog}(n)$ in computational complexity.

More precisely, at an initial stage of the protocol, we elect a set of *worker* committees, and randomly assign each job of the ORAM program to some worker committee (where the same committee may receive multiple jobs). Then, during the course of the Step 1 protocol execution, we will appropriately switch the worker committee for each job to a fresh random worker committee after they have performed an appropriate amount of work for their job.

When to switch committees. At first sight, it may seem that this idea can be implemented generically, no matter what the underlying protocol is. However, there is a major obstacle with this approach: Every time we switch committees for a job, we introduce a "switching cost," corresponding to work that must be performed in order to complete the switch itself. The danger is that switching incurs a cost not only to the worker committee assigned to the job, but also to other worker committees (in fact, this is the case for our protocol), which may in turn push these committees past their work thresholds and trigger a switching cascade. This potential cascade becomes non-trivial to bound (and also to implement in a distributed way the queue for storing what jobs to switch committees for).

Rather, we here again rely on the particular ORAM implementation, which serves as the basis of the Step 1 protocol execution. For each "job" (i.e., a memory cell node or the CPU in the ORAM), we carefully choose a cost metric dictating when to switch (roughly, the number of times the CPU accesses the node), and instruct committees to switch whenever the cost according to this metric reaches a fixed constant threshold. Note that this cost metric does not at all consider the switching cost, which intuitively avoids the above-mentioned cascading problem. But, we show that for the particular pattern of memory cell accesses induced by the specific underlying ORAM construction, this choice of metric successfully ensures the total computation and communication complexity of each worker committee is balanced, *including the associated costs of switching.* Roughly speaking, this is shown by first noting that we achieve balancing for our cost metric (we show this by abstracting out some properties of "nice" cost metrics for which balancing is obtained by our construction) and next, in a second step, we show that that any sufficiently "related" cost-metric (and both the overall computation and communication costs—including the costs of switching—can be shown to be "related" to our specific cost-metric) also are balanced, concluding that our protocol balances both computation and communication.

But we are not done yet. So far, we have only argued that on a committee level, the work is balanced. As the final step we note that, by a Chernoff bound, each party participates in roughly same number of worker committees and thus the per-party work load is also balanced.

Keeping track of who does what. There is still one final point to be addressed. In the protocol in Stage 1, everybody knew which committee was assigned to which job, and so knew who to communicate with in order to execute a given command. Now, this assignment of worker committees to jobs is dynamically changing. We thus need a way for parties to keep track of the updated information. In particular, the CPU committee must be able to access any memory cell in the ORAM database, but it cannot afford to store by itself an entire record of which committee is currently serving the role of each cell.

We here again rely on the particular tree-based ORAM construction. We show that it suffices for the CPU committee to keep track *only* of which committee is responsible for the root(s) of the ORAM tree(s),[9] and for each committee assigned the job of a memory cell in the ORAM tree to keep track only of which committees are currently responsible for their children memory cells. We emphasize that for our load-balancing analysis to work it is crucial that only a "one-way" link (from a node to its children, and not back) is stored—in particular, the cost of switching worker committees for a node in the tree will induce an update cost to his parent, but not additionally to his children. This introduces a danger of children nodes not knowing who to their parents are, to update when jobs are passed; however, this problem can be overcome for our chosen switching metric and the specific access patterns made by the [CP13] ORAM scheme, in which a child node is only accessed (and thus will only ever job switch) directly proceeding an access to his parent.

1.2.3 Step 3: Achieving Communication Locality

Finally, we modify the protocol of Step 2 to achieve *communication locality* on top of the already-attained protocol complexities. At a high level, this is done by considering a fixed network topology, and dynamically assigning committees to these nodes at the start of the protocol. From this point on, all communications travel *only* through this fixed network, via committee-to-committee message passing. Namely, at each point when a party (or committee) is to send a message to another arbitrary party (or committee), this message is routed along the network to its appropriate destination. Then, as long as this communication network graph is low degree, we can guarantee corresponding communication locality.

However, we must take care that by adding this additional level, we do not destroy the overall complexities or load balancing of the original protocol. To do so, we introduce a new primitive, which may be of independent interest:

Local load-balanced routing networks. Roughly speaking, we need to have a way to ensure that information is passed through the fixed communication network in a load-balanced way; for instance, we cannot route all messages through a particular node. Of course, we can never hope to

[9]For those familiar with existing tree-based ORAM constructions: Since the construction is recursive, recall there will actually be $\log n$ roots.

achieve this if the source and destinations are not (individually) uniform, so we here focus only on a setting where this is the case. Indeed, in the protocol resulting from Step 2, where committees are assigned at random to jobs within the protocol, this will hold (at least empirically). More precisely, we want to ensure that if the source s and the destination t are individually uniform (but may be correlated), then the *expected* number of times a node is on the path between them should be balanced.[10]

We next turn to designing such a low-degree load-balanced routing network. The general idea is as follows. We use a regular expander graph G as our communication network. To route a message from a source s to a destination t, taking inspiration from [VB81], we first take a random walk ("walking into the woods from s") of length $\omega(\log |G|)$ from s, reaching a node s'. Note that at this point, the routing is load-balanced (in expectation, as required): we start at a random node—since the graph is regular that means we are starting at the stationary distribution—and each time we take single random step on this graph we remain at the stationary distribution.

Additionally, by the mixing property of the expander graph, this ensures that we end up in a random node s' in G, *independent of s*. We then take another random walk of length $\omega(\log |G|)$ from s', this time *conditioned* on reaching t. Since as observed above, s' is an (essentially) uniform node, the distribution of the this second walk ("home from the woods") is statistically close to taking a length $\omega(\log |G|)$ random walk from t (i.e., "walking into the woods from t"). It thus follows exactly as before that also this second step is load-balanced in expectation.

Achieving computationally efficient routing. An issue with this approach, however, is that it is not clear how to efficiently implement the "walk back from the woods to t." In particular, recall that this requires sampling a random walk conditioned on reaching t, which generically may be expensive when the size of G is large (in particular, $|G| \in \tilde{\Theta}(n)$ in our setting). We resolve this final issue by relying on a particular regular $\log n$-degree expander—namely the boolean hybercube—for which the conditional random walk can be efficiently found. In fact, for this specific expander we can slightly simplify the routing (and the analysis): we can simply take a *random shortest path* to t.

Ensuring that the complete protocol is load balanced. So far we have only argued that the protocol from Step 2 is load balanced. However, now we have introduced additional "routing" cost in the protocol (i.e., the cost for routing messages in the communication network). It is thus no longer clear that the combined protocol is load balanced. We finally address this issue by noting that the same cost metric balances also the routing cost, and thus we conclude that the total cost is balanced as well.

2 Preliminaries

2.1 RAM and Oblivious RAM

A Random Access Machine (RAM) with database memory size d consists of a CPU with a small number of registers that can each store a string of length $\log d$, and an "external" memory of size

[10]This is a perhaps a seemingly weak load-balancing property in that we only balance the expectation, but it actually suffices for our application, namely achieving "actual" load balancing of the protocol from Step 2. This follows by appealing to standard concentration bounds and noting that due to the "job passing" scheduling used in the protocol, we get access to *multiple independent samples* of source-destination pairs s, t

d. The CPU executes a program Π (given n and some input x) that can access the memory by a $\mathsf{Read}(\mathsf{addr})$ and $\mathsf{Write}(\mathsf{addr}, v)$ operations, where $\mathsf{addr} \in [d]$ is an index to a memory location, and v is a word (of size $\log d$).

The CPU executes a program Π (given d and some input x) that can access the memory by $\mathsf{Read}(r)$ and $\mathsf{Write}(r, v)$ operations, where $r \in [d]$ is an index to a memory location, and v is a word (of size $\log d$). The sequence of memory cell access by such read and write operations is referred to as the *memory access pattern* of $\Pi(d, x)$ and is denoted by $\tilde{\Pi}(d, x)$. (The CPU may also execute "standard" operations on the registers, and may generate outputs).

Roughly speaking, an Oblivious RAM (ORAM) compiler enables executing a RAM program while hiding the access pattern to the memory.

Definition 2.1 ([GO96])**.** A polynomial-time algorithm $ORAM$ is an *oblivious RAM (ORAM) compiler* with computational overhead $\mathsf{ORAM\text{-}Comp}(\cdot)$ and memory overhead $\mathsf{ORAM\text{-}Mem}(\cdot)$ if $ORAM$, given $d \in \mathbb{N}$ and a deterministic RAM program Π with memory size d, outputs a program Π' with memory size $\mathsf{ORAM\text{-}Mem}(d) \cdot d$ such that for any input x, the running time of $\Pi'(d, x)$ is bounded by $\mathsf{ORAM\text{-}Comp}(d, x) \cdot T$, where T is the running time of $\Pi(d, x)$, and there exists a negligible function $\mu(d)$ such that the following properties hold:

- **Correctness:** For any $d \in \mathbb{N}$, RAM program Π, and any string $x \in \{0, 1\}^*$, with probability at least $1 - \mu(d)$, it holds that $\Pi(d, x) = \Pi'(d, x)$.
- **Obliviousness:** For any two programs Π_1, Π_2, any $d \in \mathbb{N}$, and any two inputs $x_1, x_2 \in \{0, 1\}^*$, if $|\Pi_1(d, x_1)| = |\Pi_2(d, x_2)|$, then $\tilde{\Pi}'_1(d, x_1)$ is μ-close to $\tilde{\Pi}'_2(d, x_2)$ in statistical distance, where $\Pi'_1 = ORAM(d, \Pi_1)$ and $\Pi'_2 = ORAM(d, \Pi_2)$.

A Tree-based ORAM construction. In the present work, we build upon a specific ORAM construction to achieve an additional advantageous functionality (namely, the ability to efficiently initialize the ORAM structure with data from a database; see Section 3.2). Concretely, we rely on the ORAM due to Chung and Pass [CP13], which in turn closely follows the tree-based ORAM construction of Shi *et al.* [SCSL11]. (We believe that our techniques developed in Step 1 of our protocol would also apply to the construction of Shi *et al.* [SCSL11] or the further optimized Path ORAM construction of Stefanov *et al.* [SvDS+13]; for this step relying on [CP13] mainly simplifes the analysis. However, when dealing with the locality and load-balancing issue (Steps 2 and 3), our analysis quite crucially appeals to the [CP13] construction.)

We now present the [CP13] construction. The construction proceeds by first presenting an intermediate solution achieving obliviousness, but in which the CPU must maintain a large number of registers (specifically, providing a means for securely storing d data items requiring CPU state size $\tilde{\Theta}(d/\alpha)$, where $\alpha > 1$ is any constant). Then, this solution is recursively applied $\log_\alpha d$ times to store the resulting CPU state, until finally reaching a CPU state size $\mathsf{polylog}(d)$, while only blowing up the computational overhead by a factor $\log_\alpha d$. The overall compiler is fully specified by describing one level of this recursion.

Step 1: Basic ORAM with $O(d)$ registers. The compiler $ORAM$ on input $d \in \mathbb{N}$ and a program Π with memory size d outputs a program Π' that is identical to Π but each $\mathsf{Read}(r)$ or $\mathsf{Write}(r, val)$ is replaced by corresponding commands $\mathsf{ORead}(r)$, $\mathsf{OWrite}(r, val)$ to be specified shortly. Π' has the same registers as Π and additionally has d/α registers used to store a *position map* Pos plus a polylogarithmic number of additional *work* registers used by ORead and OWrite. In its external memory, Π' will maintain a complete binary tree Γ of depth $\ell = \log(d/\alpha)$; we index nodes in the tree by a binary string of length at most ℓ, where the root is indexed by the empty string λ, and each node indexed by γ has left and right children indexed $\gamma 0$ and $\gamma 1$, respectively. Each memory cell r will be associated with a random leaf *pos* in the tree, specified by the position map Pos; as we shall see shortly, the memory cell r will be stored at one of the nodes on the path from the root λ to the leaf *pos*. To ensure that the position map is smaller than the memory size, we assign a *block* of α consecutive memory cells to the same leaf; thus memory cell r corresponding to block $b = \lfloor r/\alpha \rfloor$ will be associated with leaf $pos = \mathsf{Pos}(b)$.

Each node in the tree is associated with a *bucket* which stores (at most) K tuples (b, pos, v), where v is the content of block b and *pos* is the leaf associated with the block b, and $K \in \omega(\log d) \cap \mathsf{polylog}(d)$ is a parameter that will determine the security of the ORAM (thus each bucket stores $K(\alpha + 2)$ words). We assume that all registers and memory cells are initialized with a special symbol $\perp$.

The following is a specification of the $\mathsf{ORead}(r)$ procedure:

Fetch: Let $b = \lfloor r/\alpha \rfloor$ be the block containing memory cell r (in the original database), and let $i = r \mod \alpha$ be r's component within the block b. We first look up the position of the block b using the position map: $pos = \mathsf{Pos}(b)$; if $\mathsf{Pos}(b) = \perp$, set $pos \leftarrow [n/\alpha]$ to be a uniformly random leaf.

Next, traverse the data tree from the root to the leaf *pos*, making exactly one read and one write operation for the memory bucket associated with each of the nodes along the path. More precisely, we read the content once, and then we either write it back (unchanged), or we simply "erase it" (writing $\perp$) so as to implement the following task: search for a tuple of the form (b, pos, v) for the desired b, pos in any of the nodes during the traversal; if such a tuple is found, remove it from its place in the tree and set v to the found value, and otherwise take $v = \perp$. Finally, return the ith component of v as the output of the $\mathsf{ORead}(r)$ operation.

Update Position Map: Pick a uniformly random leak $pos' \leftarrow [n/\alpha]$ and let $\mathsf{Pos}(b) = pos'$.

Put Back: Add the tuple (b, pos', v) to the root λ of the tree. If there is not enough space left in the bucket, abort outputting $\mathsf{overflow}$.

Flush: Pick a uniformly random leaf $pos^* \leftarrow [n/\alpha]$ and traverse the tree from the roof to the leaf pos^*, making exactly one read and one write operation for every memory cell associated with the nodes along the path so as to implement the following task: "push down" each tuple (b'', pos'', v'') read in the nodes traversed so far as possible along the path to pos^* while ensuring that the tuple is still on the path to its associated leaf pos'' (that is, the tuple ends up in the node γ = longest common prefix of pos'' and pos^*.) Note that this operation can be performed trivially as long as the CPU has sufficiently many work registers to load two whole buckets into memory; since the bucket size is polylogarithmic, this is possible. If at any point some bucket is about to overflow, abort outputting $\mathsf{overflow}$.

$\mathsf{OWrite}(r, v)$ proceeds identically in the same steps as $\mathsf{ORead}(r)$, except that in the "Put Back" steps, we add the tuple (b, pos', v'), where v' is the string v but the ith component is set to v (instead of adding the tuple (b, pos', v) as in ORead). (Note that, just as ORead, OWrite also outputs the ordinal memory content of the memory cell r; this feature will be useful in the "full-fledged" construction.)

Theorem 2.2 ([CP13]). *The compiler ORAM described above is a secure Oblivious RAM compiler with* $\mathsf{polylog}(d)$ *worst-case computation overhead and* $\omega(\log d)$ *memory overhead, where d is the database memory size.*

2.2 t-Wise Independent Function Families

In our protocol, we make use of function families with good combinatorial properties in order to elect a large number of small "good" committees of parties with short description size. More specifically, a short seed s will implicitly define superpolynomially many committees of the form $C_i := F_s(i)$. We require that the resulting committees are sufficiently "random," so that each committee will have an appropriate fraction of honest parties, and no individual party appears in too many committees.

In this section, we include some useful definitions and theorems of t-wise and almost t-wise function families, which will be used to achieve this goal in Section 3.1.

Definition 2.3. We say that a family of functions $\{F_s : D \to R\}_{s \in S}$ is *t-wise independent* if for all $y_1, \ldots, y_t \in R$ and distinct $x_1, \ldots, x_t \in D$ it holds that

$$\Pr_{s \leftarrow S} \Pr[F_s(x_1) = y_1 \wedge \cdots \wedge F_s(x_t) = y_t)] = \frac{1}{|R|^t}.$$

Lemma 2.4. *For any D, R, there exists a t-wise independent function family $\{F_s : D \to R\}$ with seed size $|s| = t(\log D + \log R)$ bits.*

Definition 2.5. Let $X_1, \ldots, X_N$ be random variables taking values in the interval $[0, 1]$. We say that $X_1, \ldots, X_N$ are *t-wise independent* if for any $a_1, \ldots, a_t \in [0, 1]$, and any distinct indices $1 \le i_1, \ldots, i_t \le N$, it is the case that

$$\Pr[X_{i_1} = a_1, \ldots, X_{i_t} = a_t] = \prod_{j=1}^{t} \Pr[X_{i_j} = a_j].$$

Bellare and Rompel [BR94] proved the following Chernoff-like tail inequality for the sum of t-wise independent random variables.

Theorem 2.6. ([BR94, Lemma 2.3]) Let $t \ge 4$ be an even integer. Suppose $X_1, \ldots, X_N$ are t-wise independent random variables taking values in $[0, 1]$. Let $X = X_1 + \cdots X_N$ and $\mu = E[X]$, and let $A > 0$. Then

$$\Pr[|X - \mu| \ge A] \le C_t \cdot \left(\frac{t\mu + t^2}{A^2}\right)^{t/2},$$

where $C_t = 2\sqrt{\pi t} \cdot e^{1/6t} \cdot (5/2e)^{t/2} \le 8$ (for $t \ge 4$).

Looking ahead, we will use a function family $\{F_s : \{0,1\}^{\log^2 n} \to [n]^K\}$ to elect slightly superpolynomially many committees $C_i := F_s(i)$ of approximate size $K \in \omega(\log^2 n) \cap \mathsf{polylog}(n)$ from $[n]$ parties (approximately size K, as multiset repetitions are removed). To guarantee with overwhelming probability that each committee has sufficiently many honest parties, and that no party appears in too many committees, it will suffice that $\{F_s\}$ is *K-wise independent.* A succinct description of such a collection of committees can thus be generated using a random seed s of length $K(\log^2 n + \log |[n]^K|) = O(K^2 \log^2 n)$.

2.3 Low-Memory Committee Election + Coin Tossing Protocol

Our protocol uses as a subroutine an n-party protocol for electing a single "good" committee of size $\mathsf{polylog}(n)$, and collectively generating a random string. For our purposes, we wish this to take place within $\mathsf{polylog}(n)$ memory per party. This can be achieved by combining the memory-efficient *almost-everywhere* scalable leader election protocol of King *et al.* [KSSV06], together with simple techniques presented by Boyle *et al.* [BGT13] building atop the [KSSV06] protocol which can enable *almost-everywhere coin tossing*, together with a polling step consisting of a single broadcast per party to reach full agreement.

Namely, we make use of the following theorem. We defer the discussion and proof of this result to Appendix A.

Theorem 2.7 (Almost-Everywhere Committee Election + Coin Tossing). *Suppose there are n processors, at least $(2/3 + \epsilon)$ are honest for constant $\epsilon > 0$. Then for any $k \in \mathsf{polylog}(n)$, and $K \in \mathsf{polylog}(n) \cap \omega(\log n)$, there exists an algorithm that, given access to a single broadcast of $\mathsf{polylog}(n)$ bits per party, outputs with overwhelming probability (in n) a string $s \in \{0,1\}^k$ and a committee $C \subset [n]$ of size $|C| \in \Theta(K)$, satisfying the following properties:*

- $1 - o(1)$ *fraction of honest processors agree on s and C.*
- *The distribution of s over the random coins of the honest parties is statistically close to uniform over $\{0,1\}^k$ (i.e., statistical distance negligible in n).*
- $|C \cap H|/|C| \geq (2/3 + \epsilon/2)$*, where $H \subset [n]$ denotes honest processors.*
- *Every honest processor sends and processes only $\mathsf{polylog}(n)$ bits.*
- *The number of rounds of communication is $\mathsf{polylog}(n)$.*

Proof. Builds upon [KSSV06]. See Appendix A. □

3 Protocol 1: Scalable Memory-Balanced, Reusable MPC for RAM Programs

In this section, we present our first intermediate protocol, achieving scalable memory-balanced, reusable MPC for RAM programs. In the two sections that follow this, we show how to adapt this protocol to further achieve load balancing and communication locality.

Recall we consider a model where parties wish to evaluate a sequence of RAM programs Π on their inputs and additional maintained state from computation to computation. We assume an a priori specified size on this maintained memory. For simplicity of notation, we assume this size is $\tilde{O}(n \cdot |x|)$ (corresponding to the collective size of all parties inputs, which is anyway maintained between computations), and thus absorb this cost within the analysis; however, this can be extended to any desired polynomial in n. Each queried program Π is paired with an optional space bound space on the size of working memory required to evaluate Π. This value may be selected as "$\mathsf{unbounded}$", in which case the protocol must be able to support a growing work tape of unbounded polynomial size (while maintaining memory balancing). In such case, we define the work tape size to be the runtime $|\Pi| := Time(\Pi)$ of the program Π (as the required working memory may grow with each step of computation).

See Figure 2 for a formal description of the ideal functionality $\mathcal{F}_{\mathsf{Dyn}}$ we wish to achieve.

Ideal Functionality $\mathcal{F}_{\mathsf{Dyn}}$

$\mathcal{F}_{\mathsf{Dyn}}$ running with parties $P_1, \ldots, P_n$ and an adversary, proceeds as follows. The functionality maintains longterm storage of parties' inputs $\{x_i\}_{i \in [n]}$ and state information state transferred from computation to computation.

1. Initialize $\mathsf{state} \leftarrow \emptyset$.
2. Input commitment: Upon receiving an input $(\mathit{commit}, \mathit{sid}, \mathit{input}, x_i)$ from party P_i, record the value x_i as the input of P_i.
3. Computation: Upon receiving a tuple $(\mathit{compute}, \mathit{sid}, \Pi, \mathsf{space}, \mathsf{time})$ consisting of a RAM program Π, a space bound space (which may be $\mathsf{unbounded}$), and a time bound time, execute Π as $(\mathsf{output}, \mathsf{state}) \leftarrow \Pi(x_1, \cdots, x_n, \mathsf{state})$ with the current value of state, for time time steps, and with work tape limited to size space. Output the resulting value output to all parties.

Figure 2: The ideal functionality $\mathcal{F}_{\mathsf{Dyn}}$, corresponding to secure computation of dynamic random-access functionalities.

Theorem 3.1 (MPC for Dynamic Functionalities). *For any constant $\epsilon > 0$, there exists an n-party statistically secure (with error negligible in n) protocol that UC-realizes the functionality $\mathcal{F}_{\mathsf{Dyn}}$ for securely computing a sequence of RAM programs Π_j, as described in Figure 2, handling $(1/3 - \epsilon)$ static corruptions, and with the following complexities:*

Per-party Memory	$\tilde{O}(\lvert x\rvert + \max \mathsf{space}/n + \sum_j \lvert y_j\rvert)$
Rounds	$\mathsf{BC}+\ \tilde{O}(\sum_j \lvert\Pi_j\rvert)$
Per-party CC/comp	$\mathsf{BC}+\ \tilde{O}\left(\lvert x\rvert + \sum(\lvert\Pi_j\rvert + \lvert y_j\rvert)\right)$
Total *CC/comp*	$n\mathsf{BC}+\ \tilde{O}\big(n\lvert x\rvert + \sum_j(\lvert\Pi_j\rvert + n\lvert y_j\rvert)\big)$

Here, BC denotes one execution of a broadcast channel (or protocol) to send $\mathsf{polylog}(n)$ bits, $|x|$ denotes input size, $\sum |\Pi|$ denotes the combined (worst-case) runtime of the queried RAM programs Π, $|y|$ denotes the output size of the corresponding programs Π, and $\max \mathsf{space}$ denotes the maximum value of space over queries (Π, space). Asymptotic notation is with respect to the number of parties n.

Overview of the section. As discussed in the introduction, the backbone of our protocol will be to implement a *distributed ORAM* structure, where the CPU and memory nodes in the ORAM are each associated with committees of parties. In the following subsections, we address the challenges that arise along this path, and then combine the corresponding solutions into a complete protocol.

- In Section 3.1, we describe how to elect a large collection of "good" $\mathsf{polylog}(n)$-size committees, which will be assigned to the different roles of the distributed ORAM structure. More specifically, we show how in $\mathsf{polylog}(n)$ rounds, and $\mathsf{polylog}(n)$ bits of communication per party, plus a one-time per-party broadcast (of $\mathsf{polylog}(n)$ bits), all parties will agree on slightly super-polynomially many committees, such that, with overwhelming probability, each committee has 2/3 honest majority, and no party appears in significantly more than his fair share of committees.
- In Section 3.2, we present an ORAM compiler that supports efficient parallel initialization. That is, for any $m \leq d$ (where d is the supported database size), given L parallel processors,

one can insert L data items into an empty ORAM database in $\mathsf{polylog}(d)$ sequential rounds of computation (instead of requiring $\Omega(L)$ rounds). The analysis in this section is within the standard ORAM adversarial model, and has applications outside the realm of MPC.

- In Section 3.3, we shift focus back to the secure multi-party computation setting; we present protocols Init and $\mathsf{Compute}$ that dictate how parties collectively generate the distributed ORAM structure, efficiently insert their secret inputs into this structure, and then execute a sequence of desired RAM program computations. In particular, in this section, we describe how to implement a distributed ORAM of *dynamic size*, which is needed to efficiently support RAM programs of varying memory size requirements.
- Then, in Section 3.4, we bring all the pieces together, and present the complete Part I protocol (proving Theorem 3.1).

3.1 Committee Setup

In this section, we present a protocol for electing and reaching consensus on a collection of committees:

- C: CPU committee
- $\{C_j\}$: Input ORAM memory cells.
- $\{C_j^{\mathsf{Work}}\}$: Work Tape ORAM memory cells.

We want the property that each committee is "good," in the sense that the fraction of honest parties is greater than $\frac{2}{3}$. To ensure the memory requirements are sufficiently balanced, we do not want any party to appear in too many committees. And, in order to use the committee structure for communicating evaluation outputs to all parties, we will want that each party appears in at least one committee.

We follow a similar general approach to that of King *et al.* [KLST11], in which different committees C_i are defined by evaluations of a function with good combinatorial properties at corresponding inputs i. In our case, the function will be sampled from a $\mathsf{polylog}(n)$-wise independent function family. At a high level, our committee setup protocol proceeds as follows.

First, all parties execute an almost-everywhere coin tossing protocol to achieve almost-everywhere agreement on a single "good" committee C, together with two random $\mathsf{polylog}(n)$-bit strings s, s^{Work}. This can be done, e.g., by building atop the Scalable Leader Election protocol of King *et al.* [KSSV06], as shown in Section A.2 (Lemma A.4). Then, each party broadcasts his vote for the values of (C, s, s'), and listens to the incoming votes of a random subset of $\mathsf{polylog}(n)$ parties (ignoring the rest), taking the majority of these values as his final output (C, s, s'). The combination of these steps yields full agreement on a "good" committee C together with random strings s, s^{Work}, as shown in Corollary A.5.

The random string s (similarly, s^{Work}) will be interpreted as a seed for an almost K-wise independent hash function family $\{F_s : \{0,1\}^{\log^2 n} \to [n]^K\}$ which maps $(\log^2 n)$-bit strings ("committee ids") to multisets over $[n]$ of polylogarithmic size.[11] See Section 2.2 for a discussion on t-wise independent function families. The seed s thus implicitly defines $n^{\log n}$ committees as $C_j := F_s(j)$, where repetitions in the resulting multiset are removed. As shown in Section 2.2, such a function family can be described with seed length $|s| = K^2 \log^2 n \in \mathsf{polylog}(n)$ bits. As we show in this section, the K-wise independence of the function family guarantees that the resulting committees will

[11] Considering multisets will aid in analysis; however, we will ultimately interpret these values as proper sets.

be sufficiently random that they satisfy the desired "good" properties, but at the same time admit a concise description by communicating only the seed s (as opposed to sampling and communicating truly random committees).

Similarly, the second string s^{Work} implicitly defines committees $C_\ell^{\mathsf{Work}} := F_{s^{\mathsf{Work}}}(\ell)$.

We present the committee setup protocol $\mathsf{ComSetup}$ in its entirety in Figure 3. The complexities of the different steps of $\mathsf{ComSetup}$ are given in the following table. Unless otherwise stated, listed complexities are (worst case) per party. We denote by the symbol '–' in the Total CC column that it is simply a factor of n greater than the (per-party) CC value.

	Memory	Rounds	CC	*Total* CC
(A.e.)-Elect C, s, s^{Work}	$\tilde{O}(1)$	$\tilde{O}(1)$	$\tilde{O}(1)$	–
Agreement on C, s, s^{Work}	$\tilde{O}(1)$	$O(1)$	$\mathsf{BC} + \tilde{O}(1)$	–
Total $\mathsf{ComSetup}$	$\tilde{O}(1)$	$\tilde{O}(1)$	$\mathsf{BC} + \tilde{O}(1)$	$n\mathsf{BC} + \tilde{O}(n)$

Table 1: Per-party memory and communication complexities of $\mathsf{ComSetup}$. BC indicates one execution of the broadcast channel, and '–' denotes n times the corresponding per-party complexity.

Committee Setup Protocol $\mathsf{ComSetup}$

Fix a value $K \in \omega(\log^2 n) \cap \mathsf{polylog}(n)$, and polynomial $N = N(n)$. Let $\{F_s : \{0,1\}^{\log^2 n} \to [n]^K\}_{s \in S}$ be a K-wise independent function family (with $S = \{0,1\}^{K^2 \log^2 n}$).

Input: $\emptyset$.

Output: All parties: committees C, $\{C_j\}_{j \in [N]}, \{C_\ell^{\mathsf{Work}}\}_{\ell \in [n^{\log n}]}$.

Performed by each party P_i:

1. Achieve full agreement on CPU committee C (of approximate size K) and random strings $s, s^{\mathsf{Work}} \leftarrow \{0,1\}^{K^2 \log^2 n}$ via the protocol $\mathsf{Elect{+}CoinToss}$, as in Corollary A.5.

   The following committees are implicitly defined:

   - Input ORAM: $C_j = F_s(j) \in [n]^K$ (removing multiset repetitions).
   - Work Tape ORAM: $C_\ell^{\mathsf{Work}} = F_{s^{\mathsf{Work}}}(\ell) \in [n]^K$ (removing multiset repetitions).

Figure 3: Committee setup protocol. Executed once by all parties at the beginning of the MPC protocol.

Lemma 3.2 (Committee Setup)**.** *Suppose access to broadcast channel and a perfectly secure MPC protocol (e.g., [BGW88]). Let $K \in \omega(\log n^2) \cap \mathsf{polylog}(n)$ be fixed. Then the n-party protocol* $\mathsf{ComSetup}$ *satisfies the following. For any constants $\epsilon, \delta > 0$, any $(\frac{1}{3} - \epsilon)$ fraction of static malicious corruptions $M \subset [n]$, and polynomial $N = N(n) \leq nK$, it holds with all but negligible probability $e^{-\Omega(K)}$ that at the conclusion of* $\mathsf{ComSetup}$*, all parties agree on committees $C, \{C_j\}_{j \in [N]}, \{C_\ell^{\mathsf{Work}}\}_{\ell \in [n^{\log n}]}$ such that:*

1. It holds that

$$\frac{|C \cap M|}{|C|} \leq \frac{1}{3} - \frac{\epsilon}{2}.$$

2. For every $j \in [N]$ and $\ell \in [n^{\log n}]$,

$$\frac{|C_j \cap M|}{|C_j|}, \frac{|C_\ell^{\mathsf{Work}} \cap M|}{|C_\ell^{\mathsf{Work}}|} \leq \frac{1}{3} - \frac{\epsilon}{2}.$$

3. No party appears in an unfair share of committees. That is, for every $N \leq T \leq n^{\log n}$, and constant $\zeta > 0$, no one appears in more than $(1+\zeta)$ times or fewer than $(1-\zeta)$ times the expected number from the first $(N+T)$ committees $\{C_j\}_{j\in[N]} \cup \{C_\ell^{\mathsf{Work}}\}_{\ell\in[T]}$. Explicitly, for every party $P_i \in [n]$,

$$(1-\zeta)\mu \leq \left|\{j \in [N] : P_i \in C_j\}\right| + \left|\{\ell \in [T] : P_i \in C_\ell^{\mathsf{Work}}\}\right| \leq (1+\zeta)\mu.$$

where $\mu := \left(1 - (1-1/n)^K\right)(N+T)$ is the expected ("fair share") number of committees in which each party occurs.

Proof. Property (1): Follows directly by the properties of the Elect committee election protocol (Corollary A.2).

Property (2): At a high level, this holds since each committee is *individually* random. (Indeed, outside of this the K-wise independence of $\{F_s\}$ will not be used in this step.) By the correctness of the underlying UC-secure information theoretic MPC protocol (e.g., [BGW88]), the output distribution of the sampled seeds s, s^{Work} are each within statistical distance $\nu(k)$ to the uniform distribution $U_{K^2 \log^2 n}$. Thus, it suffices to analyze the properties of the corresponding committees assuming s, s^{Work} are *truly uniform.* In particular, since $\{F_s\}$ is a K-wise independent function family, for any single input x the distribution of $F_s(x)$ over $s \leftarrow S$ is a uniform element of the range, $[n]^K$.

Consider one randomly selected committee $\mathcal{E} \leftarrow [n]^K$. We first lower bound the number of honest parties appearing in $\mathcal{E}$, *with* repeats (i.e., as a multiset instead of set). For each index $t \in [K]$, let X_t be the indicator variable that the tth coordinate of $\mathcal{E} \in [n]^K$ falls within the subset of *honest* parties $[n] \setminus M$. Note that since $\mathcal{E} \leftarrow [n]^K$ is selected at random, $X_1, \ldots, X_K$ are independent random variables, each with expected value $\frac{n-|M|}{n} = (\frac{2}{3} + \epsilon)$. Then by a Chernoff bound,[12] for $\delta = \frac{\epsilon/4}{2/3+\epsilon}$,

$$\Pr\left[\sum_{t=1}^{K} X_t < (1-\delta) \cdot K\Big(\frac{2}{3} + \epsilon\Big)\right] < e^{-K(\frac{2}{3}+\epsilon)\delta^2/2}.$$

That is, denoting by $\mathcal{E}_t$ the tth component of $\mathcal{E} \in [n]^K$, and plugging in δ, we have

$$\Pr\left[\left|\{t : \mathcal{E}_t \notin M\}\right| < K\Big(\frac{2}{3} + \frac{3\epsilon}{4}\Big)\right] < e^{-K\frac{\epsilon^2}{32(2/3+\epsilon)}} \in e^{-\Omega(K)}.$$

Now, suppose it is the case that at least $(\frac{2}{3} + \frac{3\epsilon}{4})$ fraction of components of $\mathcal{E}$ (as a multiset) are from the set of honest parties as above. We now argue that with high probability most of the honest parties appearing will be *distinct.* In order for the number of distinct honest parties in $\mathcal{E}$ to be less than $(\frac{2}{3} + \frac{\epsilon}{2})$, it must be that $\frac{\epsilon}{4}K$ of the components $t \in K$ were parties in $[n]$ that appeared

[12]Explicit Chernoff bound used: For $X = X_1 + \cdots + X_K$ with mean μ, and any $\delta > 0$, it holds that $\Pr[X < (1-\delta)\mu] < e^{-\mu\delta^2/2}$.

already in $\mathcal{E}$. Since the total number of distinct parties appearing in $\mathcal{E}$ (which we denote by $|\mathcal{E}|$), and in particular in $\mathcal{E}_{[K]\setminus I}$ for any subset of indices $I \subset [K]$, is no greater than K, the probability of this event is bounded above by

$$\begin{aligned}
\Pr_{\mathcal{E}\leftarrow[n]^K}\left[|\mathcal{E}| \le K - \frac{\epsilon}{4}K\right] &\le \Pr_{\mathcal{E}\leftarrow[n]^K}\left[\exists I \subset [K] \text{ s.t. } |I| = \frac{\epsilon}{4}K \text{ and } \mathcal{E}_I \subset \mathcal{E}_{[K]\setminus I}\right] \\
&\le \binom{K}{\epsilon K/4} \cdot \Pr_{\mathcal{E}_I \leftarrow [n]^{\epsilon K/4}}\left[\mathcal{E}_I \subset \mathcal{E}_{[K]\setminus I}, \text{ for fixed } I\right] \\
&\le \binom{K}{\epsilon K/4} \cdot \left(\frac{K}{n}\right)^{\epsilon K/4} \le \left(\frac{K\cdot e}{\epsilon K/4}\right)^{\epsilon K/4} \cdot \left(\frac{K}{n}\right)^{\epsilon K/4} \\
&= \left(\frac{4K\cdot e}{\epsilon n}\right)^{\epsilon K/4} < e^{-\omega(K)}.
\end{aligned}$$

In particular, this means that with overwhelming probability, no more than $\frac{\epsilon}{4}K$ *honest* parties appearing in $\mathcal{E}$ are repeated. Together, these two pieces yield

$$\begin{aligned}
\Pr\left[\left|\mathcal{E}\setminus M\right| < K\Big(\frac{2}{3}+\frac{\epsilon}{2}\Big)\right] &\le \Pr\left[\left|\{t : \mathcal{E}_t \notin M\}\right| < K\Big(\frac{2}{3}+\frac{3\epsilon}{4}\Big)\right] + \Pr\left[|\mathcal{E}| \le K - \frac{\epsilon}{4}K\right] \\
&\le e^{-\Omega(K)} + e^{-\omega(K)} = e^{-\Omega(K)}.
\end{aligned}$$

Finally, since $|\mathcal{E}| \le K$, this immediately implies that $\frac{|\mathcal{E}\setminus M|}{|\mathcal{E}|} < \frac{2}{3}+\frac{\epsilon}{2}$ with probability bounded by $e^{-\Omega(K)}$.

Now, this argument was for a single randomly chosen committee $\mathcal{E} \leftarrow [n]^K$. By a simple union bound over the $(N + 2^{\log^2 n})$ total committees $\{C_j\}_{j\in[N]}, \{C_\ell^{\mathsf{Work}}\}_{\ell\in[n^{\log n}]}$ considered, we reach the following upper bound on the probability that *any* committee does not satisfy the desired property. Note here that we must be careful with our setting of parameter K, as we are taking a union bound over a slightly superpolynomial number $2^{\log^2 n}$ of total committees.

$$\begin{aligned}
\Pr\left[\exists \mathcal{E} \in \{C_j\}_{j\in[K]} \cup \{C_\ell^{\mathsf{Work}}\}_{\ell\in[2^{\log^2 n}]} \text{ s.t. } \frac{|\mathcal{E}\setminus M|}{|\mathcal{E}|} < \Big(\frac{2}{3}+\frac{\epsilon}{2}\Big)\right] &\le (N + 2^{\log^2 n}) \cdot e^{-\Omega(K)} \\
&\le e^{-\Omega(K)+\log^2 n} \le e^{-\Omega(K)},
\end{aligned}$$

since $K \in \omega(\log^2 n)$, implying an overall probability of error negligible in n.

Property (3): We now bound the probability that any party appears in too many or too few committees. This is where we will use the K-wise independence of the function F. Consider first the collection of committees $\{C_j\}_{j\in[N]}$. An identical argument will hold for each $\{C_\ell^{\mathsf{Work}}\}_{\ell\in[T]}$, $T \ge N$. (Note, in fact, our bound actually becomes better as the number of committees increases). Fix an arbitrary party $P_i \in [n]$, and define $Y_1,\dots,Y_N$ to be the indicator variables such that $Y_j = 1$ when party P_i appears in committee C_j. By the K-wise independence of the function family $\{F_s\}$, it holds that Y_j are K-wise independent random variables, each individually equal to 1 with the same probability that P_i appears in a set randomly generated as $\mathcal{E} \leftarrow [n]^K$ with multiset repetitions removed; i.e., with probability $p := (1-(1-1/n)^K)$.

Let $Y = Y_1 + \cdots + Y_N$. Note that the mean of Y is $\mu = pN = (1-(1-1/n)^K)N$. Then from a tail inequality for K-wise independent random variables (Theorem 2.6), for $\zeta > 0$,

$$\Pr\left[\left|Y-\mu\right| \ge \zeta\mu\right] \le C_t \cdot \left(\frac{K\mu + K^2}{(\zeta\mu)^2}\right)^{K/2},$$

where $C_t \leq 8$. Plugging in $\mu = pN$, and since $N \geq nK$ and $\zeta > 0$ is a fixed constant, we have

$$\Pr\left[\left|Y - pN\right| \geq \zeta pN\right] \leq \left(\frac{K[pN] + K^2}{(\zeta[pN])^2}\right)^{K/2} \leq \left(\frac{K}{\zeta^2 pN} + \frac{K^2}{(\zeta pN)^2}\right)^{K/2} \leq \left(\frac{1}{\zeta^2 pn} + \frac{1}{(\zeta pn)^2}\right)^{K/2}$$
$$\leq 2^{-K/2} \in e^{-\Omega(K)},$$

for sufficiently large n.[13]

This shows that for a single party P_i, it is unlikely that P_i appears in more than $(1+\zeta)$ times or fewer than $(1-\zeta)$ times the expected number of the N committees $\{C_j\}_{j\in[N]}$. By a straightforward union bound, the probability that this occurs for *any* party P_i is bounded above by $n \cdot e^{-\Omega(K)} = e^{-\Omega(K)}$.

Now, consider the committees $\{C_\ell^{\mathsf{Work}}\}_{\ell\in[T]}$. Since we assumed $T \geq N$, then the exact calculations from above (with N replaced by T) imply that for any single party P_i,

$$\Pr\left[|Y - pT| \geq \zeta pT\right] \leq \cdots \leq \left(\frac{K}{\zeta^2 pT} + \frac{K^2}{(\zeta pT)^2}\right)^{K/2} \leq e^{-\Omega(K)},$$

and so a union bound implies the probability that *any* party P_i appears in more than $(1+\zeta)$ times or fewer than $(1-\zeta)$ times his fair share of the T committees $\{C_\ell^{\mathsf{Work}}\}_{\ell\in[T]}$ is bounded above by $n \cdot e^{-\Omega(K)} \in e^{-\Omega(K)}$. And finally, taking a union bound over the $2^{\log^2 n}$ total possible values T, we have a total error probability $2^{\log^2 n} e^{-\Omega(K)} \in e^{-\Omega(K)}$, as desired. (Recall that $K \in \omega(\log^2 n)$).

$\square$

3.2 ORAM with Efficient Parallel Insertion

In this section, we present a novel technique for inserting several data items into an Oblivious RAM (ORAM) structure in *parallel*, within a multiprocessor setting. This feature is greatly advantageous in converting a preexisting database into one whose access patterns are hidden. The challenge is in doing this *without leaking information* about the resulting secret ORAM state.

Definition 3.3 (ORAM with Parallel Initialization)**.** An Oblivious RAM compiler $ORAM$ (as in Definition 2.1) is said to additionally have *parallel initialization complexity* $\mathsf{ORAM\text{-}Init}(\cdot)$ if for the special initialization RAM program Π_{ins} that simply writes $L \leq d$ new data items to memory to an empty data structure, it holds that the compiled program $\Pi'_{\mathsf{ins}} \leftarrow ORAM(\Pi_{\mathsf{ins}})$ requires parallel time complexity $\mathsf{ORAM\text{-}Init}(L)$ (given L distinct CPUs).

We present a construction of an ORAM compiler with $\tilde{O}(1)$ parallel initialization complexity. Namely, in a setting with L CPUs, we can simultaneously insert L data items into the ORAM structure in $\tilde{O}(1)$ rounds and $\tilde{O}(L)$ computation. (In contrast, existing schemes require $\tilde{\Omega}(L)$ rounds). At the same time, our construction maintains $\tilde{O}(1)$ worst-case memory overhead.

Theorem 3.4 (ORAM with Efficient Parallel Initialization)**.** *There exists an ORAM compiler with the following properties:*

- *Memory overhead:* $\mathsf{ORAM\text{-}Mem}(d) = \tilde{O}(1)$.

[13] Recall that $p = (1 - (1 - 1/n)^K)$, so that $pn \to \infty$. Indeed, the value pn corresponds to the expected number out of n committees that a single party will appear, which corresponds to the size of each committee, roughly K.

- *Computation overhead:* $\mathsf{ORAM\text{-}Comp}(d) = \tilde{O}(1)$.
- *Parallel initialization complexity:* $\mathsf{ORAM\text{-}Init}(d) = \tilde{O}(1)$.

For simplicity of analysis, we rely concretely on the ORAM construction given by Chung and Pass [CP13]; this construction achieves near-optimal ORAM parameters (up to polylogarithmic factors), and permits a particularly straightforward description.[14] However, we believe that our parallel insertion framework can be extended also to a variety of existing ORAM constructions that follow a similar tree-based approach originally due to Shi, Chan, Stefanov, and Li [SCSL11].

ORAM Construction of [CP13]. We here recall a high-level description of the [CP13] ORAM compiler; a more complete specification is given in Section 2.1. Recall that the [CP13] ORAM construction follows a recursive structure, decreasing the CPU secret state size by a factor of $\alpha > 1$ (a parameter taken here to be constant) in each iteration.

In each level of recursion, to securely store d secret database items, the data is split into blocks of size α, which are stored at the nodes of a binary tree with d/α leaves (where each node can store up to a polylogarithmic number of data blocks). Each block is associated with a random leaf in the tree, and appears in some node along the path from the root to this leaf in the tree. The position map $\mathsf{Pos}(i)$ from data blocks i to associated leaf nodes is of size d/α, and is maintained as part of the secret CPU state.

Secure read and write operations are constructed so as to maintain the following invariants, which are shown in [CP13] to be sufficient conditions for security of the ORAM to hold:

1. Each data block i appears in some node in the binary tree along the path from the root to the leaf node $\mathsf{Pos}(i)$ (where $\mathsf{Pos}(i)$ is the currently stored value in the CPU state).
2. None of the nodes "overflow" their allotted space: i.e., with overwhelming probability, we do not reach a state where more than K data blocks are assigned to any particular node of the binary tree, where $K \in \mathsf{polylog}(d)$ is an a priori fixed bucket size.
3. Given the sequence of all prior access patterns Access, the vector of values $(\mathsf{Pos}(1), \dots, \mathsf{Pos}(d/\alpha))$ is uniformly distributed.

See Section 2.1 for the description of how these invariants are maintained by [CP13]. Very roughly, data is accessed in memory block i by looking up $\mathsf{Pos}(i)$, accessing all nodes along the path from the root to leaf $\mathsf{Pos}(i)$, deleting the desired block i from its found location, writing it up at the root node, and then resampling a fresh value $\mathsf{Pos}(i) \leftarrow [d/\alpha]$ at random. To ensure Property 2, after each access a "flush" step is performed, in which a random path down the tree is selected, and all data items along this path are pushed to the lowest point at which their associated $\mathsf{Pos}(i)$ values agree with the selected random leaf node.[15] All these steps preserve Property (1), and since $\mathsf{Pos}(i)$ is freshly resampled after each process that depends on its value, Property (3) holds as well.

We remark, however, that their data access procedure (as with general secure ORAM constructions) is inherently sequential: In particular, to write L data blocks into the ORAM data structure will require $\tilde{\Omega}(L)$ adaptive rounds of computation, at each stage adding the new item at the root node and flushing items down. Indeed, such process is essential, as adding values anywhere lower down the tree reveals information on which data items are read in future accesses.

[14]The specific structure of [CP13] will also become imperative later in the paper, where we also incorporate load balancing and communication locality into our protocol.

[15]This simplistic flushing step is the primary distinction between the [CP13] ORAM and other related tree-based constructions.

Our ORAM: An Overview. We now describe a parallel insertion procedure for initializing a tree-based ORAM structure (as in [CP13]), while maintaining the three invariants above. For simplicity of notation, we consider insertion of L data *blocks*, as opposed to L data items (which would correspond to L/α blocks). Our construction takes the following high-level approach. Instead of inserting data blocks at the root of the tree and flushing them down over time (which is important for individual data inserts, in order to hide their eventual leaf node destinations), we insert all L blocks *directly* into the level ℓ of the tree for which $2^\ell \in \Theta(L)$. To hide where the data is inserted, it means all nodes on this level must be accessed, but this is acceptable as it entails only $2^\ell = \Theta(L)$ computation.

More explicitly, consider L CPUs $C_1, \ldots, C_L$, each beginning with a block y_i of data to be written to distinct memory positions in the underlying database (whose access patterns we wish to hide). Note that L may be smaller than the total database size d. Each C_i samples a random leaf node of the ORAM tree to be associated with the block y_i: i.e., $\mathsf{Pos}(i) \leftarrow [2^\ell]$. At the conclusion of the first phase of the $\mathsf{ParallelInsert}$ procedure, data block y_i will be written to the node at level ℓ along the path to leaf node $\mathsf{Pos}(i)$ in the largest ORAM tree. Since L data blocks are being inserted, and there are $2^\ell \in \Theta(L)$ distinct nodes at this level of the tree, with overwhelming probability no node will be assigned too many data blocks (i.e., there will not be an "overflow"). This insertion procedure will then continue recursively, storing the newly generated L/α-block position map data $(\mathsf{Pos}(1), \ldots, \mathsf{Pos}(L/\alpha))$, and so on.

However, the challenge is how to reach this memory state *without revealing information* about the resulting location of data items in the process. For example, if processor i simply writes his block y_i to its target memory position at level ℓ, then the first ℓ bits of $\mathsf{Pos}(i)$ are completely revealed, rendering the ORAM structure useless.

We solve this problem by delivering memory blocks to their target locations via a *fixed-topology routing network*. Namely, the CPUs will first write the relevant data items (and their corresponding destination addresses) to memory in fixed order, and then rearrange them in ℓ sequential rounds to the proper locations via the routing network.

We next describe the routing network that will be utilized. However, before continuing, a brief clarifying remark.

Remark 3.5 (CPU-to-CPU communication)**.** The procedures described in this section yield applications both in the setting of standard ORAM, and (as we will later show) to the setting of secure multi-party computation. In the former model, it is assumed that CPUs do not communicate directly, but rather can "send" information from one to another via writes and reads to memory in the shared database. In the latter, the "CPUs" will themselves be embodied by parties, who communicate directly. We will use these notations interchangeably across sections of the paper, and will leave it to the reader to translate them accordingly based on context.

Simple L-to-L Oblivious Routing Network. We use a routing subprotocol for delivering messages msg_i originating from L nodes (labeled $i = 1, \ldots, L$), to associated destination addresses $\mathsf{addr}_i \in [L]$. At the conclusion of the protocol, each node j should hold all messages msg_i for which $\mathsf{addr}_i = j$.

For simplicity, assume $L = 2^\ell$ for some $\ell \in \mathbb{N}$ (otherwise, consider the smallest ℓ for which $2^\ell \geq L$). The routing network has depth ℓ; in each level $t = 1, \ldots, \ell$, each node communicates with the corresponding node whose id agrees in all bit locations except for the tth. These nodes

Routing Network
L nodes, $P_1, \dots, P_L$. For simplicity, say $L = 2^\ell$ for $\ell \in \mathbb{N}$.

Input: Each P_i holds message msg_i with target destination $\mathsf{addr}_i \in [L]$, and global threshold K.
Output: Each P_i holds all messages msg_j for which $\mathsf{addr}_j = i$.

For each node $i \in [L]$ and level $t \in [\ell]$, denote by $M_{i,t}$ the set of messages held by node i at time step t, starting at time $t = 0$ with $M_{i,0} = \{\mathsf{msg}_i\}$ for each node $i \in [L]$.
For $t = 1, \dots, \ell$, do the following:

1. The nodes P_i initiate communication in pairs as dictated by the bit representation of their indices i, j. Namely, for all $2^{\ell-1}$ values of $a := \{a_c\}_{c \in [\ell] \setminus \{t\}} \in \{0,1\}^{\ell-1}$, the pair of nodes $P_{a^{(0)}}$ and $P_{a^{(1)}}$ speak together, where $a^{(b)} := a_1 a_2 \cdots a_{t-1} b\ a_{t+1} \cdots a_\ell$.
2. Each pair $(P_{a^{(0)}}, P_{a^{(1)}})$ performs the following exchange of messages:
$$M_{a^{(b)},t+1} \leftarrow \{\mathsf{msg}_j \in M_{a^{(0)},t} \cup M_{a^{(1)},t} : (\mathsf{addr}_j)_t = b\}.$$
3. If $|M_{i,t+1}| > K$, then node P_i aborts.

Figure 4: Routing network for delivering L messages originally held by L nodes to their corresponding destination addresses within $[L]$.

exchange messages according to the tth bit of their destination addresses. This is formally described in Figure 4. After the tth round, each message msg_i is held by a party whose id agrees with the destination address addr_i in the first t bits. Thus, at the conclusion of ℓ rounds, all messages are properly delivered.

Note that this pairwise communication structure frequently appears within *sorting networks*. In our setting, in contrast to sorting networks, data items are not simply maintained or swapped in each step, but rather may also be both directed to one node or the other. This will be necessary for us since (unlike sorting) the source-to-target mapping is not a one-to-one function. (Indeed, direct use of a sorting network on target addresses actually does not seem to help toward our goal, as one is left with a similar predicament of how to move sorted blocks of items to the correct destination nodes without revealing information).

We now show that, if the destination addresses addr_i are uniformly sampled, then with overwhelming probability no node will ever need to hold too many messages at any point during the routing network execution.

Lemma 3.6 (Routing Network)**.** *If L messages begin with target destination addresses addr_i distributed independently and uniformly over $[L]$ in the L-to-L node routing network in Figure 4, then with probability bounded by $(L \log L)2^{-K}$, no intermediate node will ever hold greater than K messages at any point during the course of the protocol execution.*

Proof. Consider an arbitrary node $a \in \{0,1\}^\ell$, at some level t of execution of the protocol. There are precisely 2^t possible messages m_i that could be held by node a at this step, corresponding to those originating in locations $b \in \{0,1\}^\ell$ whose final $\ell - t$ bits agree with those of a. Node a will hold message m_b at the conclusion of round t precisely if the *first* t bits of addr_b agree with those of a. For each such message m_b, the associated destination address addr_b is a random element of $[L]$, which agrees with a on the first t bits with probability 2^t.

For each $b \in \{0,1\}^\ell$ agreeing with a on the final $\ell - t$ bits, define X_b to be the indicator variable that is equal to 1 if addr_b agrees with a on the first t bits. Then the collection of 2^t random variables $\{X_b : b_i = a_i \ \forall i = t+1, \ldots, \ell\}$ are independent, and $X = \sum X_b$ has mean $\mu = 2^t \cdot 2^{-t} = 1$. Note that X corresponds to the number of messages held by node a at level t. By a Chernoff bound,[16] it holds that

$$\Pr[X \geq K] = \Pr[X \geq (1 + (K-1))\mu] < \left(\frac{e^{K-1}}{K^K}\right) < 2^{-K}.$$

Then, taking a union bound over the total number of nodes L and levels $\ell = \log L$, we have that the probability of any node experiencing an overflow at any round is bounded by $(L \log L)2^{-K}$. $\square$

Note that in our n-party MPC application, the parameters L and K will correspond to $L \in \mathsf{poly}(n)$ and $K \in \omega(\log^2 n) \cap \mathsf{polylog}(n)$, so that this probability of overflow will be negligible in n.

We remark that this routing network is parallelized, requiring no more than $\mathsf{polylog}(n)$ sequential rounds, and access-oblivious, in that the communication pattern of who speaks to whom during the protocol is independent of the list of addresses $\mathsf{addr}_1, \ldots, \mathsf{addr}_L$.

ORAM Compiler with Parallel Initialization. We now present a full description and analysis of the new ORAM compiler. The compiler itself is precisely that of [CP13], with the modification that each bucket size is doubled from K to $2K$ (recall the bucket size is the number of data blocks that can be stored at each ORAM tree node without overflow), and with the additional parallel data insertion procedure ParallelInsert, given in Figure 5. Note that the ParallelInsert procedure begins with L processors, and concludes the recursion with a single CPU with $\mathsf{polylog}(d)$-size secret state, which then acts as the standard CPU for future data accesses, as dictated by the [CP13] ORAM scheme.

Proof of Theorem 3.4. We analyze the complexity and security of the presented ORAM compiler.

Complexity. Consider the following complexity measures of the resulting compiler.

Memory overhead: The total amount of allocated memory is precisely twice that of the [CP13] construction, since we maintain the same structure of memory nodes, but double each one's ("bucket") storage size. Thus, our memory overhead, as with [CP13], is $\tilde{O}(1)$.

Computation overhead: Our ORAM compiler induces identical computation overhead as in [CP13] for each Read and Write data access, with a multiplicative factor of 2 due to the doubled bucket size. In particular, this overhead is $\tilde{O}(1)$. It thus remains to analyze the computation requirements for the parallel insertion procedure.

Consider first a single instance of the recursive ParallelInsert procedure (i.e., insetting data within a single binary tree), and suppose we are inserting L data items within this tree. The entirety of the CPU computation work is expended in the execution of the L-to-L routing network. In each level of this network, each of the L CPUs communicates with one other CPU and computes a $\mathsf{polylog}(d)$-size computation (corresponding to combining their $\mathsf{polylog}(d)$-size buckets of memory, and then divvying the blocks according to the relevant bit of their destination addresses). As there are precisely $\log L$ levels of the routing network, and L CPUs,

[16]Exact Chernoff bound used: $\Pr[X > (1+\delta)\mu] < \left(\frac{e^\delta}{(1+\delta)^{1+\delta}}\right)^\mu$ for any $\delta > 0$.

$\mathsf{ParallelInsert}$: L data blocks and CPUs C_i, into database size d (w.r.t. recursion parameter α):

Input: Each CPU holds data block (i, x_i). All ORAM memory nodes begin empty (i.e., $\perp$).
Output: ORAM memory node contents consistent with each data block x_i inserted at location i.

Call $\mathsf{ParallelInsert\text{-}Recurse}$ with L CPUs (and data blocks to be inserted), into database size d.

$\mathsf{ParallelInsert\text{-}Recurse}$($L_{\mathsf{curr}}$ CPUs with data blocks, d_{curr} size database)

1. (Check for base case). If $L_{\mathsf{curr}} = 1$, then terminate $\mathsf{ParallelInsert}$, taking the single currently held data block as the final output CPU secret state. Otherwise, continue.
2. (Sample target destinations for blocks). Each CPU C_i (for $i \in [L_{\mathsf{curr}}]$) performs the following steps. Sample a random leaf node $\mathsf{Pos}(i) \leftarrow [d_{\mathsf{curr}}]$ to be associated with data block i. Write $\mathsf{Pos}(i)$ to the secret CPU state. Let $\ell_{\mathsf{curr}} = \log_2 L_{\mathsf{curr}}$, and let addr_i denote the address of the ℓth level node in the current ORAM tree along the path to leaf $\mathsf{Pos}(i)$.
3. (Insert items to ORAM tree). All L_{curr} CPUs participate in an execution of the L_{curr}-to-L_{curr} routing network from Figure 4 to move each message block x_i to destination address addr_i within the ℓ_{curr}'th level of the ORAM tree.
4. (Recursion setup). For each $i \in [L_{\mathsf{curr}}]$, CPU C_i sends his value $\mathsf{Pos}(i)$ (sampled in Step 1) to CPU $\lfloor i/\alpha \rfloor + 1$ (in the ORAM setting, this communication is done through memory). The $\lceil L_{\mathsf{curr}}/\alpha \rceil$ CPUs $C_1, C_{\alpha+1}, C_{2\alpha+1}, \ldots$ then initiate $\mathsf{ParallelInsert\text{-}Recurse}$ into database size $\lceil d_{\mathsf{curr}}/\alpha \rceil$, where each such CPU C_j holds data block $[\mathsf{Pos}(j), \mathsf{Pos}(j+1), \ldots, \mathsf{Pos}(j+\alpha-1)]$.

Figure 5: Parallel Insertion procedure, used to initialize an ORAM structure with several data items in parallel.

the total amount of computation expended will be $O(L \cdot \mathsf{polylog}(d) \cdot \log L) \in \tilde{O}(L)$, since $L \leq d$ (where the asymptotic $\tilde{O}$ notation is with respect to the full database size d).

Now, the complete $\mathsf{ParallelInsert}$ procedure requires a logarithmic number $\log_\alpha d$ recursive instances of this step. The single-instance $\mathsf{ParallelInsert}$ procedure is repeated $\log_\alpha d$ times in sequence (where $\alpha \in \Omega(1)$ is chosen as a fixed parameter), thus implying a total of $\tilde{O}(L)$ computation. Since the procedure achieves insertion of L data items, the computation overhead is thus $\tilde{O}(1)$.

Parallel complexity of initialization: Consider first a single instance of the ORAM recursion (i.e., populating a single binary tree with data). The insertion procedure $\mathsf{ParallelInsert}$ for L elements within a single data tree takes place via a single execution of the L-to-L fixed-topology routing network, which has logarithmic depth $\log L$.

Finally, recall that the complete ORAM compiler induces a logarithmic number $\log_\alpha d$ of recursive levels. Thus, the total parallel computation time is $O(\log L \cdot \log_\alpha d) \in \tilde{O}(1)$ (where again asymptotics are with respect to database size d).

Security. By [CP13], it suffices to prove that this insertion procedure maintains the three invariants (1)-(3) listed above. We address these one by one.

Invariant 1: *Data block i is stored within a node along the path from the root to leaf $\mathsf{Pos}(i)$.*

Holds directly. That is, in our insertion procedure data block i is written to the ℓth level node along the path to its assigned leaf node $\mathsf{Pos}(i)$. This state is maintained for all future read and write operations.

Invariant 2: *No node will "overflow" its $2K$-size allocated memory bucket (with overwhelming probability).*

The analysis of [CP13] shows that when beginning with an empty ORAM state, and implementing the flushing step after each data access operation, then with overwhelming probability there will never be a time at which any bucket must store greater than K data items. By Lemma 3.6, with overwhelming probability no size-K node will overflow during the execution of the routing procedure. This includes the final resulting assignment of data blocks to nodes within the data tree. Now, recall that the parallel initialization procedure populates only nodes within a single level of the tree. Thus, it holds in particular that our procedure will not assign more than K data blocks along any *path* in the data tree (from the root to any leaf node). Since we set the bucket size in our ORAM protocol to be $2K$, the desired property follows.

Invariant 3: *Given the sequence of access patterns Access up to any given time, the corresponding vector $(\mathsf{Pos}(i))_{i \in d}$ is distributed uniformly over $[d/\alpha]^d$.*

Recall the original values $\mathsf{Pos}(i)$ are each sampled uniformly and independently. Now, consider the access patterns induced by the parallel initialization procedure. This consists only of recursive implementation of the fixed-topology message-routing protocol. Because the communication access patterns have a fixed topology, this means they are completely independent of the values of $\mathsf{Pos}(i)$. Thus, conditioned uniformity of $\mathsf{Pos}(i)$ follows. This is maintained for all future accesses by the security of the underlying [CP13] ORAM compiler.

□

3.3 Securely Initializing and Computing: DistribInit and Compute Protocols

We now shift back to the setting of secure multiparty computation. In this section, we present two distributed protocols: DistribInit and Compute.

DistribInit: The DistribInit protocol enables secure distributed execution of a parallel ORAM initialization algorithm (e.g., the one given in the previous section) on parties' committed secret inputs, assuming a structure of "good" committees (e.g., as generated by the Committee Setup procedure detailed in Section 3.1), and assuming parties' inputs are properly secret shared amongst corresponding input committees. At the conclusion of DistribInit, the committees $C, \{C_j\}_{j \in [N]}$ corresponding to the CPU and Input ORAM memory nodes hold secret sharings of the state and data information consistent with having inserted the specified inputs securely into the ORAM structure.

Compute: The Compute protocol is used to execute a given RAM program Π on data stored within a distributed ORAM structure. As above, it assumes a structure of "good" committees serving the role of nodes (CPU and memory nodes) of an ORAM data structure. At the conclusion of Compute, all parties receive the output of Π evaluated on the ORAM-stored data, and committees hold secret sharings of the new ORAM CPU and memory node states. Note that our protocol will similarly support the case where different parties receive different outputs; however, we address the simpler case for the sake of clarity (especially when considering a load-balanced solution in Section 4).

Syntax of ORAM-Compiled Programs. Before proceeding, it will be useful to introduce formal notation for the structure of ORAM-compiled RAM programs.

Consider the standard ORAM setting, where nodes are honest, and memory content and CPU computations are hidden, for a *single* data structure. Here, an ORAM-compiled program $\Pi' \leftarrow ORAM(\Pi)$ can be thought of itself as a generic RAM program, consisting of an arbitrary sequence of local CPU computations and data accesses. The guaranteed security is that, assuming the activity of the CPU is kept secret, then the distribution of data access patterns made by the CPU to memory nodes is simulatable, independent of the contents of the memory nodes.

In our setting, we require slightly stronger guarantees.

First, we may no longer assume all information regarding local CPU computations is hidden. Rather, the CPU instructions will be emulated via a committee MPC, which will succeed in hiding the content of the computation, but will reveal its *length*: Namely, if the CPU must perform different computations whose complexities differ as a function of the data, then this information will be revealed. We address this issue by assuming that ORAM-compiled programs maintain a regular pattern of computation and data accesses. That is, we assume that an ORAM-compiled program makes precisely one data access after every fixed-size computation step. (Alternatively, this can be forced with only a $\mathsf{polylog}(n)$ multiplicative blowup in complexity, since each data access incurs $\mathsf{polylog}(n)$ complexity cost).

Second, recall that in our protocol we will actually maintain *two* ORAM-compiled data structures: one to hold parties' inputs and remnant state information maintained between computations, and a second to hold temporary computation-specific working memory (see discussion on Distributed ORAM of Dynamic Size in the introduction). While the ORAM guarantees that access patterns *within each data structure* do not reveal information about the corresponding stored data, no guarantees are provided for cross-structure access patterns. For example, different values of a particular data item in ORAM 1 may result in CPU commands that make different numbers of accesses to ORAM 2. We battle this by taking such an ORAM-compiled program to necessarily make a single data access to *both* ORAMs after every fixed-size computation step (once more introducing only a constant-factor blowup in its complexity). This can be viewed as a rudimentary implementation of an ORAM over two data entities (which are each individually protected), in which one simply touches both entities in each access.

Combining these observations, we have the syntax in Remark 3.7. We consider two types of ORAM-compiled programs: First, those with a single ORAM data structure, running with multiple processors in parallel (to be used for the parallel input insertion); Second, those executed via a single CPU, within a *dual-ORAM* data structure (to be used for evaluating programs Π using the Input and Work Tape ORAM data structures).

Remark 3.7 (Syntax of ORAM-Compiled Programs.)**.** We assume the following syntax of ORAM-compiled programs:

(Single-ORAM) Parallel m-Processor Data Insertion Program:

$$\Pi'_{\mathsf{ins}} = \left\{ \begin{array}{c} \Big((\Pi_1^{(1)}, \mathsf{Access}_1^{(1)}), (\Pi_2^{(1)}, \mathsf{Access}_2^{(1)}), \cdots\Big) \\ \vdots \\ \Big((\Pi_1^{(m)}, \mathsf{Access}_1^{(m)}), (\Pi_2^{(m)}, \mathsf{Access}_2^{(m)}), \cdots\Big) \end{array} \right\},$$

where each (possibly random) subcomputation $\Pi_t^{(i)}$ has equal and input-independent running time, all $\Pi_t^{(i)}$ (respectively, $\mathsf{Access}_t^{(i)}$) are executed simultaneously in parallel during time step t, and $\Pi_t^{(i)}$ and $\mathsf{Access}_t^{(i)}$ have the following form:

- $(\mathsf{state}, \mathsf{op}, \mathsf{addr}) \leftarrow \Pi_t^{(i)}(\mathsf{state}, v)$.
  On input a CPU state and data value v (from the previous data access), $\Pi_t^{(i)}$ outputs: an updated state, a data-access operation instruction op, and a corresponding data address addr. Note that for our application, $\mathsf{state}, \mathsf{op}$, and v will be maintained as secret values, and addr will be public.
- $(v_1, v_2) \leftarrow \mathsf{Access}_t^{(i)}(\mathsf{op}, v)$.
  On input a data-access operation instruction (from the CPU), and a data value v (from the memory node), $\mathsf{Access}_t^{(i)}$ outputs two updated memory values v_1, v_2 (which will be given to the CPU and memory node, respectively). An explicit definition of $\mathsf{Access}_t^{(i)}$ is given in Figure 6.

Note that the data insertion program has no public outputs.

(Dual-ORAM) Computation Program:

$$\Pi' = \Big((\Pi'_1, \mathsf{Access}_1, \mathsf{Access}_1^{\mathsf{Work}}), \ldots, (\Pi'_{q'}, \mathsf{Access}_{q'}, \mathsf{Access}_{q'}^{\mathsf{Work}})\Big),$$

where Access_t, $\mathsf{Access}_t^{\mathsf{Work}}$ are both as $\mathsf{Access}_t^{(i)}$ above, and Π'_t has the following form:

- $\big(\mathsf{state}, (\mathsf{op}, \mathsf{addr}), (\mathsf{op}^{\mathsf{Work}}, \mathsf{addr}^{\mathsf{Work}}), \mathsf{output}\big) \leftarrow \Pi'_t(\mathsf{state}, v, v^{\mathsf{Work}})$.
  On input a CPU state and *two* data values (corresponding to the previous Input and Work Tape data accesses), Π'_t outputs: an updated state, *two sets* $((\mathsf{op}, \mathsf{addr}), \mathsf{op}^{\mathsf{Work}}, \mathsf{addr}^{\mathsf{Work}}))$

Function $\mathsf{Access}(\mathsf{op}, v)$: Performed by two entities (CPU C and memory node C_{addr})

Input:	C: data-access instruction op.	C_{addr}: data value v
Output:	C: data value v_1.	C_{addr}: updated data value v_2.

1. Parse op as containing a read or write command:
   - For $\mathsf{op} = \mathsf{Read}$: Set $v_1 \leftarrow v$. Keep $v_2 = v$.
   - For $\mathsf{op} = (\mathsf{Write}, v')$: Set $v_1 \leftarrow \emptyset$ and $v_2 \leftarrow v'$.
2. Output v_1 to C and v_2 to C_{addr}.

Figure 6: The function Access, used to implement a Read or Write command held within the CPU as op.

of data-access instruction and data address values, and additionally an output value and party identifier. Note that $\mathsf{op}, \mathsf{op}^{\mathsf{Work}}$ will correspond to one data access in each of the Input and Work Tape ORAM structures. At the final time step q', the computation will also yield the final output value, output (before this step, $\mathsf{output} = \perp$).

The Comm Hybrid Model. We present our protocols within a UC hybrid model, in which the parties have access to a collection of ideal communication-related functionalities, as listed in Figure 7. This provides us with a cleaner interface for communication, without needing to repeatedly specify the underlying details of implementation. We refer to this as the "Comm *hybrid model.*" The Enc, Dec, etc notation refers to the intuition that secret sharing a value among a committee serves as a form of information theoretic "encryption." Note that each functionality involves at most two committees of parties.

We may securely realize each such functionality via a direct use of any information theoretic, UC secure MPC protocol, such as [BGW88]. The associated communication complexity overheads for n' parties, implemented with [BGW88], are given as follows (where each given cost is *per party*):

Functionality	Memory	Rounds	Computation	CC
$\mathcal{F}^{i,\mathcal{E}}_{\mathsf{Enc}}(x)$	$O(n' \cdot \lvert x\rvert)$	$O(1)$	$\mathsf{poly}(\lvert x\rvert)$	$O(\lvert x\rvert \cdot (n')^2)$
$\mathcal{F}^{\mathcal{E},\mathcal{E}'}_{\mathsf{Command}}(\mathsf{op}, v)$	$O(n' \cdot (\lvert\mathsf{op}\rvert + \lvert v\rvert))$	$O(\lvert\mathsf{op}\rvert)$	$O(\lvert\mathsf{op}\rvert)$	$O((\lvert\mathsf{op}\rvert + \lvert v\rvert) \cdot (n')^2)$
$\mathcal{F}^{\mathcal{E}}_{\mathsf{Compute}}(f)$	$O(\lvert f\rvert)$	$O(\lvert f\rvert)$	$O(\lvert f\rvert)$	$O(\lvert f\rvert \cdot (n')^2)$
$\mathcal{F}^{\mathcal{E},i}_{\mathsf{Dec}}(x)$	$O(n' \cdot \lvert x\rvert)$	$O(1)$	$\mathsf{poly}(\lvert x\rvert)$	$O(\lvert x\rvert \cdot (n')^2)$

In particular, note that if executed by committees of size $n' = \mathsf{polylog}(n)$, and with inputs and function description sizes $|x|, |\mathsf{op}|, |f| \in \mathsf{polylog}(n)$, then *all* the above metrics will be $\mathsf{polylog}(n)$.

3.3.1 Distributed Initialization Protocol: $\mathsf{DistribInit}$

We now construct the $\mathsf{DistribInit}$ protocol.

Recall in Section 3.2 we presented an ORAM compiler with an efficient parallel insertion mechanism. Our analysis in that section was within the typical ORAM setting, in which computation is conducted by honest CPU entities, who may read and write data to honest external memory nodes. We now discuss the analogous goal within a modified setting of MPC. Here, all parties begin as symmetric entities, and a constant fraction of them are *malicious.* To mimic the actions of honest

Ideal Functionalities in Comm Hybrid Model:

Party-to-Committee "Encrypt" $\mathcal{F}^{i,\mathcal{E}}_{\mathsf{Enc}}$

Input: Party P_i: input x.

Compute: Generate secret shares $[x]_{\mathcal{E}} \leftarrow \mathsf{Share}(x, \mathcal{E})$.

Output: Parties in committee $\mathcal{E}$: secret shares $[x]_{\mathcal{E}}$

Committee-to-Committee "Command" $\mathcal{F}^{\mathcal{E},\mathcal{E}'}_{\mathsf{Command}}$

Input: Parties in committee $\mathcal{E}$ and $\mathcal{E}'$: secret shares $[\mathsf{op}]_{\mathcal{E}}$ and $[v]_{\mathcal{E}'}$ (respectively).

Compute: Reconstruct $\mathsf{op} \leftarrow \mathsf{Reconst}([\mathsf{op}]_{\mathcal{E}})$, $v \leftarrow \mathsf{Reconst}([v]_{\mathcal{E}'})$. Evaluate $(y_1, y_2, y_{\mathsf{out}}) \leftarrow U(\mathsf{op}, v)$.

Secret share $[y_1]_{\mathcal{E}} \leftarrow \mathsf{Share}(y_1, \mathcal{E})$ and $[y_2]_{\mathcal{E}'} \leftarrow \mathsf{Share}(y_2, \mathcal{E}')$.

Output: Parties in committee $\mathcal{E}$: shares $[y_1]_{\mathcal{E}}$. Parties in committee $\mathcal{E}'$: shares $[y_2]_{\mathcal{E}'}$. All parties: y_{out}.

Committee-to-Self "Compute" $\mathcal{F}^{\mathcal{E}}_{\mathsf{Comp}}$

Input: Parties in committee $\mathcal{E}$: secret shares $[v]_{\mathcal{E}}$, (possibly randomized) function f.

Compute: Reconstruct $v \leftarrow \mathsf{Reconst}([v]_{\mathcal{E}})$. Evaluate $(y, y_{\mathsf{out}}) \leftarrow f(v)$. Secret share $[y]_{\mathcal{E}} \leftarrow \mathsf{Share}(y, \mathcal{E})$.

Output: Parties in committee $\mathcal{E}$: y_{out}, secret shares $[y]_{\mathcal{E}}$.

Committee-to-Party "Decrypt" $\mathcal{F}^{\mathcal{E},i}_{\mathsf{Dec}}$

Input: Parties in committee $\mathcal{E}$: secret shares $[v]_{\mathcal{E}}$.

Compute: Reconstruct value $v \leftarrow \mathsf{Reconst}([v]_{\mathcal{E}})$.

Output: Party i: value v.

Figure 7: Ideal functionalities assumed within the "Comm hybrid model."

agents (i.e., to emulate the ORAM security setting), in our MPC protocol every role of the ORAM (both CPU and memory nodes) will be undertaken by a committee of parties, who may perform computations and communicate with other committees.

More explicitly, suppose all parties agree on a collection of N committees $\{C_j\}_{j\in[N]}$—one for each memory node in the Input ORAM database—such that each committee is at least 2/3 honest. Suppose the first n of these committees each begin with (honestly generated) secret shares of a corresponding input value x_i.[17] These committees $C_1, \ldots, C_n$ will serve the role of the n CPUs in the multi-processor ORAM insertion procedure ParallelInsert.

The protocol DistribInit**:**

Denote the sequence of n parallel CPU instructions as dictated by ParallelInsert as

$$\Pi'_{\mathsf{ins}} = \left\{ \begin{array}{c} \Big((\Pi_1^{(1)}, \mathsf{Access}_1^{(1)}), (\Pi_2^{(1)}, \mathsf{Access}_1^{(1)}), \cdots \Big) \\ \vdots \\ \Big((\Pi_1^{(n)}, \mathsf{Access}_1^{(n)}), (\Pi_2^{(n)}, \mathsf{Access}_1^{(n)}), \cdots \Big) \end{array} \right\},$$

as in Remark 3.7. Then each of the committees C_i, $i \in [n]$ executes the corresponding ith sequence of dictated steps, as follows:

- Initialization: At the beginning of the ParallelInsert execution, each committee C_i initializes empty secret shares $[\mathsf{state}^{(i)}]_{C_i} = \emptyset$ and $[v]_{C_i} = \emptyset$.
- Each Π_t: In parallel, each committee C_i initiates an execution of the ideal functionality $\mathcal{F}^{C_i}_{\mathsf{Compute}}$ on his instruction $f = \Pi_t^{(i)}$, and with input secret shares $[\mathsf{state}^{(i)}]_{C_i}$ and $[v]_{C_i}$. Recall the output of the ideal functionality is composed of secret shares of the output value $\Pi_t^{(i)}(\mathsf{state}, v)$, evaluated on the reconstructed values state, v (see Figure 7), and that this $\Pi_t^{(i)}$ output consists of a triple $(\mathsf{state}, \mathsf{op}, \mathsf{addr})$ of updated state information, a data-access operation instruction, and a (public) data address (see Remark 3.7). That is,

$$([\mathsf{state}]_{C_i}, [\mathsf{op}]_{C_i}, \mathsf{addr}^{(i)}) \leftarrow \mathcal{F}^{C_i}_{\mathsf{Compute}}(\Pi_t^{(i)})([\mathsf{state}]_{C_i}, [v]_{C_i}).$$

- Each Access_t: In parallel, each committee C_i initiates an execution of the ideal functionality $\mathcal{F}^{C_i, C_{\mathsf{addr}^{(i)}}}_{\mathsf{Command}}$ to evaluate the procedure Access (as defined in Figure 6) together with committee $C_{\mathsf{addr}^{(i)}}$, where $\mathsf{addr}^{(i)}$ is the public data address output in the previous step by CPU i. More formally, C_i initiates interaction by having each of its parties send a public "start of command" message to all parties in $C_{\mathsf{addr}^{(i)}}$. Then, the committees submit their respective inputs to the ideal functionality: C_i inputs shares $[\mathsf{op}]_{C_i}$, and $C_{\mathsf{addr}^{(i)}}$ inputs shares $[v]_{C_{\mathsf{addr}^{(i)}}}$; the output of $\mathcal{F}^{C_i, C_{\mathsf{addr}^{(i)}}}_{\mathsf{Command}}$ consists of secret shares $[v_1]_{C_i}$ to committee C_i, and secret shares $[v_2]_{C_{\mathsf{addr}^{(i)}}}$ to committee $C_{\mathsf{addr}^{(i)}}$. That is,

$$([v_1]_{C_i}, [v_2]_{C_{\mathsf{addr}^{(i)}}}) \leftarrow \mathcal{F}^{C_i, C_{\mathsf{addr}^{(i)}}}_{\mathsf{Command}}([\mathsf{op}]_{C_i}, [v]_{C_{\mathsf{addr}^{(i)}}}),$$

[17]For simplicity of exposition, we describe the protocol for the case when parties' inputs are of short size $\mathsf{polylog}(n)$, so that a single committee can be assigned to each input. In the case of large input size, the same process will take place, except that parties will first break their inputs into $\mathsf{size} \in \mathsf{polylog}(n)$-size pieces, secret share each of these $|x'| := |x|/\mathsf{size}$ pieces amongst a different committee (in parallel), and then execute the parallel ORAM input insertion procedure collectively on $n \cdot |x'|$ data items (instead of just n).

where op is executed as dictated by the procedure Access, described in Figure 6. Recall that op encodes a Read or (Write, v') command, the input value v corresponds to the data stored at address $\mathsf{addr}^{(i)}$, and the output values v_1, v_2 are the appropriate data values given to C_i and $C_{\mathsf{addr}^{(i)}}$ (respectively) at the conclusion of the Read or Write command.

We first analyze the (per-party) communication and space complexities of $\mathsf{DistribInit}$. Overall, the protocol corresponds to one execution of $\mathcal{F}^{\mathcal{E}}_{\mathsf{Compute}}$ or $\mathcal{F}^{\mathcal{E},\mathcal{E}'}_{\mathsf{Command}}$ per computation step of the ORAM parallel insertion program $\mathsf{ParallelInsert}$. Each ideal functionality execution will take place between $\mathsf{polylog}(n)$ many parties (corresponding to at most two committees) on input size $\mathsf{polylog}(n)$ (since we w.l.o.g. may consider each input as split into $\mathsf{polylog}(n)$-size blocks). By implementing the ideal functionality via a UC secure protocol such as [BGW88], each execution will take memory, round complexity, and communication complexity $\mathsf{polylog}(n)$. Now, from the previous section, we have that $\mathsf{ParallelInsert}$ runs in parallel time complexity $\mathsf{polylog}(n)$ and has $\mathsf{polylog}(n)$ memory overhead. (Note that the database size $d = n|x|$ is polynomial in n, thus $\mathsf{polylog}(d) \in \mathsf{polylog}(n)$). Further, the computation overhead per CPU is $\mathsf{polylog}(n)$, implying that each CPU need only perform $\mathsf{polylog}(n)$ ideal functionality accesses. Since communications take place at each computation step, parties' computation are in all cases within $\mathsf{polylog}(n)$ factor of their communication.

This means each party maintains $\mathsf{polylog}(n)$ bits of memory and communicates $\tilde{O}(|x|)$ bits for each committee role he takes part in. By the properties of the ORAM compiler, which provides $\mathsf{polylog}(n)$ overhead in memory requirements, the total number of committees will only be a $\mathsf{polylog}(n)$ multiplicative factor greater than the size of the required underlying (non-compiled) databases. This means the total collective memory requirement of the protocols (combining all parties) is $\tilde{O}(n \cdot |x|)$. Finally, by Lemma 3.2, with overwhelming probability no party appears in more than $\mathsf{polylog}(n)$ times more than "his share" of committees. That is, no party need to have memory greater than $\tilde{O}(|x|)$, or communicate more than $\tilde{O}(|x|)$ bits. Hence, the final per-party memory, round, and CC/computation requirements of $\mathsf{DistribInit}$ are as given in Table 2.

	Memory	Rounds	CC/comp	*Total* CC/comp
$\mathsf{DistribInit}$	$\tilde{O}(\lvert x\rvert)$	$\tilde{O}(1)$	$\tilde{O}(\lvert x\rvert)$	–

Table 2: Per-party memory and communication complexities of $\mathsf{DistribInit}$. The symbol '–' in the totals column denotes n times the corresponding per-party complexity.

We now proceed to prove the security of $\mathsf{DistribInit}$. We prove that the protocol $\mathsf{DistribInit}$ securely realizes an ideal functionality $\mathcal{F}^{\mathsf{addr}}_{\mathsf{ins}}$ described in Figure 8. The functionality $\mathcal{F}^{\mathsf{addr}}_{\mathsf{ins}}$ accepts secret shares from all parties (for each committee they are participating) and outputs secret shares consistent with all parties' inputs being inserted into the Input ORAM structure (i.e., held as secret shares among the appropriate ORAM committees, and with corresponding updated CPU state secret shared among the CPU committee C), and which leaks no information beyond the output secret shares and the data access patterns made to the ORAM structure during the insertion process. (This access pattern information will later be removed when using $\mathsf{DistribInit}$ within our final protocol, appealing to the security of the parallel initalization procedure of the ORAM).

More specifically, we prove that $\mathsf{DistribInit}$ securely UC-realizes $\mathcal{F}^{\mathsf{addr}}_{\mathsf{ins}}$ for a *limited class* of environments, that is restricted to selecting parties' inputs consistent with valid sets of secret shares (note, of course, adversarially controlled parties can always ignore these values and use

Ideal Functionality $\mathcal{F}_{\mathsf{ins}}^{\mathsf{addr}}$:

Input: For $j \in [n]$, each committee C_j: shares $[x_j]_{C_j}$.

Compute: Perform the following steps.

1. Reconstruct the values $x_j \leftarrow \mathsf{Reconst}([x_j]_{C_j})$.
2. Evaluate the ORAM parallel insertion procedure on these inputs:
$$\Big(\big(\mathsf{addr}_t^{(1)}, \ldots, \mathsf{addr}_t^{(n)}\big)_{t \in [q]}, \big(\mathsf{state}, \{v_j\}_{j \in [N]}\big) \Big) \leftarrow \mathsf{ParallelInsert}(x_1, \ldots, x_n).$$
Let $\mathsf{AccessPatterns} := (\mathsf{addr}_t^{(1)}, \ldots, \mathsf{addr}_t^{(n)})_{t \in [q]}$ denote the sequence of memory access patterns induced by this multiprocessor procedure.
3. Generate secret shares of secret output values: $[\mathsf{state}]_C \leftarrow \mathsf{Share}(\mathsf{state}, C)$; for each $j \in [N]$, $[v_j]_{C_j} \leftarrow \mathsf{Share}(v_j, C_j)$.

Output: The committees receive the following values:

- CPU committee C: shares $[\mathsf{state}]_C$.
- Each Input ORAM committee C_j ($j \in [N]$): shares $[v_j]_{C_j}$.
- All parties: $\mathsf{AccessPatterns}$.

Figure 8: The ideal functionality $\mathcal{F}_{\mathsf{ins}}^{\mathsf{addr}}$ to be realized by DistribInit.

arbitrary inputs of their choice). When we put the separate pieces of our MPC protocol together, we will be guaranteed that the inputs to DistribInit (corresponding to the outputs of the Committee Setup phase) will indeed be of this form.[18]

Lemma 3.8 (Input Initialization for MPC)**.** *Suppose all parties agree on committees $C, \{C_j\}_{j \in [D]}$ as in Lemma 3.2. Then the protocol* DistribInit *securely UC-realizes the ideal functionality $\mathcal{F}_{\mathsf{ins}}^{\mathsf{addr}}$ given in Figure 8, within the* Comm *hybrid model, when restricting to environments who select parties' inputs with valid sets of secret shares.*

Proof. At a high level, the security (including correctness) of the protocol DistribInit follows directly from the UC security of the underlying ideal functionalities in the Comm hybrid model (see Figure 7), together with the secrecy and correctness/robustness of the secret sharing scheme. Indeed, the entire DistribInit protocol consists of a parallel sequence of public communications and calls to the underlying ideal functionalities made by the CPU committees amongst themselves (to execute each $\Pi_t^{(i)}$), and between CPU committee plus memory node committee pairs (to execute each data access). We need only argue that the outputs of these ideal functionalities (in particular, the secret shares given to malicious parties) do not reveal too much information, and that the process of continually reconstructing shares, computing, and resharing maintains correctness of the underlying secret computed value.

[18]An alternative approach to this situation would be to prepend to the DistribInit protocol an additional VSS phase in which parties commit to their input secret shares (by sharing these shares among the parties in the corresponding committees), in which case DistribInit will securely UC-realize $\mathcal{F}_{\mathsf{ins}}^{\mathsf{addr}}$ in the standard sense. However, for simplicity of protocol construction, we elect to skip this step (which is redundant in the fully composed protocol), and simply analyze with respect to restricted environments.

To formally prove security, we construct an ideal-world simulator $\mathcal{S}$ who simulates the entire output of the real-world execution of $\mathsf{DistribInit}$ given only access to the ideal functionality $\mathcal{F}_{\mathsf{ins}}^{\mathsf{addr}}$.

The simulator $\mathcal{S}_{\mathsf{ins}}$. Denote by $M \subset [n]$ the subset of malicious parties corrupted by the adversary.

1. Pre-computation. Let

$$\Pi'_{\mathsf{ins}} = \left\{ \begin{array}{c} \Big((\Pi_1^{(1)}, \mathsf{Access}_1^{(1)}), \cdots (\Pi_q^{(1)}, \mathsf{Access}_q^{(1)}), \cdots\Big) \\ \vdots \\ \Big((\Pi_1^{(n)}, \mathsf{Access}_1^{(m)}), (\Pi_q^{(n)}, \mathsf{Access}_q^{(n)}), \cdots\Big) \end{array} \right\}$$

   denote the n parallel sequences of processor instructions dictated by the n-item parallel data item ORAM insertion procedure $\mathsf{ParallelInsert}$, as in Remark 3.7. In particular, let q denote the number of (parallel) computation steps.

2. The simulator $\mathcal{S}$ executes the ideal functionality $F_{\mathsf{ins}}^{\mathsf{addr}}$ using arbitrary inputs on behalf of corrupted parties (recall that these correspond to a minority of secret shares, and we are guaranteed that the honest parties' shares will be consistent and reconstructable). In response, $\mathcal{S}$ receives output secret shares $[\mathsf{state}]_{C \cap M}$ and $[v_j]_{C_j \cap M}$ (for each $j \in [N]$) given to corrupted parties in the corresponding committees, and a description $\mathsf{AccessPatterns}$ of the sequence of communication access patterns made during the protocol.

3. For $t = 1, \ldots, q$, the simulator $\mathcal{S}$ proceeds as follows.

   (a) Simulate parallel computation of $\Pi_t^{(1)}, \ldots, \Pi_t^{(n)}$ by committees $C_1, \ldots, C_n$:
   This constitutes simulating the outputs of each ideal functionality $\mathcal{F}_{\mathsf{Compute}}^{C_i}(\Pi_t^{(i)})$ on the corresponding collection of secret shares $([\mathsf{state}^{(i)}]_{C_i}, [v^{(i)}]_{C_i})$ (including shares held by honest parties). Recall the output of $\mathcal{F}_{\mathsf{Compute}}^{C_i}(\Pi_t^{(i)})$ consists of new sets of secret shares $[\mathsf{state}^{(i)}]_{C_i}, [\mathsf{op}^{(i)}]_{C_i}$ for the committee C_i, corresponding to updated state, data-access operation instruction, and output value, and public data address $\mathsf{addr}_t^{(i)}$, corresponding to the next-step memory access location.
   To simulate this output, $\mathcal{S}$ does the following:
   - $\mathcal{S}$ samples *random* secret shares for each malicious party in C_i for $[\mathsf{state}^{(i)}]_{C_i}$ and $[\mathsf{op}^{(i)}]_{C_i}$.
   - Output the value $\mathsf{addr}_t^{(i)}$ given by the ideal functionality (as part of *Access*$\mathsf{Patterns}$).

   (b) Simulate parallel computation of $\mathsf{Access}_t^{(1)}, \ldots, \mathsf{Access}_t^{(n)}$ (among C_i and $C_{\mathsf{addr}_t^{(i)}}$):
   This constitutes simulating the output of the ideal functionality $\mathcal{F}_{\mathsf{Command}}^{C_i, C'_i}(\mathsf{Access}_t^{(i)})$ for $C'_i := C_{\mathsf{addr}_t^{(i)}}$, on input the collection of secret shares $([\mathsf{op}^{(i)}]_{C_i}, [v]_{C'_i})$ (including shares held by honest parties). Recall the output of $\mathcal{F}_{\mathsf{Command}}^{C, C'_i}(\mathsf{Access}_t^{(i)})$ consists of new sets of secret shares $[v_1]_{C_i}, [v_2]_{C'_i}$ to C and C'_i (corresponding to the outputs of the Read or Write command specified by $\mathsf{op}^{(i)}$).
   To simulate this output, $\mathcal{S}$ samples *random* secret shares of $[v_1]_{C_i}$ for each malicious party in C, and of $[v_2]_{C'_i}$ for each malicious party in C'_i.

This concludes the description of the simulator $\mathcal{S}$.

To prove the indistinguishability of the output of the ideal-world simulator $\mathcal{S}$ and the output of the real-world experiment, we consider a sequence of hybrids. The output of each hybrid experiment ℓ run with adversary $\mathcal{A}$ and environment $\mathcal{Z}$ is given by the output of the environment, denoted by $\mathsf{Hyb}_\ell\left(1^k, \mathcal{A}_\ell, \mathcal{Z}\right)$.

Hybrid 0. The real world: i.e., the adversary interacts with honest parties in the real-world experiment running the protocol $\mathsf{DistribInit}$.

Hybrid 1. (Secret Share robustness).

In this hybrid, the sequence of committee communications and ideal functionality executions is replaced by a *single* ideal functionality $\mathcal{F}_{\mathsf{ins}}^{ss,\mathsf{addr}}$ which accepts secret shares from all parties, ignores the shares of corrupted parties, reconstructs the corresponding secret values using honest parties' shares, and then computes on these values directly, as dictated by the protocol $\mathsf{DistribInit}$. The ideal functionality outputs precisely the values dictated by the stages of the real-world protocol (e.g., it outputs the computed values $\{\mathsf{addr}_t^{(i)}\}_{i\in[n]}$ at each time step t of the computation, in addition to honestly generated secret shares of the current values $[\mathsf{state}^{(i)}]_{C_i}, [\mathsf{op}^{(i)}]_{C_i}$ at each time step (for each $i \in [n]$).

The only differences are that (1) corrupted parties' shares are ignored in the first step (which will not affect anything when considering these restricted environments), and (2) in the process of computing, $\mathcal{F}_{\mathsf{ins}}^{ss,\mathsf{addr}}$ continues to use the *reconstructed* values, instead of accepting secret shares of these values as input from parties, reconstructing the values from the secret shares, and computing on these values.

Claim 3.9. *For every adversary $\mathcal{A}_0$ in Hybrid 0, there exists an adversary $\mathcal{A}_1$ in Hybrid 1 such that for any environment $\mathcal{Z}$ restricted to selecting parties' inputs as consistent valid secret shares,* $\mathsf{Hyb}_0\left(1^k, \mathcal{A}_0, \mathcal{Z}\right) \equiv \mathsf{Hyb}_1\left(1^k, \mathcal{A}_1, \mathcal{Z}\right)$.

Proof. The claim holds directly by the robust reconstruction property of the secret sharing scheme, together with the restricted environment guarantee (i.e., that honest parties' initial inputs to the protocol are consistent sets of secret shares). The secret share robustness guarantees that secret shares of malicious parties (who form a $< \frac{1}{3}$ minority of each committee) are irrelevant to the secret share reconstruction process, since honest parties will all hold consistent shares of the correct value. Thus, since honest parties faithfully follow the protocol as instructed, the "true" reconstructed values at each stage of computation will be equal to the values used within this hybrid.

Namely, for any adversary $\mathcal{A}_0$ in Hybrid 0, the claim holds for the corresponding adversary $\mathcal{A}_1$ in Hybrid 1 who simply submits bogus secret shares to the new ideal functionality on behalf of corrupted parties (say, all 0s), and simulates the actions of $\mathcal{A}_0$ consistent with the resulting ideal functionality outputs. $\square$

Hybrid 2. (Secret Share secrecy).

Same as the previous hybrid, except that the ideal functionality $\mathcal{F}_{\mathsf{ins}}^{ss,\mathsf{addr}}$ is replaced by a more restricted functionality $\mathcal{F}_{\mathsf{ins}}^{\mathsf{addr}}$ which accepts the same inputs, and performs the same computation, but *no longer outputs intermediate secret shares* to malicious parties. That is, $\mathcal{F}_{\mathsf{ins}}^{\mathsf{addr}}$ takes the same inputs and performs the same computation steps as $\mathcal{F}_{\mathsf{ins}}^{ss,\mathsf{addr}}$, but only outputs the values $\{\mathsf{addr}_t^{(i)}\}_{t\in[q]}, \mathsf{shares}_q$, where shares_q denotes shares of the *final* state of the computation.

Claim 3.10. *For every adversary $\mathcal{A}_1$ in Hybrid 1, there exists an adversary $\mathcal{A}_2$ in Hybrid 2 such that for any environment $\mathcal{Z}$,* $\mathsf{Hyb}_1\left(1^k, \mathcal{A}_1, \mathcal{Z}\right) \equiv \mathsf{Hyb}_2\left(1^k, \mathcal{A}_2, \mathcal{Z}\right)$.

Proof. The holds by the secrecy property of the secret sharing scheme. That is, since malicious parties only ever see a $< \frac{1}{3}$ minority of shares of any secret value, the distribution of these shares will simply be uniform, independent of the corresponding secret values.

Formally, fix an adversary $\mathcal{A}_1$ in Hybrid 1. Consider the adversary $\mathcal{A}_2$ in Hybrid 2 who simulates the output of $\mathcal{F}_{\Pi'}^{ss,\mathsf{addr}}$ by providing the output of his own ideal functionality $\mathcal{F}_{\Pi'}^{\mathsf{addr}}$ (which outputs everything except intermediate secret shares), and simulates the additional sets of secret shares $(\mathsf{shares}_t)_{t\in[q'-1]}$ by sampling *uniform* shares.

By the perfect secrecy of the secret sharing scheme, the simulated output of $\mathcal{F}_{\mathsf{ins}}^{ss,\mathsf{addr}}$ by $\mathcal{A}_2$ in Hybrid 2 is the *exact* same distribution as the true output of $\mathcal{F}_{\mathsf{ins}}^{ss,\mathsf{addr}}$ in Hybrid 1. Thus, the outputs of the two experiments are identical. □

Note that Hybrid 2 corresponds to the ideal-world experiment as according to ideal functionality $\mathcal{F}_{\mathsf{ins}}^{\mathsf{addr}}$. Security of the DistribInit protocol follows.

□

3.3.2 Distributed RAM-Program Evaluation Protocol: Compute

We now show how the parties can securely evaluate a RAM program Π on the current (dynamic) system state. This is done via a protocol Compute, which we now describe.

At a high level, Compute will follow a very similar structure to the protocol DistribInit from the previous section. Namely, given a RAM program Π, we consider its ORAM-compiled version $\Pi' \leftarrow ORAM(\Pi)$, and evaluate Π' by assigning committees of parties to play the roles of the ORAM CPU and memory nodes. However, there are two major departures from the DistribInit protocol:

1. In DistribInit, we worked with a single ORAM structure (the Input ORAM). Here, in Compute, we use *two* ORAM structures: an Input ORAM, which is first populated by parties' inputs and then maintains cross-computation state information; and a Work Tape ORAM, which stores the current working memory of a single computation Π. As discussed in Remark 3.7, we assume (without loss of generality or efficiency) that after each sub-computation step made by the CPU, precisely one data access is made to both the Input ORAM and to the Work Tape ORAM (in this order).
2. The second Work Tape ORAM does not have a fixed bounded size. Rather, along with each queried RAM program Π is supplied an (optional) bound space on the associated working memory (see Figure 2 for the final ideal functionality we wish to realize).
    - If space is an explicit polynomial $p(n)$, then the protocol will maintain a Work Tape ORAM of size ORAM-Mem$(p(n))$ (corresponding to a database of size $p(n)$). That is, only the first ORAM-Mem$(p(n))$ committees C_ℓ^{Work} will be "activated" in their role during the execution of Π.
    - If space = "unbounded," then the protocol will (implicitly) maintain an ORAM structure of *super-polynomial* size. That is, any of the (implicitly defined) committees C_ℓ^{Work} for $\ell \in$

$n^{\omega(1)}$ can be "activated" as instructed by the ORAM protocol. However, only $\mathsf{polylog}(n)$ many committees will every be activated per time step of the RAM program, by the $\mathsf{polylog}(d)$ worst-case computation and memory overhead of the ORAM protocol, even for databases of super polynomial size say $d = 2^{\log^2 n} \in n^{\omega(1)}$ (as $\mathsf{polylog}(d) = \mathsf{poly}(\log^2 n) \in \mathsf{polylog}(n)$).

At the conclusion of each computation phase, completing the evaluation of some program Π, the memory allocated in the Work Tape ORAM can be released. At any such time, if wished, the CPU can issue a global message "release current Work Tape," and the members of committees C_ℓ^{Work} will each deallocate all associated memory. However, sending such a command to all will incur a $\tilde{O}(n)$ communication cost. In cases where this cost is prohibitive (e.g., if computing low-complexity programs Π, where $\tilde{O}(n)$ overhead is undesirably large), the protocol can instead support a "lazy" memory cleanup process. This is done by maintaining a global session id, which increments (by the CPU) for each new queried RAM program Π. Then each command issued by the CPU committee to other committees is tagged by the current session id, and the receiving committee will be free to deallocate all memory associated with past session id's.

To maintain clean exposition of the protocol, in what follows we omit explicit reference to the memory cleanup process. However, when analyzing the communication and memory requirements of the protocol, we will provide a discussion of both cases.

The protocol $\mathsf{Compute}(\Pi, \mathsf{space})$**:**

Denote the sequence of CPU instructions as dictated by $\Pi' \leftarrow ORAM(\Pi)$ executing time timesteps and with work tape size bound space as

$$\Pi' = \Big((\Pi'_1, \mathsf{Access}_1, \mathsf{Access}_1^{\mathsf{Work}}), \ldots, (\Pi'_{q'}, \mathsf{Access}_{q'}, \mathsf{Access}_{q'}^{\mathsf{Work}})\Big),$$

as in Remark 3.7. Then the CPU committee C executes the sequence of dictated steps, as follows:

- Initialization: At the beginning of the $\mathsf{Compute}$ execution, the CPU committee C increments $\mathsf{SessionId} \leftarrow \mathsf{SessionId} + 1$ and initializes empty secret shares $[\mathsf{state}]_C = \emptyset$ and $[v]_C = \emptyset$ and $[v^{\mathsf{Work}}]_C = \emptyset$.
- Each Π'_t: The CPU committee C initiates an execution of the ideal functionality $\mathcal{F}^C_{\mathsf{Compute}}$ on his instruction $f = \Pi'_t$, and with input secret shares $[\mathsf{state}]_C$ and $[v]_C$ and $[v^{\mathsf{Work}}]_C$. Recall the output of the ideal functionality is composed of secret shares of the output resulting from evaluating $\Pi'_t(\mathsf{state}, v)$, where $\mathsf{state} \leftarrow \mathsf{Reconst}([\mathsf{state}]_C)$, $v \leftarrow \mathsf{Reconst}([v]_C)$, and $v^{\mathsf{Work}} \leftarrow \mathsf{Reconst}([v^{\mathsf{Work}}]_C)$ (see Figure 7), and that this Π'_t output consists of a tuple $(\mathsf{state}, (\mathsf{op}, \mathsf{addr}), (\mathsf{op}^{\mathsf{Work}}, \mathsf{addr}^{\mathsf{Work}}), \mathsf{output}$ of updated state information, two pairs of data-access instruction and (public) data addresses, and output value if the computation has concluded (see Remark 3.7). Explicitly,

$$\Big([\mathsf{state}]_C, ([\mathsf{op}]_C, \mathsf{addr}), ([\mathsf{op}^{\mathsf{Work}}]_C, \mathsf{addr}^{\mathsf{Work}}), \mathsf{output}\Big) \leftarrow \mathcal{F}^C_{\mathsf{Compute}}(\Pi'_t)\Big([\mathsf{state}]_C, [v]_C, [v^{\mathsf{Work}}]_C\Big).$$

- Each Access_t: The CPU committee C initiates an execution of the ideal functionality $\mathcal{F}^{C,C_{\mathsf{addr}}}_{\mathsf{Command}}$ to evaluate the function Access (as defined in Figure 6) together with committee C_{addr}, where addr is the public data address output in the previous step. More formally, C initiates interaction by having each of its parties send a public "start of command" message to all parties

Final Output Delivery Procedure $\mathsf{DeliverOutput}$
Input: Parties in C hold value output.
Output: All parties learn output.

Consider the first nK elected committees C_i, where $K \in \omega(\log n) \cap \mathsf{polylog}(n)$ is a fixed parameter, and temporarily relabel the committees (in order) with binary strings $\emptyset, 0, 1, 00, 01, 10, 11, 000, \ldots$ of length from 0 to $\log_2(nK)$, beginning with the CPU committee receiving label $\emptyset$.

1. Initialize $\mathsf{output}_\emptyset := \mathsf{output}$.
2. For $\ell = 1$ to $\log_2(nK)$:
   (a) In parallel, for each string $v \in \{0,1\}^{\ell-1}$:
   Committee v communicates its value output_v to the two committees $v||0$ and $v||1$ via two executions of the ideal functionality $\mathcal{F}^{v,v'}_{\mathsf{Command}}(\emptyset, \mathsf{output}_v)$.
   (b) Each committee v' on the receiving end of an $\mathcal{F}_{\mathsf{Command}}$ interaction sets $\mathsf{output}_{v'}$ to value received as the result of the interaction.
3. Each party P_i outputs the value output_v for the lexicographically first committee v in which he appears.

Figure 9: Procedure for efficiently disseminating the evaluation output from the CPU committee C to *all* parties.

in C_{addr}. Then, the committees submit their respective inputs to the ideal functionality: C's inputs are shares $[\mathsf{op}]_C$, and C_{addr}'s inputs are shares $[v]_{C_{\mathsf{addr}}}$; the output of $\mathcal{F}^{C,C_{\mathsf{addr}}}_{\mathsf{Command}}$ consists of secret shares $[v_1]_C$ to committee C, and secret shares $[v_2]_{C_{\mathsf{addr}}}$ to committee C_{addr}. That is,

$$([v_1]_C, [v_2]_{C_{\mathsf{addr}}}) \leftarrow \mathcal{F}^{C,C_{\mathsf{addr}}}_{\mathsf{Command}}([\mathsf{op}]_C, [v]_{C_{\mathsf{addr}}}),$$

where op is executed as dictated by the Access procedure, described in Figure 6. Recall that op encodes a Read or (Write, v') command, the input value v corresponds to the data stored at address addr, and the output values v_1, v_2 are the appropriate data values given to C and C_{addr} (respectively) at the conclusion of the Read or Write command.

- Each $\mathsf{Access}^{\mathsf{Work}}_t$: The CPU committee C initiates an execution of the ideal functionality $\mathcal{F}^{C,C^{\mathsf{Work}}_{\mathsf{addr}^{\mathsf{Work}}}}_{\mathsf{Command}}$ to evaluate the function Access together with committee $C^{\mathsf{Work}}_{\mathsf{addr}^{\mathsf{Work}}}$, with inputs $[\mathsf{op}^{\mathsf{Work}}]_C$ and $[v]_{C^{\mathsf{Work}}_{\mathsf{addr}^{\mathsf{Work}}}}$, directly analogous to the Access_t step above.
- Final Output Delivery: At the conclusion of the final q'th step of the program execution, the CPU committee C will have learned the desired program output $y = \Pi(x_1, \ldots, x_n)$. It communicates this output to all parties via a simple tree-structure communication network, as described in Figure 9. Note that this takes place in $\mathsf{polylog}(n)$ rounds.

The Output Delivery Procedure will be used in other parts of the paper, and as such it will be convenient to include a separate lemma stating its correctness and complexity properties.

Lemma 3.11 (Output Delivery Procedure). *Suppose all parties agree on committees $C, \{C_j\}_{j \in N}$ satisfying the properties of Lemma 3.2, for $N = nK$.*[19] *Then with overwhelming probability, if the*

[19]Recall (loosely) Lemma 3.2 guarantees each committee is 2/3 honest, and no party occurs in too many or too few committees.

honest parties in C initiate protocol $\mathsf{DeliverOutput}$ *(Figure 9) with the same value* output*, then at the conclusion all honest parties will agree on* output*, and will use the following* per-party *resources:*

	Memory	*Rounds*	*CC/comp*	Total *CC/comp*
$\mathsf{DeliverOutput}$	$\tilde{O}(\lvert\mathsf{output}\rvert)$	$\tilde{O}(1)$	$\tilde{O}(\lvert\mathsf{output}\rvert)$	-

Further, the exact computation (and communication) requirements for each party are tightly bound to the average: namely, for every party P_i, for any constant $\delta > 0$, with overwhelming probability in n,

$$(1-\delta)\frac{\mathsf{cost}}{n} \le \mathsf{cost}(P_i) \le (1+\delta)\frac{\mathsf{cost}}{n}.$$

Proof. By Lemma 3.2, we have that the committees $C, \{C_j\}$ are each of size $\in \Theta(K) \in \mathsf{polylog}(n)$ and each consist of at least $2/3$ honest majority. Further, it guarantees that for constant $\delta > 0$, no party appears in more than $(1+\delta)$ times or fewer than $(1-\delta)$ times his fair share of committees $\{C_j\}_{j\in[N]}$. In particular, each party occurs in *at least one* committee, and no one appears in more than $\frac{K}{n}\mathbb{N} \in O(\mathsf{polylog}(n))$ (since the size of each committee is bounded by K).

The $2/3$ honest majority implies that the correct value output will be disseminated at each step, by the correctness of $\mathcal{F}_{\mathsf{Command}}$. The protocol consists of a sequence of calls to the ideal functionality $\mathcal{F}_{\mathsf{Command}}$ among $\mathsf{polylog}(n)$-size collections of parties. Each call requires memory and computation resources $\mathsf{cost}_{\mathcal{F}} = \tilde{O}(\lvert\mathsf{output}\rvert)$. A party participates in two such calls for each committee in which he appears. In particular, each party will incur cost $\tilde{O}(\lvert\mathsf{output}\rvert)$. Further, by the guarantees of Lemma 3.2, each party appears in nearly the same number of committees; thus, explicitly, each party's resource requirements will be no greater than $(1+\delta)$ times the average value, and no smaller than $(1-\delta)$ times the average value.

The fact that each party appears in at least one committee guarantees that at the conclusion *all* parties will learn the correct value output. And, by construction, the protocol terminates in $\log_2(nK) \in \mathsf{polylog}(n)$ rounds. □

Remark 3.12 (Output Delivery for Large Outputs)**.** In the protocol $\mathsf{DeliverOutput}$, committees each hold and communicate state information corresponding to the total output size $\lvert\mathsf{output}\rvert$. Later in the paper, when we wish to add load balancing to the protocol, it will be important that committees hold only a $\mathsf{polylog}(n)$-size state and any snapshot in time. (Jumping ahead, this is because a committee who has worked too much at a particular job will pass this job, together with all associated state information, to another committee to balance the workload; thus large state implies high cost of job switching). If one wishes to execute a program Π with large output size, output dissemination can be achieved while maintaining $\mathsf{polylog}(n)$ committee state size by evaluating a version of the program Π that reveals the output in $\mathsf{polylog}(n)$-size pieces, and then executing $\mathsf{DeliverOutput}$ in sequence for each of these pieces. Since the output size is bounded by the runtime of the program Π, this change will only inflict additive round complexity overhead $\tilde{O}(\lvert\Pi\rvert)$, which we can afford.

We first analyze the (per-party) communication and space complexities of $\mathsf{Compute}$, for a given RAM program Π and memory bound space.

Overall, the protocol corresponds to a constant number of executions of $\mathcal{F}^{\mathcal{E}}_{\mathsf{Compute}}$ or $\mathcal{F}^{\mathcal{E},\mathcal{E}'}_{\mathsf{Command}}$ per computation step of the ORAM-compiled program Π', in addition to the output delivery procedure. Each ideal functionality execution will take place between $\mathsf{polylog}(n)$ many parties (corresponding to at most two committees) on input size $\mathsf{polylog}(n)$ (since we w.l.o.g. may consider each input as split into $\mathsf{polylog}(n)$-size blocks). By implementing the ideal functionality via a UC secure protocol such as [BGW88], each execution will take memory, round complexity, and computational complexity $\mathsf{polylog}(n)$. We thus have total round and computation complexity $\tilde{O}(|\Pi|)$. Note, however, that *not all parties* take part in the executions of $\mathcal{F}_{\mathsf{Compute}}$ and $\mathcal{F}_{\mathsf{Command}}$. Rather, each execution takes place only between $\mathsf{polylog}(n)$ parties. (In contrast, in $\mathsf{DistribInit}$, there were n simultaneous parallel executions of these commands). Thus, the total communication complexity suffers only a blowup of $\mathsf{polylog}(n)$ of the per-party CC / computation, and not a factor of n. Further, by the good spread of the elected committees C_i (Lemma 3.2), the output delivery stage will incur only $\mathsf{polylog}(n) \cdot |\mathsf{output}|$ bits of communication per party, where $|\mathsf{output}|$ is the size of the computation output (since each party occurs in only $\mathsf{polylog}(n)$ of the output delivery committees). Thus, altogether, the total communication complexity of the protocol is $\tilde{O}(|\Pi|)$.

Each party maintains $\mathsf{polylog}(n)$ bits of memory for each committee role he takes part in. (Note, in particular, the CPU committee does *not* need to maintain the entire working memory of the computation Π, as in [BGT13], but rather outsources this memory storage to the Work Tape ORAM committees). By the properties of the ORAM compiler, which provides $\mathsf{polylog}(n)$ overhead in memory requirements, the total number of committees will only be a $\mathsf{polylog}(n)$ multiplicative factor greater than the size of the required underlying (non-compiled) databases. This means the total collective memory requirement of the computation protocols (combining all parties) is $\tilde{O}(n \cdot |x| + \mathsf{space})$, corresponding to the memory required by the Input ORAM (contributing $n \cdot |x|$) and the single-execution Work Tape ORAM (contributing $\tilde{O}(\mathsf{space})$). Finallyl, by Lemma 3.2, with overwhelming probability no party appears in more than $\mathsf{polylog}(n)$ times more than "his share" of committees. That is, no party need to have memory greater than $\tilde{O}(|x| + \mathsf{space}/n)$. In addition, the output delivery portion requires $\mathsf{polylog}(n) \cdot |\mathsf{output}|$ memory per party.

We now address the memory requirements for *repeated execution* of $\mathsf{Compute}(\Pi, \mathsf{space})$. For a single execution, we have space complexity as described above. However, after each computation Π, the parties no longer need to maintain the working memory from this computation. As discussed above, this can either be handled by issuing a massive remove-memory request, or by "lazy" memory release, maintaining a session id tag and, upon receiving a message tagged with a new session id, deallocating any memory from previous sessions. In this fashion, we have that at any given time, the total memory requirement of parties for this purpose is bounded by the *maximum space* $\max \mathsf{space}$ required by any one program Π (with $\mathsf{polylog}(n)$ overhead). We thus have the complexities for the repeated-execution case as given in Table 3.

	Memory	Rounds	CC/comp	*Total* CC/comp
$\mathsf{Compute}(\Pi, \mathsf{space})$	$\tilde{O}(\lvert x\rvert + \max \mathsf{space}/n)$	$\tilde{O}(\lvert\Pi\rvert)$	$\tilde{O}(\lvert\Pi\rvert + \lvert y\rvert)$	$\tilde{O}(\lvert\Pi\rvert + n\lvert y\rvert)$

Table 3: Memory and communication complexities of *repeated execution* of $\mathsf{Compute}(\Pi, \mathsf{space})$. The notation $\max \mathsf{space}$ indicates the maximum value of space encountered during the course of a sequence of $\mathsf{Compute}$ calls (where a call $\mathsf{Compute}(\Pi, \mathsf{unbounded})$ defaults to size $\mathsf{space} = |\Pi|$).

We now address the security of the protocol $\mathsf{Compute}$. We prove that $\mathsf{Compute}$ securely UC-realizes an ideal functionality $\mathcal{F}_{\Pi}^{\mathsf{addr}}$, described in Figure 10, when the environment is restricted to selecting parties' inputs consistent with a valid set of secret shares (note, of course, the corrupted parties can always choose to modify their received inputs). As in the case of $\mathsf{DistribInit}$, we will

The ideal functionality $\mathcal{F}_{\Pi}^{\mathsf{addr}}$:

Input: The committees hold secret shares:

- CPU committee C: shares $[\mathsf{state}]_C$.
- Each Input ORAM committee C_j: shares $[v_j]_{C_j}$.
- Each Work Tape ORAM committee C_ℓ^{Work}: shares $[v_\ell^{\mathsf{Work}}]_{C_\ell^{\mathsf{Work}}}$.

Compute:

1. Reconstruct all sets of secret shares: $\mathsf{state} \leftarrow \mathsf{Reconst}([\mathsf{state}]_C)$, $v_j \leftarrow \mathsf{Reconst}([v_j]_{C_j})$, for each j, and $v_\ell^{\mathsf{Work}} \leftarrow \mathsf{Reconst}([v_\ell^{\mathsf{Work}}]_{C_\ell^{\mathsf{Work}}})$ for each ℓ.
2. Denote by $\Pi' \leftarrow ORAM(\Pi)$ the ORAM-compiled version of the program Π. Execute
$$\left(\left(\mathsf{addr}_t, \mathsf{addr}_t^{\mathsf{Work}}, \mathsf{output}\right)_{t\in[q']}, \left(\mathsf{state}, \{v_j\}, \{v_\ell^{\mathsf{Work}}\}\right)\right) \leftarrow \Pi'\left(\mathsf{state}, \{v_j\}, \{v_\ell^{\mathsf{Work}}\}\right),$$
corresponding to the evaluation of the ORAM-compiled program Π' on the preexisting values, where $(\mathsf{addr}_t, \mathsf{addr}_t^{\mathsf{Work}}, \mathsf{output})_{t\in[q']}$ denote the sequence of partial-evaluation output values at each time step t of the execution of Π'.
3. Generate secret shares of each resulting secret value: $[\mathsf{state}]_C \leftarrow \mathsf{Share}(\mathsf{state}, C)$, $[v_j]_{C_j} \leftarrow \mathsf{Share}(v_j, C_j)$ for each j, and $[v_\ell^{\mathsf{Work}}]_{C_\ell^{\mathsf{Work}}} \leftarrow \mathsf{Share}(v_\ell^{\mathsf{Work}}, C_\ell^{\mathsf{Work}})$ for each ℓ.

Output: The parties receive the following outputs:

- Parties in C: shares $[\mathsf{state}]_C$.
- Parties in each C_j: shares $[v_j]_{C_j}$. Parties in each C_ℓ^{Work}: shares $[v_\ell^{\mathsf{Work}}]_{C_\ell^{\mathsf{Work}}}$.
- All parties: output, $(\mathsf{addr}_t, \mathsf{addr}_t^{\mathsf{Work}})_{t\in[q']}$.

Figure 10: The ideal functionality $\mathcal{F}_{\Pi}^{\mathsf{addr}}$ to be realized by Compute.

be guaranteed when composing Compute in the final protocol that the inputs to Compute in every execution (i.e., the outputs of DistribInit or a prior execution of Compute) will indeed have this format. The functionality $\mathcal{F}_{\Pi}^{\mathsf{addr}}$ executes the ORAM-compiled program Π' on the values currently held in the emulated ORAM database (held in secret shared form by the corresponding ORAM memory node committees), outputs new secret shares of the updated ORAM memory nodes and CPU secret state to the appropriate committees, together with the public output to all parties, and leaks only the sequence of access patterns made to the ORAM database during the computation. (We will prove later, in the combined protocol, that these access patterns do not reveal any useful information, by relying on the security of the ORAM compiler).

Lemma 3.13. *Suppose that all parties agree on committees $C, \{C_j\}, \{C_\ell^{\mathsf{Work}}\}$, satisfying the properties in Lemma 3.2. Then the protocol* Compute *securely UC-realizes the ideal functionality $\mathcal{F}_{\Pi}^{\mathsf{addr}}$ given in Figure 10, within the* Comm *hybrid model, when restricted to environments $\mathcal{Z}$ who select parties' inputs corresponding to consistent sets of secret shares.*

Proof. At a high level, the security (including correctness) of the protocol follows directly from the security of the underlying ideal functionalities, together with the secrecy and correctness/robustness of the secret sharing scheme. Indeed, the entire Compute consists of a sequence of calls to the underlying ideal functionalities in the Comm hybrid model. We need only argue that the outputs of these ideal functionalities (in particular, the secret shares given to malicious parties) do not reveal too much information, and that the process of continually reconstructing shares, computing, and resharing maintains correctness of the underlying secret value.

To formally prove security, we construct an ideal-world simulator $\mathcal{S}$ who simulates the entire output of the real-world execution of the $\mathsf{Compute}$ protocol given only access to the ideal functionality $\mathcal{F}_\Pi$.

The simulator $\mathcal{S}_{\mathsf{Compute}}$. Denote by $M \subset [n]$ the subset of malicious parties corrupted by the adversary.

1. Pre-computation. Let $\mathsf{space}(n)$ be an upper bound on the working memory required by Π', and define $p(n) = \mathsf{space}(n) \cdot \mathsf{ORAM\text{-}Mem}(\mathsf{space}(n))$. The simulator evaluates the ORAM compiler $\Pi' \leftarrow ORAM(\Pi)$. Express
$$\Pi' = \Big((\Pi'_1, \mathsf{Access}_1, \mathsf{Access}_1^{\mathsf{Work}}), \ldots, (\Pi'_{q'}, \mathsf{Access}_{q'}, \mathsf{Access}_{q'}^{\mathsf{Work}}) \Big),$$
as in Remark 3.7. In particular, let q' be the number of data access queries to the Input and Work Tape ORAM memory structures.
2. The simulator executes the ideal functionality $\mathcal{F}_\Pi^{\mathsf{addr}}$ using arbitrary inputs on behalf of corrupted parties. (Recall these inputs correspond to a minority of secret shares, and that we are guaranteed honest parties hold a consistent set of shares). In return, the simulator receives sets of secret shares of corrupted parties $[\mathsf{state}]_{C \cap M}$, $\{[v_j]_{C_j \cap M}\}_j$, $\{[v_\ell^{\mathsf{Work}}]_{C_\ell^{\mathsf{Work}} \cap M}\}_\ell$, the output of the evaluation output, and the set of access patterns $(\mathsf{addr}_t, \mathsf{addr}_t^{\mathsf{Work}})_{t \in [q']}$.
3. For $t = 1, \ldots, q'$, $\mathcal{S}$ simulates as follows.
   (a) Simulate computation of Π'_t by C:
   This constitutes simulating the output of the ideal functionality $\mathcal{F}^C_{\mathsf{Compute}}(\Pi'_t)$ on the collection of secret shares $([\mathsf{state}]_C, [v]_C, [v^{\mathsf{Work}}]_C)$ (including shares held by honest parties). Recall the output of $\mathcal{F}^C_{\mathsf{Compute}}(\Pi'_t)$ consists of new sets of secret shares $[\mathsf{state}]_C, [\mathsf{op}]_C, [\mathsf{op}^{\mathsf{Work}}]_C$ for the committee C (corresponding to updated state, data-access operation instructions), and public values $\mathsf{addr}_t, \mathsf{addr}_t^{\mathsf{Work}}, \mathsf{output}$ (corresponding to the next-step memory accesses and public output).
   To simulate this output, $\mathcal{S}$ does the following:
   - $\mathcal{S}$ samples *random* secret shares for each malicious party $P_j \in C$, for each of $[\mathsf{state}]_C$, $[\mathsf{op}]_C$, $[\mathsf{op}^{\mathsf{Work}}]_C$.
   - Output the values addr_t, $\mathsf{addr}_t^{\mathsf{Work}}$, and output (when necessary) given by the ideal functionality (in $\mathsf{AccessPatterns}$).

   (b) Simulate computation of Access_t (among C and C_{addr_t}):
   This constitutes simulating the output of the ideal functionality $\mathcal{F}^{C, C_{\mathsf{addr}_t}}_{\mathsf{Command}}(\mathsf{Access}_t)$ on the collection of secret shares $([\mathsf{op}]_C, [v]_{C_{\mathsf{addr}_t}})$ (including shares held by honest parties). Recall the output of $\mathcal{F}^{C, C_{\mathsf{addr}_t}}_{\mathsf{Command}}(\mathsf{Access}_t)$ consists of new sets of secret shares $[v_1]_C, [v_2]_{C_{\mathsf{addr}_t}}$ to C and C_{addr_t}(corresponding to the outputs of the Read or Write command specified by op). To simulate this output, $\mathcal{S}$ samples *random* secret shares of $[v_1]_C$ for each malicious party in C, and of $[v_2]_{C_{\mathsf{addr}_t}}$ for each malicious party in C_{addr_t}.

(c) Simulate computation of $\mathsf{Access}_t^{\mathsf{Work}}$ (among C and $C^{\mathsf{Work}}_{\mathsf{addr}_t^{\mathsf{Work}}}$):
Analogous to simulation of Access_t above. Namely, $\mathcal{S}$ simulates the output of the ideal functionality $\mathcal{F}_{\mathsf{Command}}^{C,C^{\mathsf{Work}}_{\mathsf{addr}_t^{\mathsf{Work}}}}(\mathsf{Access}_t^{\mathsf{Work}})$ by sampling *random* secret shares $[v_1]_C$ and $[v_2]_{C^{\mathsf{Work}}_{\mathsf{addr}_t^{\mathsf{Work}}}}$ for each malicious party in C and $C^{\mathsf{Work}}_{\mathsf{addr}_t^{\mathsf{Work}}}$.

4. Output delivery. Simulate honestly with public output value output.

This concludes the description of the simulator $\mathcal{S}$.

Proof of indistinguishability. To prove the indistinguishability of the output of the ideal-world simulator $\mathcal{S}$ and the output of the real-world experiment, we consider a sequence of hybrids. The output of each hybrid experiment ℓ run with adversary $\mathcal{A}$ and environment $\mathcal{Z}$ is given by the output of the environment, denoted by $\mathsf{Hyb}_\ell\left(1^k, \mathcal{A}_\ell, \mathcal{Z}\right)$.

Hybrid 0. The real world: i.e., the adversary interacts with honest parties in the real-world experiment running the protocol $\mathsf{Compute}$.

Hybrid 1. (Output Delivery Success).
The same as the real world, except that the experiment ends in fail if the output delivery procedure fails: i.e., if the procedure begins with value output held by committee C, and at the conclusion there exists a party who outputs a value $\neq \mathsf{output}$.

Claim 3.14. *For every adversary $\mathcal{A}_0$ in Hybrid 0, there exists an adversary $\mathcal{A}_1$ in Hybrid 1 such that for any environment $\mathcal{Z}$,* $\mathsf{Hyb}_0\left(1^k, \mathcal{A}_0, \mathcal{Z}\right) \cong \mathsf{Hyb}_1\left(1^k, \mathcal{A}_1, \mathcal{Z}\right)$.

Proof. Follows by Lemma 3.11. □

Hybrid 2. (Secret Share robustness).
In this hybrid, the sequence of committee communications and ideal functionality executions is replaced by a *single* ideal functionality $\mathcal{F}_{\Pi'}^{ss,\mathsf{addr}}$ which accepts secret shares from all parties, reconstructs the corresponding secret values, and then computes on these values directly, as dictated by the protocol. The ideal functionality outputs precisely the values dictated by the stages of the real-world protocol (e.g., it outputs the computed values $(\mathsf{addr}_t, \mathsf{addr}_t^{\mathsf{Work}}, \mathsf{party}_t)$ at each time step t of the computation, in addition to honestly generated secret shares $[\mathsf{state}]_C, [\mathsf{op}]_C, [\mathsf{op}^{\mathsf{Work}}]_C$, etc.)
The only difference is that, in the process of computing, $\mathcal{F}_{\Pi'}^{ss,\mathsf{addr}}$ continues to use the *reconstructed* values, instead of accepting secret shares of these values as input from parties, reconstructing the values from the secret shares, and computing on these values.
More explicitly, $\mathcal{F}_{\Pi'}^{ss,\mathsf{addr}}$ is defined as follows:

Input: $\mathcal{F}_{\Pi'}^{ss,\mathsf{addr}}$ accepts secret shares of currently held values from *all* parties: i.e., $[\mathsf{state}]_C, \{[v_j]_{C_j}\}_{j\in[N]}$, $\{[v_\ell^{\mathsf{Work}}]_{C_\ell^{\mathsf{Work}}}\}_{\ell\in[p(n)]}$.

Compute:

1. Reconstruct (once and for all) the CPU secret state and the value of each Input ORAM and Work Tape ORAM memory node from the set of secret shares received:
   (a) Take $\mathsf{state} \leftarrow \mathsf{Reconst}([\mathsf{state}]_C)$.
   (b) For each $j \in [N]$, take $v_j \leftarrow \mathsf{Reconst}([v_j]_{C_j})$.

(c) For each $\ell \in [p(n)]$, take $v_\ell^{\mathsf{Work}} \leftarrow \mathsf{Reconst}([v_\ell^{\mathsf{Work}}]_{C_\ell^{\mathsf{Work}}})$.

2. Evaluate Π' on this collection of values. That is, initialize $v_{\mathsf{temp}} = \emptyset$ and $v_{\mathsf{temp}}^{\mathsf{Work}} = \emptyset$. For each $t = 1, \ldots, q'$:
   (a) Compute $(\mathsf{state}, (\mathsf{op}, \mathsf{addr}), (\mathsf{op}^{\mathsf{Work}}, \mathsf{addr}^{\mathsf{Work}}), \mathsf{output}) \leftarrow \Pi'_t(\mathsf{state}, v_{\mathsf{temp}}, v_{\mathsf{temp}}^{\mathsf{Work}})$
   Generate shares: $[\mathsf{state}]_C \leftarrow \mathsf{Share}(\mathsf{state}, C)$; $[\mathsf{op}]_C \leftarrow \mathsf{Share}(\mathsf{op}, C)$; $[\mathsf{op}^{\mathsf{Work}}]_C \leftarrow \mathsf{Share}(\mathsf{op}^{\mathsf{Work}}, C)$.
   (b) Compute $(v_{\mathsf{temp}}, v_{\mathsf{addr}}) \leftarrow \mathsf{Access}(\mathsf{op}, v_{\mathsf{addr}})$.
   Generate shares: $[v_{\mathsf{temp}}]_C \leftarrow \mathsf{Share}(v_{\mathsf{temp}}, C)$; $[v_{\mathsf{addr}}]_{C_{\mathsf{addr}}} \leftarrow \mathsf{Share}(v_{\mathsf{addr}}, C_{\mathsf{addr}})$.
   (c) Compute $(v_{\mathsf{temp}}^{\mathsf{Work}}, v_{\mathsf{addr}}^{\mathsf{Work}}) \leftarrow \mathsf{Access}(\mathsf{op}^{\mathsf{Work}}, v_{\mathsf{addr}^{\mathsf{Work}}}^{\mathsf{Work}})$.
   Generate shares: $[v_{\mathsf{temp}}^{\mathsf{Work}}]_C \leftarrow \mathsf{Share}(v_{\mathsf{temp}}^{\mathsf{Work}}, C)$; $[v_{\mathsf{addr}}^{\mathsf{Work}}]_{C_{\mathsf{addr}^{\mathsf{Work}}}} \leftarrow \mathsf{Share}(v_{\mathsf{addr}}^{\mathsf{Work}}, C_{\mathsf{addr}^{\mathsf{Work}}})$.

   Denote the collection of all generated secret shares by
   $\mathsf{shares}_t := ([\mathsf{state}]_C, [\mathsf{op}]_C, [\mathsf{op}^{\mathsf{Work}}]_C, [v_{\mathsf{temp}}]_C, [v_{\mathsf{addr}}]_{C_{\mathsf{addr}}}, [v_{\mathsf{temp}}^{\mathsf{Work}}]_C, [v_{\mathsf{addr}^{\mathsf{Work}}}^{\mathsf{Work}}]_{C_{\mathsf{addr}^{\mathsf{Work}}}})$.

Output: The collection of all intermediate output values and secret shares: i.e.,

$$\left(\mathsf{addr}_t, \mathsf{addr}_t^{\mathsf{Work}}, \mathsf{output}_t, \mathsf{shares}_t\right)_{t \in [q']},$$

where the secret shares are received by the corresponding committee parties, and each output_t is received by party P_{party_t} (and all parties learn $\mathsf{addr}_t, \mathsf{addr}_t^{\mathsf{Work}}, \mathsf{party}_t$).

Claim 3.15. *For every adversary $\mathcal{A}_1$ in Hybrid 1, there exists an adversary $\mathcal{A}_2$ in Hybrid 2 such that for any environment $\mathcal{Z}$ restricted to selecting parties' inputs as consistent sets of secret shares,* $\mathsf{Hyb}_1\left(1^k, \mathcal{A}_1, \mathcal{Z}\right) \equiv \mathsf{Hyb}_2\left(1^k, \mathcal{A}_2, \mathcal{Z}\right)$.

Proof. The claim holds directly by the robust reconstruction property of the secret sharing scheme: namely, secret shares of malicious parties (who form a $< \frac{1}{3}$ minority of each committee) are irrelevant to the secret share reconstruction process, since honest parties will all hold consistent shares of the correct value. Thus, since honest parties faithfully follow the protocol as instructed, the "true" reconstructed values at each stage of computation will be equal to the values used within this hybrid.

Namely, for any adversary $\mathcal{A}_1$ in Hybrid 1, the claim holds for the corresponding adversary $\mathcal{A}_2$ in Hybrid 2 who simply submits bogus secret shares to the new ideal functionality on behalf of malicious parties (say, all 0s), and simulates the actions of $\mathcal{A}_1$ consistent with the resulting ideal functionality outputs.

□

Hybrid 3. (Secret Share secrecy). Same as the previous hybrid, except that the ideal functionality $\mathcal{F}_{\Pi'}^{ss,\mathsf{addr}}$ is replaced by a more restricted functionality $\mathcal{F}_{\Pi'}^{\mathsf{addr}}$ which accepts the same inputs, and performs the same computation, but *no longer outputs intermediate secret shares* to malicious parties. That is, $\mathcal{F}_{\Pi'}^{\mathsf{addr}}$ takes the same inputs and performs the same computation steps as $\mathcal{F}_{\Pi'}^{ss,\mathsf{addr}}$, but only outputs the values $(\mathsf{addr}_t, \mathsf{addr}_t^{\mathsf{Work}}, \mathsf{output}_t)_{t \in [q']}, \mathsf{shares}_{q'}$ where $\mathsf{shares}_{q'}$ denotes shares of the *final* state of the computation.

Claim 3.16. *For every adversary $\mathcal{A}_2$ in Hybrid 2, there exists an adversary $\mathcal{A}_3$ in Hybrid 3 such that for any environment $\mathcal{Z}$,* $\mathsf{Hyb}_2\left(1^k, \mathcal{A}_2, \mathcal{Z}\right) \equiv \mathsf{Hyb}_3\left(1^k, \mathcal{A}_3, \mathcal{Z}\right)$.

Proof. The holds by the secrecy property of the secret sharing scheme. That is, since malicious parties only ever see a $< \frac{1}{3}$ minority of shares of any secret value, the distribution of these shares will simply be uniform, independent of the corresponding secret values.

Formally, fix an adversary $\mathcal{A}_2$ in Hybrid 2. Consider the adversary $\mathcal{A}_3$ in Hybrid 3 who simulates the output of $\mathcal{F}_{\Pi'}^{ss,\mathsf{addr}}$ by providing the output of his own ideal functionality $\mathcal{F}_{\Pi'}^{\mathsf{addr}}$ (which outputs everything except intermediate secret shares), and simulates the additional sets of secret shares $(\mathsf{shares}_t)_{t\in[q'-1]}$ by sampling *uniform* shares.

By the perfect secrecy of the secret sharing scheme, the simulated output of $\mathcal{F}_{\Pi'}^{ss,\mathsf{addr}}$ by $\mathcal{A}_3$ in Hybrid 3 is the *exact* same distribution as the true output of $\mathcal{F}_{\Pi'}^{ss,\mathsf{addr}}$ in Hybrid 2. Thus, the outputs of the two experiments are identical. □

Note that Hybrid 3 corresponds to the ideal-world experiment as according to $\mathcal{F}_{\Pi}^{\mathsf{addr}}$. Security of Compute follows.

□

3.4 Putting the Pieces Together: Secure MPC for Dynamic Functionalities

The final desired secure computation protocol is now simply formed by combining the three protocols developed in the previous sections: (1) Committee Setup, (2) Distributed Initialization, and (3) Compute, for each desired RAM program instance Π. Formally, our complete MPC protocol is given in Figure 11.

Remark 3.17 (Handling Large Inputs)**.** For simplicity of exposition, in Figure 11 we describe the protocol when parties' inputs are of size below a fixed parameter $\mathsf{size} \in \mathsf{polylog}(n)$, so that we may assign a single committee to each input. In the more general case, where parties' inputs may be large, each party will first split his input into $|x'| := |x|/\mathsf{size}$ blocks of appropriate size, and then commit in parallel to each block among a *separate* committee during Step 1 of the Input Commitment and Initialization phase. For a given choice of input size capability, the committee setup procedure can be straightforwardly modified so as to elect the corresponding number $n|x'|$ of required good input committees. Security of the protocol will follow in an identical fashion.

Security of the constructed protocol follows by the UC security of the underlying pieces, together with the secrecy and robustness properties of the secret sharing scheme.

Proof of Theorem 3.1. We first analyze the complexity measures of the combined protocol. Table 4 combines the complexity measures of the individual sub-protocols from the previous sections, with the exception of a one-time broadcast of $\mathsf{polylog}(n)$ bits made by each party in the ComSetup phase.

We now prove security of protocol MPC-Dyn via a sequence of intermediate hyrids. The output of each hybrid experiment ℓ run with adversary $\mathcal{A}$ and environment $\mathcal{Z}$ is given by the output of the environment, denoted by $\mathsf{Hyb}_\ell\left(1^k, \mathcal{A}_\ell, \mathcal{Z}\right)$.

Hybrid 0. The real-world execution of protocol MPC-Dyn.

Hybrid 1. (ComSetup protocol correctness).

Same as Hybrid 0, except that the experiment terminates in abort at the conclusion of ComSetup if it is *not* the case that all honest parties agree on "good" committees $C, \{C_j\}_{j\in[N]}$, $\{C_\ell^{\mathsf{Work}}\}_{\ell\in[K]}$, as specified in Lemma 3.2.

MPC Protocol for Dynamic Functionalities MPC-Dyn
Securely realizing the functionality $\mathcal{F}_{\mathsf{Dyn}}$

Committee Setup

1. All parties execute the Committee Setup procedure, as described in Figure 3. Denote the resulting committees by $C, \{C_j\}_{j\in[N]}, \{C_\ell^{\mathsf{Work}}\}_{\ell\in n^{\log n}}$.

Input commitment and initialization

1. In parallel, each party P_j, $j \in [n]$, commits his input x_j to committee C_j (elected during the committee setup), via an execution of the ideal functionality $\mathcal{F}_{\mathsf{Enc}}^{j,C_j}(x_j)$ (as described in Figure 7).[20]
2. All parties execute the ORAM initialization procedure DistribInit to commit their inputs to the ORAM structure, as described in Section 3.3.1. Denote the resulting secret shared information as $[\mathsf{state}]_C$, $[v_j]_{C_j}$, for $j \in M$, and $[v_\ell^{\mathsf{Work}}]_{C_\ell^{\mathsf{Work}}}$.

Computation

1. For each queried RAM program Π with specified memory bound space, evaluate Π by executing the protocol $\mathsf{Compute}(\Pi, \mathsf{space})$, as described in Section 3.3.2.

Figure 11: MPC for dynamic functionalities: securely realizing the ideal functionality $\mathcal{F}_{\mathsf{Dyn}}$ given in Figure 2.

	Memory	Rounds	CC/comp	*Total* CC/comp
ComSetup	$\tilde{O}(1)$	$\tilde{O}(1)$	$\tilde{O}(1)$	–
Input Commit	$\tilde{O}(\|x\|)$	$\tilde{O}(1)$	$\tilde{O}(\|x\|)$	–
DistribInit	$\tilde{O}(\|x\|)$	$\tilde{O}(1)$	$\tilde{O}(\|x\|)$	–
Compute	$\tilde{O}(\max \mathsf{space}/n)$	$\tilde{O}(\sum \|\Pi\|)$	$\tilde{O}(\sum(\|\Pi\| + \|y\|))$	$\tilde{O}(\sum(\|\Pi\| + n\|y\|))$
Total MPC-Dyn	$\tilde{O}(\|x\| + \max \mathsf{space}/n)$	$\tilde{O}(\sum \|\Pi\|)$	$\tilde{O}(\|x\| + \sum(\|\Pi\| + \|y\|))$	$\tilde{O}(n\|x\| + \sum(\|\Pi\| + n\|y\|))$

Table 4: Per-party complexities of overall protocol MPC-Dyn, assuming an additional one-time broadcast of $\tilde{O}(1)$ bits per party. Here, max space denotes the maximum value of space over queries (Π, space); $|\Pi|$ denotes (worst-case) runtime of Π; and '–' denotes n times the per-party complexity.

Claim 3.18. *For every adversary $\mathcal{A}_0$ in Hybrid 0, there exists an adversary $\mathcal{A}_1$ in Hybrid 1 such that for any environment $\mathcal{Z}$,* $\mathsf{Hyb}_0\left(1^k, \mathcal{A}_0, \mathcal{Z}\right) \cong \mathsf{Hyb}_1\left(1^k, \mathcal{A}_1, \mathcal{Z}\right)$.

Proof. By Lemma 3.2, the probability of terminating in abort in Hybrid 1 is $\mathsf{negl}(n)$, as desired. □

Hybrid 2. (Security of DistribInit protocol).

Same as Hybrid 1, except that the execution of protocol DistribInit is replaced by the ideal functionality $\mathcal{F}_{\mathsf{ins}}^{\mathsf{addr}}$, as defined in Lemma 3.8. Recall that this functionality accepts secret shares from the first n Input ORAM committees $C_1, \ldots, C_n$, which were generated via an execution of the ideal functionality $\mathcal{F}_{\mathsf{Enc}}$ (and thus are guaranteed to be consistent). The

[20]For simplicity, we describe the case where parties' inputs are of small size $\mathsf{polylog}(n)$. See Remark 3.17 for the case of large inputs.

functionality reconstructs the corresponding values x_i from the shares, evaluates the ORAM parallel insertion procedure $\mathsf{ParallelInsert}$ on the values and ORAM structure, and outputs all the intermediate outputs of this computation (including the data access patterns), together with freshly generated secret shares of the final resulting states of the CPU and Input ORAM memory nodes. See Lemma 3.8 for a formal description of $\mathcal{F}^{\mathsf{addr}}_{\mathsf{ins}}$.

Claim 3.19. *For every adversary $\mathcal{A}_1$ in Hybrid 1, there exists an adversary $\mathcal{A}_2$ in Hybrid 2 such that for any environment $\mathcal{Z}$,* $\mathsf{Hyb}_1\left(1^k, \mathcal{A}_1, \mathcal{Z}\right) \cong \mathsf{Hyb}_2\left(1^k, \mathcal{A}_2, \mathcal{Z}\right)$ *.*

Proof. Note that the inputs to the protocol $\mathsf{DistribInit}$ in Hybrid 1, corresponding to secret shares of parties' inputs, come directly from the ideal functionality $\mathcal{F}_{\mathsf{Enc}}$, which is guaranteed to output consistent sets of secret shares. This means that we are safely within the case of restricted environments required by Lemma 3.8 in order to ensure UC security. The claim thus follows by the UC security of $\mathsf{DistribInit}$. □

Hybrid 3. (Security of $\mathsf{Compute}$ protocol).

Same as Hybrid 2, except that each execution of protocol $\mathsf{Compute}$ is replaced by the ideal functionality $\mathcal{F}^{ss,\mathsf{addr}}_{\Pi'}$, as defined in Lemma 3.13. Recall that this functionality accepts secret shares from all parties, reconstructs their values, evaluates the ORAM-compiled program Π' on the values, and outputs all the intermediate outputs of this computation (including the data access patterns), together with freshly generated secret shares of the final resulting state values (see Lemma 3.13 for a formal definition of $\mathcal{F}^{ss,\mathsf{addr}}_{\Pi'}$).

Claim 3.20. *For every adversary $\mathcal{A}_2$ in Hybrid 2, there exists an adversary $\mathcal{A}_3$ in Hybrid 3 such that for any environment $\mathcal{Z}$,* $\mathsf{Hyb}_2\left(1^k, \mathcal{A}_2, \mathcal{Z}\right) \cong \mathsf{Hyb}_3\left(1^k, \mathcal{A}_3, \mathcal{Z}\right)$ *.*

Proof. We consider replacing each single execution of $\mathsf{Compute}$ with $\mathcal{F}^{\mathsf{addr}}_{\Pi}$ one a time, beginning with the first. Note that the inputs to the first execution of $\mathsf{Compute}$ in Hybrid 2 come directly from the ideal functionality $\mathcal{F}^{\mathsf{addr}}_{\mathsf{ins}}$ (which replaced $\mathsf{DistribInit}$), which is guaranteed to output consistent sets of secret shares. This means that we are safely within the case of restricted environments required by Lemma 3.13 in order to ensure UC security. The first step thus follows by the UC security of $\mathsf{Compute}$.

For each following execution of $\mathsf{Compute}$, the inputs to the protocol come directly from the ideal functionality $\mathcal{F}^{\mathsf{addr}}_{\Pi}$ (which replaced the previous $\mathsf{Compute}$ execution), which is guaranteed to output consistent sets of secret shares. This means we are again safely within the case of the required restricted environments, and can appeal to Lemma 3.13. The claim hence follows by the UC security of $\mathsf{Compute}$, together with a standard hybrid argument.

□

Hybrid 4. (Robustness of secret sharing).

In this hybrid, the sequence of all committee communications and ideal functionality executions is replaced by a *single* ideal functionality $\mathcal{F}^{ss,\mathsf{addr}}_{\mathsf{Dyn}}$ which accepts inputs x_i from each party P_i, and then computes on these values directly, as dictated by the protocol. The ideal functionality outputs precisely the values dictated by the previous hybrid. The only difference is that, in the process of computing, $\mathcal{F}^{ss,\mathsf{addr}}_{\mathsf{Dyn}}$ continues to use the "true" reconstructed values, instead of accepting secret shares of these values as inputs from parties, reconstructing the values from the secret shares, and computing on these values in each step.

Note that $\mathcal{F}_{\mathsf{Dyn}}^{ss,\mathsf{addr}}$ performs the same functionality as the final desired $\mathcal{F}_{\mathsf{Dyn}}$, but also "leaks" intermediate secret share values and data access patterns.

Claim 3.21. *For every adversary $\mathcal{A}_3$ in Hybrid 3, there exists an adversary $\mathcal{A}_4$ in Hybrid 4 such that for any environment $\mathcal{Z}$,* $\mathsf{Hyb}_3\left(1^k, \mathcal{A}_3, \mathcal{Z}\right) \cong \mathsf{Hyb}_4\left(1^k, \mathcal{A}_4, \mathcal{Z}\right)$.

Proof. The claim holds directly by the robust reconstruction property of the secret sharing scheme: namely, secret shares of malicious parties (who form a $< \frac{1}{3}$ minority of each committee) are irrelevant to the secret share reconstruction process, since honest parties will all hold consistent shares of the correct value. Thus, since honest parties faithfully follow the protocol as instructed, the "true" reconstructed values at each stage of computation will be equal to the values used within this hybrid.

Namely, for any adversary $\mathcal{A}_3$ in Hybrid 3, the claim holds for the corresponding adversary $\mathcal{A}_4$ in Hybrid 4 who simply submits bogus secret shares to the new ideal functionality on behalf of malicious parties (say, all 0s), and simulates the actions of $\mathcal{A}_3$ consistent with the resulting ideal functionality outputs.

□

Hybrid 5. (Secrecy of secret sharing).

Same as the previous hybrid, except that the ideal functionality $\mathcal{F}_{\mathsf{Dyn}}^{ss,\mathsf{addr}}$ is replaced by a more restricted functionality $\mathcal{F}_{\mathsf{Dyn}}^{\mathsf{addr}}$ which accepts the same inputs, and performs the same computation, but *no longer outputs intermediate secret shares.* That is, $\mathcal{F}_{\mathsf{Dyn}}^{\mathsf{addr}}$ outputs the same values as the desired functionality $\mathcal{F}_{\mathsf{Dyn}}$ together with the corresponding data access patterns during computation (i.e., $(\mathsf{addr}_t, \mathsf{addr}_t^{\mathsf{Work}}, \mathsf{party}_t)$ for each computation step i).

Claim 3.22. *For every adversary $\mathcal{A}_4$ in Hybrid 4, there exists an adversary $\mathcal{A}_5$ in Hybrid 5 such that for any environment $\mathcal{Z}$,* $\mathsf{Hyb}_4\left(1^k, \mathcal{A}_4, \mathcal{Z}\right) \equiv \mathsf{Hyb}_5\left(1^k, \mathcal{A}_5, \mathcal{Z}\right)$.

Proof. The lemma claim by the secrecy property of the secret sharing scheme. That is, since malicious parties only ever see a $< \frac{1}{3}$ minority of shares of any secret value, the distribution of these shares will simply be uniform, independent of the corresponding secret values.

Formally, fix an adversary $\mathcal{A}_4$ in Hybrid 4. Consider the adversary $\mathcal{A}_5$ in Hybrid 5 who simulates the output of $\mathcal{F}_{\mathsf{Dyn}}^{ss,\mathsf{addr}}$ by providing the output of his own ideal functionality $\mathcal{F}_{\mathsf{Dyn}}^{\mathsf{addr}}$ (which outputs everything except intermediate secret shares), and simulates the additional sets of secret shares by sampling *uniform* shares. By the perfect secrecy of the secret sharing scheme, the simulated output of $\mathcal{F}_{\mathsf{Dyn}}^{ss,\mathsf{addr}}$ by $\mathcal{A}_5$ in Hybrid 5 is the *exact* same distribution as the true output of $\mathcal{F}_{\mathsf{Dyn}}^{ss,\mathsf{addr}}$ in Hybrid 4. Thus, the outputs of the two experiments are identical. □

Hybrid 6. (ORAM Security).

The ideal-world experiment. That is, the ideal functionality $\mathcal{F}_{\mathsf{Dyn}}^{\mathsf{addr}}$ from the previous hybrid is replaced by the desired functionality $\mathcal{F}_{\mathsf{Dyn}}$ which performs the same computation but no longer reveals the data access patterns.

Claim 3.23. *For every adversary $\mathcal{A}_5$ in Hybrid 5, there exists an adversary $\mathcal{A}_6$ in Hybrid 6 such that for any environment $\mathcal{Z}$,* $\mathsf{Hyb}_5\left(1^k, \mathcal{A}_5, \mathcal{Z}\right) \cong \mathsf{Hyb}_6\left(1^k, \mathcal{A}_6, \mathcal{Z}\right)$.

Proof. The claim holds by the security of the ORAM compiler.

Recall that (without loss of generality) the input RAM program Π makes an equal number of memory accesses to each of its two memory databases (Input / Permanent memory, and its Temporal Work Tape), one access from each per computation step. Indeed, this corresponds to first compiling an arbitrary Π via an intermediate naïve ORAM complier on two data blocks, which simply accesses both blocks each step of computation. It thus remains to simulate the data access patterns of each of these two individual blocks of memory (corresponding to the two ORAM structures) independently.

But, by the statistical obliviousness property of the ORAM, it holds that the distribution of data accesses $(\mathsf{addr}_1, \dots, \mathsf{addr}_q)$ to the Input (respectively, Work Tape) ORAM data structure as dictated by the ORAM-compiled program Π' is statistically close to the corresponding distribution $(\mathsf{addr}'_1, \dots, \mathsf{addr}'_q)$ formed by running the ORAM-compiled program Π' with respect to a *dummy* database, populated by all 0s. We refer to the process of simulating this execution on dummy inputs and retrieving the corresponding vector of data access addresses $(\mathsf{addr}'_1, \dots, \mathsf{addr}'_q)$ as "sampling" from the distribution D'. Thus, for any adversary $\mathcal{A}_5$ in Hybrid 5, we construct a corresponding adversary $\mathcal{A}_6$ in Hybrid 6, who simulates the output of $\mathcal{F}_{\mathsf{Dyn}}^{\mathsf{addr}}$ by combining the output of his own ideal functionality $\mathcal{F}_{\mathsf{Dyn}}$, with simulated data access pattern information sampled from $\mathcal{D}'$. Since the simulated distribution is statistically close to the corresponding real output of $\mathcal{F}_{\mathsf{Dyn}}^{\mathsf{addr}}$, the claim follows.

$\square$

$\square$

4 Load-Balanced Secure Evaluation of Dynamic RAM Functionalities

Observe that while our protocol in the previous section achieves memory balancing, both communication and computational complexity of the parties are highly unbalanced. In particular, the parties in the CPU committee as well as the ORAM root node committees are constantly active throughout the protocol execution and perform significantly more work than other parties. In this section, we address this issue and modify our protocol such that the total communication and computational complexity is preserved, while additionally achieving a strong load balancing property—with high probability, throughout the protocol execution, each party performs close to $1/n$ fraction of current total work, up to an additive $\mathsf{polylog}(n)$ amount of work, for both communication and computational complexity. Formally, we prove the following theorem.

Theorem 4.1 (Load-Balanced MPC for Dynamic RAM Functionalities)**.** *For any constant $\epsilon, \delta > 0$, there exists an n-party statistically secure (with error negligible in n) protocol that UC realizes the functionality $\mathcal{F}_{\mathsf{Dyn}}$ for securely computing a sequence of RAM programs Π_j, as described in Figure 2, handling $(1/3 - \epsilon)$ static corruptions, and with the same per-party memory, total computation and communication complexity, and round complexity as specified in Theorem 3.1 and additionally satisfying the following strong* on-line load balancing property *for per-party communication and computation complexity.*

- *With all but negligible probability in n, the following holds at* all times *during the protocol: Let* cc *and* $\mathsf{cc}(P_i)$ *denote the total communication complexity and communication complexity of* *party i, and* time *and* $\mathsf{time}(P_i)$ *denote the total time complexity and time complexity of party i, we have*

$$\frac{(1-\delta)}{n}\mathsf{cc} - \mathsf{polylog}(n) \leq \mathsf{cc}(P_i) \leq \frac{(1+\delta)}{n}\mathsf{cc} + \mathsf{polylog}(n)$$
$$\frac{(1-\delta)}{n}\mathsf{time} - \mathsf{polylog}(n) \leq \mathsf{time}(P_i) \leq \frac{(1+\delta)}{n}\mathsf{time} + \mathsf{polylog}(n).$$

In particular, the per-party communication and computation complexities are each improved from $\tilde{O}(|x| + |y| + \sum_j |\Pi_j|)$ to $\tilde{O}(|x| + |y| + \sum_j |\Pi_j|/n)$.

It can be shown that our protocol from Section 3 is already load balanced in both the committee election and input insertion phases. We thus focus on the Computation phase. Recall that our protocol is committee-based, where each committee is elected to perform one specific role of an ORAM program and the committees jointly emulate the underlying ORAM program and perform output delivery. Note that the output delivery in the Computation Phase is load balanced up to a constant factor. We can achieve $(1 \pm \delta)$ factor load balancing by properly padding. Therefore, it remains to focus on achieving load balancing for the ORAM-compiled program execution, which consists of sequences of invocations to the $\mathcal{F}_{\mathsf{Command}}^{\mathcal{E},\mathcal{E}'}$ and $\mathcal{F}_{\mathsf{Comp}}^{\mathcal{E}}$ ideal functionalities to execute the corresponding sequence of computation steps and data accesses.

Our main idea is a very natural one—we simply let each committee pass its job to a new committee when it performs "enough work" for this job, which we call *job-passing* scheduling algorithm. Note that crucially, each committee of our protocol has $\mathsf{polylog}(n)$ parties and only needs to hold a small $\mathsf{polylog}(n)$-size (secret) state (either the CPU state or the memory content). Thus, one committee C_1 can pass its job to a new committee C_2 by simply "sending the state" to C_2, using a generic MPC protocol that on input shares of the state from each party of C_1 output fresh shares of the state for each party of C_2, while only incurring $\mathsf{polylog}(n)$ cost in both communication and computational complexity. By the UC security of the MPC protocol and the secrecy property of secret sharing schemes, as long as there are less than one-third adversaries for both committees, the adversary learns no information about the underlying state information theoretically, and hence the protocol remains secure. Additionally, note that for this reason there is no security issue for a committee to take multiple roles at the same time.

This leads us to the following natural framework: We instead elect a fixed set of *worker* committees, assign each job/role of the ORAM program to some worker (where one worker can take multiple roles at the same), and switch the workers for each job throughout the protocol to achieve load balancing. The goal is to design a scheduling algorithm for assigning jobs to workers so that the load is balanced for both communication and computational complexity simultaneously.

However, there are two issues need to be addressed. First, note that the above framework considers load balancing on the committee level, instead of for each party. Nevertheless, note that the workload of each party is balanced inside each committee, and if there are sufficiently many worker committees (e.g., $N \geq n \cdot \mathsf{polylog}(n)$ worker committees), each of which consisting of a *random* set of $\mathsf{polylog}(n)$ parties, then each party will participate in roughly the same number of worker committees (as proved in Section 3.1), and hence load balancing on the committee level implies load balancing over parties as well.

Secondly, recall that in our previous protocol, the assignment of committees to "jobs" was public, static information, so the CPU committee knew which parties to talk to for each ORAM tree node. In our framework, each role of the ORAM program is dynamically assigned to a different worker committee over time. We thus need a (distributed) mechanism for the CPU worker to find right workers to talk to, without blowing up the memory complexity of parties. For example, the CPU worker cannot afford to store all this information itself. We address this issue by taking the advantage of the access pattern of the tree-based ORAM we use. Specifically, note that in the ORAM-compiled program, the CPU always traverses the ORAM data tree from the root to a random leaf. To enable this sequence of accesses within the protocol, we simply let the CPU worker keep track of the workers for each ORAM tree root (i.e., one root per recursion level in the ORAM structure; see Section 2.1), each ORAM tree node worker keep track of the workers of its two children nodes, and update the link dynamically. It is not hard to see that the update can be done efficiently (as a simple data structure exercise) and each ORAM role only need to keep $\mathsf{polylog}(n)$ bits of information. Note that maintaining the link incurs extra cost for passing the jobs, which creates subtleties in achieving load balancing. It turns out to be crucial for us to exploit the specific access pattern of the tree-based ORAM and store *only* "one-way link" to achieve load balancing with a simple job-passing algorithm. We discuss further details in Section 4.2.

Therefore, the question reduces to the above scheduling problem for load balancing both communication and computational complexity simultaneously. We study this problem in a clean abstract model in the next section, where we first focus on load balancing a single "*nice*" base cost metric, and show that our job-passing scheduling algorithm is *robust* in that it also achieves load balancing for any "related" cost metric. We then show in Section 4.3 that by properly choosing the base cost metric, we can achieve load balancing for both communication and computational complexity simultaneously.

4.1 An Abstract Load Balancing Model

In this section, we consider a load-balancing problem for assigning jobs in the following abstract model. We focus on analyzing a natural *job-passing* scheduling algorithm S in this model, defined with respect to a certain "*nice*" base cost metric CP. We show that S not only achieves load balancing for CP, but also for a class of cost metrics that satisfy certain relations with CP. We will use the job-passing scheduling algorithm S with a properly chosen base cost metric to achieve load balancing for both communication and computational complexity simultaneously.

Consider that there are M jobs $J_1, \dots, J_M$ and N workers $W_1, \dots, W_N$ in the following process. The process proceeds in rounds for $T = \mathsf{poly}(N)$ rounds. At each round, some jobs are activated, which is specified by $\mathsf{Act}(\cdot,\cdot)$. Namely, $\mathsf{Act}(t,i) = 1$ iff job J_i is activated at round t, and otherwise, $\mathsf{Act}(t,i) = 0$. When a job is activated, it may incur some activation cost (specified later). Throughout the process, we need to assign each job to a worker, which is specified by $\mathsf{Ass}[\cdot]$. Namely, job J_i is assigned to worker W_j iff $\mathsf{Ass}[i] = j$. We assume that the job activation pattern is independent of the job assignment. There is no restriction on the number of jobs assigned to the same worker. A *scheduling algorithm* can initially assign jobs arbitrarily without any cost, and in between rounds, update the job assignment arbitrarily, but which may incur some switch cost (specified below).

There may be multiple cost metrics associated with the process. A cost metric is specified by a cost profile $\mathsf{CP} = (c, sc_i, \beta_i, \gamma_j)$ described as follows.

- $c(\cdot)$ is a cost vector that specifies the cost of each job when it is activated. Namely, at each round t, job i incurs a cost of $\mathsf{Act}(t,i) \cdot c(i)$ charged to the assigned worker $\mathsf{Ass}[i]$.
- For each $i \in [M]$, $sc_i(\cdot)$ and $\beta_i \in [0,1]$ together specify the switch cost of job i. More precisely, switching the assigned worker for job i may incur cost not only for job i itself, but also for other

jobs (e.g., those jobs that help the job-switch), and $sc_i(i')$ is the cost of job i' incurred by the job-switch of J_i (again, charged to the assigned worker $\mathsf{Ass}[i']$) for every $i' \neq i$. Additionally, for job i itself, when J_i is switched from worker W_j with $j = \mathsf{Ass}[i]$ to worker $W_{j'}$, the switching cost incurred by job J_i is split between W_j and $W_{j'}$ with ratio specified by β_i. Namely, $\beta_i \cdot sc_i(i)$ and $(1-\beta_i)\cdot sc_i(i)$ cost are charged to W_j and $W_{j'}$, respectively.

- In most of cost metrics we consider, the activation cost incurred by job J_i is charged to the assigned worker $W_{\mathsf{Ass}[i]}$ at the time. We call such cost metrics *typical*. However, as required in the application from the next section (where we further achieve communication locality), we consider a more general scenario that when some job J_i assigned to worker W_j (i.e., $\mathsf{Ass}[i] = j$) incurs some cost x, the cost is not charged to W_j directly, but may further be distributed to all workers in a certain randomized way (for example, W_j may ask its nearby workers to help with the job) depending on the worker W_j. We specify this information by a random variable $\gamma_j(\cdot)$ that satisfies $\sum_{j'} \gamma_j(j') = 1$ with probability 1, where each time when some x cost is incurred with assigned worker W_j, an independent sample of $\gamma_j(\cdot)$ is drawn, and each worker $W_{j'}$ is charged $\gamma_j(j')\cdot x$ cost. The case of typical cost metric corresponds to γ_j's being deterministic vectors with $\gamma_j(j') = 1$ iff $j = j'$.

Intuitively, the ideal goal is to schedule the jobs in the way so that for all cost metrics, the cost is (i) *load-balanced* in the sense that every worker is charged close to $1/N$-fraction of total cost throughout the process, and (ii) with *small overhead* in the sense that the switch cost is small relative to the activation cost.

We here focus on analyzing a natural randomized *job-passing* scheduling algorithm S defined with respect to certain *nice* cost metric CP and a threshold parameter τ. Roughly, S initially assigns each job to an independent uniformly random worker, and whenever the incurred activation cost of a job to its assigned work exceeds the threshold τ, S reassigns the job to a new random worker. We show that if CP satisfies certain nice properties, then S achieves load balancing for CP with small overhead (formalized properly). We further show that while S is defined with respect to CP, it also achieves load balancing for a large class of cost metrics related to CP *simultaneously*. Looking forward, this allows us to load balance multiple cost metrics simultaneously by using S with respect to a properly chosen cost metric.

We proceed to the formal treatment. We focus on *nice* cost metric $\mathsf{CP} = (c, sc_i, \beta_i, \gamma_j)$ defined as follows.

Definition 4.2. A cost metric $\mathsf{CP} = (c, sc_i, \beta_i, \gamma_j)$ is a *nice* cost metric if it satisfies the following properties. (i) CP is typical, as defined above, (ii) $c(i) = 1$ for all $i \in [M]$, (iii) for every $i \in [M]$, $\beta_i \cdot sc_i(i) = (1-\beta_i)\cdot sc_i(i) = 1$ and $sc_i(i') \in \{0,1\}$ for $i' \neq i$ (that is, switch cost for both old and new workers of the switched job is 1, and for other jobs is either 0 or 1), and (iv) *activation-dominance* property: for every $i \in [M]$, in between any two activations of J_i there is at most one activation of some job $J_{i'}$ with $sc_{i'}(i) = 1$, and there is no activation of such $J_{i'}$ before the first activation of J_i.

Focusing on nice cost metrics (in particular, one with the activation-dominance property), and identifying such a cost metric in our protocol is the key for us to achieve load balancing with the following simple job-passing algorithm.

Let $\tau \in \mathbb{N}$ be a threshold. We define a randomized *job-passing* scheduling algorithm S with respect to CP and τ as follows.

- Initially, S assigns each job J_i to a random worker W_j. Namely, $\mathsf{Ass}[i] \leftarrow_R [N]$ for every $i \in [M]$.

- For each $i \in [M]$, S keeps track of the accumulated activation cost of J_i for the currently assigned worker $W_{\mathsf{Ass}[i]}$, denoted by $a[i]$. That is, at each round $a[i] \leftarrow a[i] + \mathsf{Act}(t,i) \cdot c(i)$ for every $i \in [M]$.
- Whenever an $a[i]$ reaches τ, S switches the job J_i to a uniformly random worker and resets $a[i]$. That is, S updates $\mathsf{Ass}[i] \leftarrow_R [N]$ and sets $a[i] = 0$.

We call CP the *base* cost metric for S. We now show that S achieves load balancing for the cost metric CP. At a high level, the key observation here is that since CP is typical (i.e., cost of a job is charged directly to the assigned worker) and the activation pattern is independent of the worker assignments, the time for switching workers of each job is in fact *independent* of the assignment of the worker for each job. Therefore, we can re-interpret the process as that (i) first, S determines the time to switch jobs without assigning the workers, which partitions each job into multiple "*job-chunks*" where each job-chunk is supposed to be assigned to one (random) worker, and then (ii) assigns each job-chunk to an i.i.d. uniformly random worker. Now, each job-chunk consists of at most τ activation cost by construction, and one can show by the activation-dominance property of CP that each job-chunk consists of at most $\tau + 3$ switch cost (see proof of Lemma 4.3 for a formal argument). We can thus view the process as throwing balls with weight at most $O(\tau)$ into random bins, and by standard concentration bounds, each bin receives balls with roughly the same amount of total weight, which implies load balancing. This also shows that the overhead is small.

In order to state a formal lemma, we introduce the following notation to describe the (accumulated) individual/total cost throughout the process. Let (i) $\mathsf{cost}(t, J_i) = \mathsf{Act}(t,i) \cdot c(i)$, (ii) $\mathsf{cost}(t, W_j)$ and (iii) $\mathsf{cost}(t)$ denote the activation cost (i) incurred by job J_i, (ii) charged to worker W_j, and (iii) total activation cost at round t, respectively. Analogously, we use $\mathsf{tcost}(\cdot)$ to denote the corresponding total cost that sums up both activation and switch cost. Namely, $\mathsf{tcost}(t, J_i), \mathsf{tcost}(t, W_j)$, and $\mathsf{tcost}(t, W_j)$ denote the total cost of J_i, W_j and all jobs at round t, respectively. Finally, we let $\mathsf{Cost}, \mathsf{tCost}$ denote "accumulated" cost. Namely, $\mathsf{Cost}(t, \cdot) = \sum_{t' \in [t]} \mathsf{cost}(t', \cdot)$ and tCost is defined analogously. Thus, for example, $\mathsf{tCost}(T)$ is the total cost of the whole process, and $\mathsf{tCost}(t, W_j)$ is the total cost of W_j up to round t. We are ready to state our lemma.

Lemma 4.3. *Let* $\mathsf{CP} = (c, sc_i, \beta_i, \gamma_j)$ *be a nice cost metric defined above. Let* $\tau \geq 4 \in \mathbb{N}$ *be a threshold, and* $\delta > 0$ *a constant. Then the job-passing scheduling algorithm* S *with respect to* CP *and* τ *defined above satisfies the following property with respect to* CP*:*

- **Load Balancing:** *With all but negligible probability in* N*, throughout the process for every round* $t \in [T]$ *and every worker* W_j*,*

$$(1-\delta) \cdot \frac{\mathsf{tCost}(t)}{N} - 4\tau \cdot \log^2 N \leq \mathsf{tCost}(t, W_j) \leq (1+\delta) \cdot \frac{\mathsf{tCost}(t)}{N} + 4\tau \cdot \log^2 N. \quad (1)$$

- **Small Overhead:** *The total cost can be upper bounded by three times total activation cost, i.e.,*

$$\mathsf{tCost}(T) \leq 3\mathsf{Cost}(T).$$

Proof. For the load balancing property, it suffices to prove Eq. (1) for fixed t and W_j, and the property follows by a union bound over t and j. Fix a round $t \in [T]$ and a worker W_j. Note that by the independence assumption between job activation pattern and job assignment and the fact that CP is typical, we can view the process as following. (i) First the job activation pattern $\mathsf{Act}(\cdot,\cdot)$ throughout the process up to round t is determined, which in turn determines the switch pattern and switch cost. The switch pattern partitions the jobs into "job-chunks" $B_1, \dots, B_s$, where each job-chunk is assigned to a single worker by S. (ii) Then each job-chuck is assigned to an i.i.d uniform worker. Let $\mathsf{cost}(B_k)$ and $\mathsf{tcost}(B_k)$ denote the activation and total cost of job-chunk B_k, respectively, for every $k \in [s]$.

We claim that $\mathsf{tcost}(B_k) \le 2\tau + 3 \le 4\tau$ for every k. By construction, we have $\mathsf{cost}(B_k) = \tau$. Let J_i be the underlying job of B_k. Note that the switch of J_i incurs at most $(1-\beta_i)\cdot sc_i(i) + \beta_i \cdot sc_i(i) = 2$ switch cost in B_k (starting and end of B_k). Additionally, by the activation-dominance property of CP, the switch cost from switching other jobs $J_{i'}$ is at most $\tau + 1$, since there can be at most one switch cost incurred to J_i in between two activations of J_i.

Now, for every $k \in [s]$, define random variable $X_k = \mathsf{tcost}(B_k)/4\tau \in [0,1]$ if the job-chunk B_k is assigned to the fixed worker W_j, and 0 otherwise. Let $X = \sum_k X_k$ and $\mu = E[X]$. Clearly, we have $\mu = \mathsf{tCost}(t)/(4\tau N)$, since each job-chunk is assigned to one of N workers at random. By a standard Chernoff bound, we have that for $\delta' = \delta + \log^2 N/\mu$,

$$\Pr[X \ge (1+\delta)\mu + \log^2 N] \le \left(\frac{e^{\delta'}}{(1+\delta')^{1+\delta'}}\right)^{\mu} \le \left(\frac{e^{\delta}}{(1+\delta)^{(1+\delta)}}\right)^{\log^2 N} \le \mathsf{negl}(N),$$

which implies $\mathsf{tCost}(t, W_j) \le (1+\delta)\cdot \frac{\mathsf{tCost}(t)}{N} + 4\tau \cdot \log^2 N$. The other inequality follows analogously. A union bound over t and j then completes the proof of the load balancing property.

For the small overhead property, note again that for each job J_i, the switch cost of J_i incurred by the switch of J_i itself is at most $2/\tau$ fraction of the activation cost of J_i, and the switch cost of J_i incurred by the switch of other jobs is at most the activation cost by the activation-dominance property of CP. It follows that the total switch cost of J_i is at most $(2/\tau)\cdot \mathsf{Cost}(T, J_i) + \mathsf{Cost}(T, J_i) \le 2\mathsf{Cost}(T, J_i)$, which implies that $\mathsf{tCost}(T) \le \mathsf{Cost}(T) + 2\mathsf{Cost}(T) = 3\mathsf{Cost}(T)$.

□

We now state some sufficient properties of a cost metric CP' such that the job-passing scheduling algorithm S with respect to the above base cost metric CP also achieves load balancing for cost metric CP' as well. Let $\mathsf{CP}' = (c', sc'_i, \beta'_i, \gamma'_j)$ be a cost metric. We say that CP' is *α-bounded* by CP if (i) for every $i \in [M]$, $c'(i) \le \alpha \cdot c(i)$, and (ii) for every $i \in [M]$, $sc'_i(i') \le \alpha \cdot sc_i(i')$ for $i' \neq i$, $\beta'_i sc'_i(i') \le \alpha \cdot \beta_i sc_i(i')$ and $(1-\beta'_i)sc'_i(i') \le \alpha \cdot (1-\beta_i)sc_i(i')$. Namely, all the cost incurred in CP' is bounded by α times the corresponding cost incurred in CP. It is not hard to prove that if CP' is typical and α-bounded by CP for small α, then CP' is also load balanced by S as well.

We further consider *non-typical* cost metrics. Recall this means that the cost charged to an assigned worker W_j for a particular job is further distributed to other workers in a randomized way specified by γ_j's. We say that a cost metric CP' is *fair* if $E_{j \leftarrow [N]}[\gamma'_j(j')] = 1/N$ for every $j' \in [N]$. This means that the fraction of cost distributed from a *random* worker W_j to all workers $W_{j'}$ are the same. Note that a typical cost metric is also fair. We show in the following lemma that S achieves load balancing for any cost metric CP' that is fair and α-bounded by CP. Here, we extend the notation $\mathsf{cost}, \mathsf{tcost}, \mathsf{Cost}, \mathsf{tCost}$ to $\mathsf{cost}', \mathsf{tcost}', \mathsf{Cost}', \mathsf{tCost}'$ to describe the corresponding cost quantities with respect to CP'.

Lemma 4.4 (Robust Load Balancing). *Let CP, τ, S be specified as in Lemma 4.3. Let CP' be a cost metric that is fair and α-bounded by CP for some $\alpha > 0$. Let $\delta > 0$ be a constant. We have with all but negligible probability in N, throughout the process for every round $t \in [T]$ and every* *worker W_j,*

$$(1-\delta)\cdot\frac{\mathsf{tCost}'(t)}{N} - 4\alpha\tau\cdot\log^2 N \le \mathsf{tCost}'(t, W_j) \le (1+\delta)\cdot\frac{\mathsf{tCost}'(t)}{N} + 4\alpha\tau\cdot\log^2 N. \tag{2}$$

Proof. As before, it suffices to prove the lemma for fixed $t \in [T]$ and worker W_j and apply a union bound. Recall in the proof of Lemma 4.3, we argue that the process can be viewed as first partitioning the jobs into job-chunks $B_1, \ldots, B_s$, and then assigning each job-chuck to an i.i.d uniform worker. For every $k \in [s]$, let $\mathsf{tcost}'(B_k)$ (resp., $\mathsf{tcost}'(B_k, W_j)$) denote the total cost of job-chunk B_k (resp., total cost of B_k charged to W_j). By the α-bounded property, we have $\mathsf{tcost}'(B_k) \le \alpha\mathsf{tcost}(B_k)$. Note that for non-typical CP', $\mathsf{tcost}'(B_k, W_j)$ is a random variable depending on the assigned worker $W_{j'}$ of the job-chunk B_k and the corresponding random variable $\gamma_{j'}$. Nevertheless, since B_k is assigned to a random worker, we know that $E[\mathsf{tcost}'(B_k, W_j)] = \mathsf{tcost}'(B_k)/N$.

Define $X_k = \mathsf{tcost}'(B_k, W_j)/(4\alpha\tau) \in [0,1]$. Let $X = \sum_k X_k$ and $\mu = E[X]$. By the fair property, we have $\mu = \mathsf{tCost}'(t)/(4\alpha\tau N)$. By a standard Chernoff bound, we have that let $\delta' = \delta + \log^2 N/\mu$,

$$\Pr[X \ge (1+\delta)\mu + \log^2 N] \le \left(\frac{e^{\delta'}}{(1+\delta')^{1+\delta'}}\right)^{\mu} \le \left(\frac{e^{\delta}}{(1+\delta)^{(1+\delta)}}\right)^{\log^2 N} \le \mathsf{negl}(N),$$

which implies $\mathsf{tCost}'(t, W_j) \le (1+\delta)\cdot\frac{\mathsf{tCost}'(t)}{N} + 4\alpha\tau\cdot\log^2 N$. The other inequality follows analogously. □

4.2 Protocol Construction

In this section, we discuss how to modify our protocol $\mathsf{MPC\text{-}Dyn}$ from the previous section to achieve load balancing. We first provide an outline of the modification and then present further details. We choose to keep our presentation high level and slightly informal for clarity of exposition.

Following the aforementioned framework, we elect worker committees to perform the jobs/roles of the dual ORAM program, and let the worker committees implement the job-passing scheduling algorithm (in a distributed fashion) described above to achieve load balancing. (We will use the terms jobs and roles of ORAM interchangeably in this section.) The crux here is to carefully implement the job-passing mechanism so that the worker committees can execute the job-passing scheduling algorithm with respect to a *nice* base cost metric specified in the previous section. We denote the modified protocol $\mathsf{LB\text{-}MPC\text{-}Dyn}$.

1. In the Committee Setup Phase, we additionally set up a set of Worker committees W_j's of size $N = n \cdot \mathsf{polylog}(n)$.
2. At the end of the Input Initialization Phase, we assign each initialized role of the dual ORAM program Π' to a random worker committee—this includes the CPU and the nodes in the ORAM trees of the Input tape. On the other hand, for the nodes of the Work tape ORAM, we initialize them on-the-fly since there are super-polynomially many nodes. Instead, we will initialize each Work tape ORAM node when it is accessed by the CPU.
3. The worker committees executes the roles of the dual ORAM program assigned to them. To enable the worker committees to find the right worker committee to talk to (for the roles they are assigned to), we let each role stores additional (public) *communication links* to the workers assigned to their "neighbours." Specifically, the CPU stores links to the roots of all

ORAM trees (including all recursion levels) for both Input and Work tape ORAM, and each node stores links to its both children. Such one-way link information suffices since the access pattern of dual ORAM program only consists of the CPU traversing each of the ORAM trees from the root to a (random) leaf, and as we shall see, the access pattern and the fact that we only maintain *one-way* links is the key for us to obtain a nice cost metric for the job-passing scheduling algorithm.

4. The worker committees jointly implement the job-passing scheduling algorithm with respect to a properly chosen nice base cost metric CP and threshold τ specified in the next bullet. For a worker committee W to pass a job (a role in the dual ORAM) J to another worker committee W', we need to (1) let W "send" the internal state of J to W' and (2) update the relevant communication links. We perform both steps through (properly extended) ideal functionalities.

   The crucial point here is to make sure that the job switch incurs switch cost to fewer jobs so that the activation-dominance property of a nice cost metric can be satisfied. For part (1), we simply let W send the state of J to W' using the Command ideal functionality $\mathcal{F}^{\mathcal{E},\mathcal{E}'}_{\mathsf{Comp}}$. For part (2), since we only store one-way links, we only need to update the link of J's "parent." Specifically, for a non-root node of a ORAM tree, we only need to update its parent node, for the root node, we need to update CPU's link to the root, and for the CPU, no update is needed. One question here is that if we only store one-way links, how can J find its "parent" job? Here we (again) rely on the special access pattern of the dual ORAM program that only consists of the CPU traversing each of the ORAM trees from the root to a (random) leaf. Thus, when a node role J is activated by the CPU, the CPU knows J's parent and can pass the information to J. Now if job J needs to be switched, J (or its assigned worker) can tell to its parent to update the link directly. By doing so, switching a job J only incurs switch cost to J itself and its "parent."

5. We now define a typical base cost metric $\mathsf{CP} = (c, sc_i, \beta_i, \gamma_j)$ and argue that it satisfies the nice properties we need. We count each invocation of ideal functionalities $\mathcal{F}^{\mathcal{E},\mathcal{E}'}_{\mathsf{Command}}$ or $\mathcal{F}^{\mathcal{E}}_{\mathsf{Comp}}$ as one unit activation cost (note that these are the only two ideal functionality invoked for ORAM program execution). In other words, each invocation of an ideal functionality is counted as one round of the scheduling process, which activates at most two jobs, one of which is the CPU. For a job J_i, let $J_{i'}$ be its "parent" job (that is, for a non-root node, $J_{i'}$ is its parent, for a root node, $J_{i'}$ is the CPU, and for the CPU, it has no parent). We set $\beta_i = 0.5, sc_i(i) = 2$ (that is, switch of job J_i takes on unit cost to both original and new worker), $sc_i(i') = 1$ (one unit cost for its "parent" job), and $sc_i(i'') = 0$ for all other i''. Finally, the typical property determines γ_j's.

   We now check that CP is a nice cost metric. Clearly, the first three nice properties follows by definition. For the activation-dominance property, first note that the CPU is activated in every round, so the property holds for the CPU. For each ORAM node role J_i, note that only its two child nodes $J_{i'}$ with $sc_{i'}(i) = 1$. Recall that the access pattern of dual ORAM program only consists of the CPU traversing each of the ORAM trees from the root to a (random) leaf. Thus, at most one child of J_i is activated once in between any two activations of J_i, and the activation-dominance property is satisfied.

6. Finally, for output delivery, note that although the CPU role is executed by some worker W_j as opposed to a fixed CPU committee C, we can still use the same output delivery procedure

as defined in Figure 9, starting by W_j sending the output value to 0 and 1 committees. By inspection, the output delivery procedure is already load-balanced up to a constant factor. We simply use a proper padding to make output-delivery load balanced.

We now elaborate on further details of our modification.

Committee Setup Phase. The protocol is identical as in Figure 3, except that C additionally choose Worker committees W_j for $j \in [N]$ with $N = nK\mathsf{polylog}(n) = n \cdot \mathsf{polylog}(n)$ in the same way as choosing the ORAM memory node committees. Namely, C executes a coin tossing (or generic MPC) protocol to sample a seed $s^{\mathsf{Worker}} \leftarrow \{0,1\}^{K^2 \log^2 n}$ for the K-wise independent function family $\{F_s\}$. Each committee W_i is implicitly defined by $W_j = F_{s^{\mathsf{Worker}}}(j)$ (where repetitions are removed). Then each $P_i \in C$ broadcasts the seed s^{Worker} to all parties. The follow lemma analogous to Lemma 3.2 for the Worker committees follows by an identical analysis.

Lemma 4.5. *Suppose access to broadcast channel, and coin tossing protocol* $\mathsf{CoinToss}$*. For any constant* $\epsilon, \delta > 0$*,* $(\frac{1}{3} - \epsilon)$ *fraction of static malicious corruptions* $M \subset [n]$*. Then with overwhelming probability in* n*, at the conclusion of* $\mathsf{ComSetup}$*, all parties agree on Worker committees* $\{W_i\}_{i \in N}$ *such that:*

1. *For every* $j \in [N]$*,*
$$\frac{|W_j \cap M|}{|W_j|} \leq \frac{1}{3} - \frac{\epsilon}{2}.$$

2. (Load Balancing) *For every party* $P_i \in [n]$*,*
$$(1-\delta)\mu \leq \left|\{j \in [N] : P_i \in W_j\}\right| \leq (1+\delta) \cdot \mu,$$
where $\mu := \left(1 - (1 - 1/n)^K\right) \cdot N$ *is the expected ("fair share") number of committees in which each party occurs.*

Input Initialization Phase. The Input Initialization Phase protocol is identical to the previous section, but appended with the following Initial Worker Assignment procedure.

- The CPU committee C and each of the Input ORAM committee C_j sample a random (public) worker index in $[N]$ to be its initial assigned worker committee (via a committee-wise coin tossing protocol).
- Each of the above committees sends the sampled index to its parent committee, where the root node committee sends the sampled index to the CPU committee (via the Command ideal functionality $\mathcal{F}^{\mathcal{E},\mathcal{E}'}_{\mathsf{Command}}$), and stores the received value as a link vector, denoted by link (namely, the CPU committee C stores link and each of the Input ORAM committee C_j stores link_j).
- The CPU committee C additionally samples random indices in $[N]$ for each root node of the ORAM Work tape, and append the samples to its link vector link.
- Define $\mathsf{content} = ([\mathsf{state}]_C, \mathsf{link})$ to be the content of the CPU committee C, where $[\mathsf{state}]_C$ is the internal state of the CPU, and $\mathsf{content}_j = ([v_j]_{C_j}, \mathsf{link}_j)$ be the content of the Input ORAM committee C_j, where $[v_j]_{C_j}$ is the memory content of C_j.
- Each of the above committees sends its content to its sampled Worker committee via $\mathcal{F}^{\mathcal{E},\mathcal{E}'}_{\mathsf{Command}}$.

Note that at the end of the protocol, the whole dual ORAM structure is assigned to the Worker committees, where each role is assigned to a random worker. Both the CPU committee C and Input ORAM committees C_j's will not participate in the protocol execution in the Computation Phase and thus can release their memory.

Computation Phase. We modify the protocol in the Computation Phase so that the Worker committees take over the $\mathsf{Compute}$ execution completely. Namely, the worker for each role of the ORAM takers over the job of the corresponding CPU and Input committees. Recall that in our ORAM program, the CPU always traverses all ORAM trees from the root to a random leaf, the CPU worker traces which workers to talk to through the link information stored in the content of each role.

Recall that the nodes of ORAM Work tape is not initialized before the Computation Phase, but the worker for each root of ORAM tree for the Work tape is chosen and stored in the link vector of the CPU content. Each node will be initiated when the CPU worker first talk to it. More precisely, when the CPU worker initiates an Access Command to an uninitiated ORAM Work tape node u by $\mathcal{F}^{\mathcal{E},\mathcal{E}'}_{\mathsf{Command}}$ to a Worker committee W_j, W_j initiates the node with content $\mathsf{content}_u = ([0], \mathsf{link}_u)$ where link_u consists of two random worker indices in $[N]$ that assign the workers of its two children (note that this is incorporated in the operation/instruction op of the Command ideal functionality).

For output delivery, note that although the CPU role is executed by some worker W_j as opposed to a fixed CPU committee C, we can still use the same output delivery procedure as defined in Figure 9, starting by W_j sending the output value to 0 and 1 committees. By inspection, the output delivery procedure is already load-balanced up to a constant factor. We simply use a proper padding to make output-delivery load balanced.

To achieve load balancing for ORAM program execution, the worker committees implements the job-passing scheduling algorithm with respect to the base cost metric CP defined above and threshold $\tau = 4$ (an arbitrary constant). Specifically, the workers keeps track of the number of invocations of ideal functionalities $\mathcal{F}^{\mathcal{E},\mathcal{E}'}_{\mathsf{Command}}$ and $\mathcal{F}^{\mathcal{E}}_{\mathsf{Comp}}$ for each of their assigned roles. When the number of a role J_i reach τ, the assigned worker W_j samples a random worker $W_{j'}$ (via a coin-tossing protocol) and send the state of J_i to $W_{j'}$ using $\mathcal{F}^{\mathcal{E},\mathcal{E}'}_{\mathsf{Command}}$ functionality (recall that the state is of $\mathsf{polylog}(n)$ size). If J_i is a non-CPU role, W_j also updates the worker $W_{j''}$ of J_i's "parent" role $J_{i'}$ the link information about J_i for $J_{i'}$. In order for W_j to know which worker $W_{j''}$ to talk to, we let the CPU worker to keep track of this information and send the information to W_j when talk to W_j in the invocation of $\mathcal{F}^{\mathcal{E},\mathcal{E}'}_{\mathsf{Command}}$ (which is the reason for the activation cost of J_i to reach τ). Note that only the activation cost is counted for job-passing (but not the switch cost). Also note that each invocation of $\mathcal{F}^{\mathcal{E},\mathcal{E}'}_{\mathsf{Command}}$ and $\mathcal{F}^{\mathcal{E}}_{\mathsf{Comp}}$ involves at most two worker committees, and one of which is the CPU worker. We extend the functionality so that both committees knows whether the threshold is reached for one or both of them. If both are reached, we let the non-CPU one to perform the job switch first and then the CPU one.

4.3 Analysis

We now prove Theorem 4.1 by analysing the protocol $\mathsf{LB\text{-}MPC\text{-}Dyn}$ constructed in the previous section.

Proof. First note that the security of $\mathsf{LB\text{-}MPC\text{-}Dyn}$ follows in an identical way to the security proof of Theorem 3.1, since by Lemma 4.5, all Worker committees consist of more than two-third of honest party with overwhelming probability, and our modification only invokes ideal functionalities with secret information operated under ideal functionalities that produce fresh secret shares, which preserves the secrecy information theoretically.

Observe that $\mathsf{LB\text{-}MPC\text{-}Dyn}$ and $\mathsf{MPC\text{-}Dyn}$ carry the same computation with the difference that (i) the computation in $\mathsf{LB\text{-}MPC\text{-}Dyn}$ is carried by the Worker committees, as opposed to the CPU, Input and Work tape committees, (ii) $\mathsf{LB\text{-}MPC\text{-}Dyn}$ has additional Initial Worker Assignment procedure, and (iii) the worker committees of $\mathsf{LB\text{-}MPC\text{-}Dyn}$ additional implements the job-passing scheduling algorithm with respect to the base cost metric CP defined in the previous section and threshold $\tau = 4$.

We first argue that the round complexity and total communication and computation complexity are preserved. Note that for this, it suffices to show that the job-passing scheduling algorithm does not incur too much switch cost in terms of these complexities. By Lemma 4.3, the switch cost is at most 2 times the activation cost when measured in the base cost metric CP. Note that in CP, we count both activation and switch costs by one unit cost, but the actual protocol execution of $\mathcal{F}^{\mathcal{E}}_{\mathsf{Comp}}$ and $\mathcal{F}^{\mathcal{E},\mathcal{E}'}_{\mathsf{Command}}$ takes some $\mathsf{polylog}(n)$ round, communication, and computation complexity. Nevertheless, since all are upper bounded by $\mathsf{polylog}(n)$, we can conclude that the round complexity and total communication and computational complexity are preserved up to an additive $\mathsf{polylog}(n)$ factor.

We next consider the memory complexity. The preservance total memory complexity follows by noting that the extra memory used in $\mathsf{LB\text{-}MPC\text{-}Dyn}$ is only that for storing the Worker committees and the additional links associated with each role of the ORAM program, and the preservance of per-party memory complexity follows by a similar argument to that in Theorem 3.1 that with overwhelming probability, each party participates in a number of committees that is $(1 \pm \delta)$ factor in its expectation, and that each worker is assigned to a number of jobs that is $(1 \pm \delta)$ factor in its expectation.

We proceed to prove the load balance property of the per-party communication and computation complexity. We focus on the ORAM program execution in the Computational Phase since by inspection, this is achieved in both Committee Setup Phase and Input Initialization Phase, as well as the output-delivery in the Computation Phase (by padding). We first show that the load is balanced on the worker committee level and argue that this implies that the load is balanced also on the per-party level. Note that in the Computation Phase, the protocol consists of sequence of invocations of $\mathcal{F}^{\mathcal{E}}_{\mathsf{Comp}}$, $\mathcal{F}^{\mathcal{E},\mathcal{E}'}_{\mathsf{Command}}$, and $\mathcal{F}^{\mathcal{E},i}_{\mathsf{Dec}}$, where the invocation of $\mathcal{F}^{\mathcal{E},i}_{\mathsf{Dec}}$ is for output delivery, which we do not consider for load balancing. The main observation is that both communication and computation complexity of $\mathcal{F}^{\mathcal{E}}_{\mathsf{Comp}}$ and $\mathcal{F}^{\mathcal{E},\mathcal{E}'}_{\mathsf{Command}}$ ideal functionalities are typical cost metrics that are $\mathsf{polylog}(n)$-bounded by the base cost metric CP. Indeed, both ideal functionalities have $\mathsf{polylog}(n)$ communication and computation complexity. Therefore, by the robust load balancing property from Lemma 4.4, both communication and computation complexity are load balanced on the committee level in the sense that with overwhelming probability throughout the protocol execution, each worker committee shares $(1 \pm \delta)/N$ faction of total work up to an additive $\mathsf{polylog}(n)$ term.

Now, we have that both complexities are load balanced on the committee level in the sense that with overwhelming probability throughout the protocol execution, each worker committee shares $(1 \pm \delta)/N$ faction of total work up to an additive $\mathsf{polylog}(n)$ term, and by Lemma 4.5, each party participate in $(1 \pm \delta)N$ number of worker committees. Therefore, with overwhelming probability, throughout the protocol execution, each party shares $(1 \pm 2\delta)/n$ fraction of total communication and computational complexity, up to an additive $\mathsf{polylog}(n)$ term, which completes the analysis. $\square$

5 Load-Balanced Secure Evaluation of Dynamic RAM Functionalities with $\mathsf{polylog}(n)$ Locality

In this section, we show how to further modify our protocol to achieve $\mathsf{polylog}(n)$-locality, where each party only need to talk to at most $\mathsf{polylog}(n)$ parties throughout the protocol execution, with an additional help of a single broadcast per party in the Committee Setup Phase. Formally, we prove the following theorem.

Theorem 5.1 (Load-Balanced MPC for Dynamic Functionalities with $\mathsf{polylog}(n)$-locality). *For any constant $\epsilon, \delta > 0$, there exists an n-party statistically secure (with error negligible in n) protocol that UC realizes the functionality $\mathcal{F}_{\mathsf{Dyn}}$ for securely computing dynamic functionalities Π handling $(1/3 - \epsilon)$ static corruptions, with identical properties as stated in Theorem 4.1. Additionally, except for the per-party single-use of a broadcast with $\mathsf{polylog}(n)$-bits messages, the protocol has $\mathsf{polylog}(n)$ communication locality.*

By inspection, there are two issues where our load-balanced protocol $\mathsf{LB\text{-}MPC\text{-}Dyn}$ from previous section does not have small communication locality:

- In the Input Initialization Phase, if the per-party input size is $|x| = \mathsf{polylog}(n)$, then the protocol has $\mathsf{polylog}(n)$ communication locality, since per-party complexity is $\mathsf{polylog}(n)$. However, if the input size is large, the locality becomes poor.
- In the Computation Phase, all $N = n \cdot \mathsf{polylog}(n)$ worker committees may communicate with each other since they play different roles over time.

For clarity of exposition, we first focus on addressing the second issue, assuming the input size is $|x| = \mathsf{polylog}(n)$. That is, we first focus on achieving both load balancing and locality for the dual ORAM program execution. We discuss how to handle large inputs in Section 5.2.

5.1 Making Computation Phase Local

Recall that the execution of the dual ORAM program in the Computation Phase consists of a sequence of invocation to the $\mathcal{F}^{\mathcal{E},\mathcal{E}'}_{\mathsf{Command}}$ and $\mathcal{F}^{\mathcal{E}}_{\mathsf{Comp}}$ ideal functionalities, where the worker committees jointly execute the assigned roles/jobs in the dual ORAM structure, and implement the job-passing scheduling algorithm to achieve load balancing. Since the $\mathcal{F}^{\mathcal{E}}_{\mathsf{Comp}}$ ideal functionality only involves a single committee, it clearly preserves the communication locality. On the other hand, the main issue comes from the $\mathcal{F}^{\mathcal{E},\mathcal{E}'}_{\mathsf{Command}}$ ideal functionality. In particular, note that the CPU worker committee needs to use the $\mathcal{F}^{\mathcal{E},\mathcal{E}'}_{\mathsf{Command}}$ ideal functionality to talk to all node roles in the dual ORAM structure, which breaks the communication locality. For convenience, let us call the CPU worker the source committee W_s and the other the destination committee W_t of $\mathcal{F}^{\mathcal{E},\mathcal{E}'}_{\mathsf{Command}}$.

Relying on the fact that the committees are in fact "honest agents," we can achieve communication locality readily by the following natural idea. We create a $\mathsf{polylog}(n)$-degree routing network G on the worker committees, and instead of having any two committees W_s and W_t talk to each other (using $\mathcal{F}^{\mathcal{E},\mathcal{E}'}_{\mathsf{Command}}$), we let them communicate through the routing network G by, say, "pass the command" using a sequence of $\mathcal{F}^{\mathcal{E},\mathcal{E}'}_{\mathsf{Command}}$ calls of neighboring committees in G along a path from s to t. Indeed, it is not hard to formalize this idea (we discuss further details later), and by doing so, we achieve $\mathsf{polylog}(n)$ communication locality throughout the protocol.

However, the main issue is that the protocol may no longer be load balanced. For example, it is conceivable that some committees may perform significantly larger amount of routing work than other committees. At a high level, achieving both communication locality and load balancing simultaneously via such routing network approach is significantly more challenging, since routing incurs "non-local" cost for multiple committees along a path that depends on both the source and destination committees.

Local load-balanced routing networks. To deal with the issue, we introduce a primitive called local load-balanced routing networks. Roughly speaking, we need to have a way to ensure that the routing network we use ensures that the message-passing in done in a load-balanced way. Of course, we can never hope to achieve this if the source and destination are not (individually) uniform, so we here focus only on a setting where this is the case. This suffices since our job-passing scheduling algorithm from the previous section implies that the source and destination are from uniformly random worker committees. More precisely, we want to ensure that if the source s and the destination t are individually uniform (but may be correlated), then the *expected* number of times a node is on the path we take between them should be balanced. This is a perhaps a seemingly weak load-balancing property in that we only balance the expectation, but allows us to define a *fair* cost metric that is $\mathsf{polylog}(n)$-bounded by the base cost metric used in the job-passing algorithm in the previous section, which implies load balancing by Lemma 4.4.

We next provide some intuition about how to design such a local load-balanced routing network. The general idea is as follows. We use a regular expander graph G as our communication network. To route a message from a source s to a destination t, we first take a random walk ("walking into the woods from s") of length $\omega(\log|G|)$ from s, reaching a node s'. Note that at this point, the routing is load-balanced (in expectation, as required): we start at a random node—since the graph is regular that means we are starting at the stationary distribution—and each time we take single random step on this graph we remain at the stationary distribution.

Additionally, by the mixing property of the expander graph, this ensures that we end up in a random node in G (independent of s). We then take another random walk of length $\omega(\log|G|)$, from s' this time *conditioned* on reaching t. Since as observed above, s' is an (essentially) uniform node, the distribution of the this second walk ("home from the woods") is statistically close to taking a length $\omega(\log|G|)$ random walk from t (i.e., "walking into the woods from t"). It thus follows exactly as before that also this second step is load-balanced in expectation.

A final issue with this approach, however, is that it is not clear how to implement the "walk back from the woods to t". In particular, recall that this requires picking a random walk conditioned on reaching t which may not be efficiently computable. We resolve this final issue by relying on a particular regular expander—namely the boolean hybercube—for which the conditional random walk can be efficiently found. In fact, for this specific expander we can simply take a random shortest path from s to s' and from s' to t which somewhat simplifies the analysis. This is reminiscent to the work of Valiant and Brebner [VB81], which performs a similar hybercube routing in a different context (that we will use in Section 5.2).

Implementing $\mathcal{F}^{\mathcal{E},\mathcal{E}'}_{\mathsf{Command}}$ through routing. We here briefly discuss further details on how to implement $\mathcal{F}^{\mathcal{E},\mathcal{E}'}_{\mathsf{Command}}$ by "passing commands" through a routing network G. Note that $\mathcal{F}^{\mathcal{E},\mathcal{E}'}_{\mathsf{Command}}$ involves a source committee W_s and a destination committee W_t with the semantic that W_s sends a (secret) operation $[\mathsf{op}]$ to W_t, who "performs" the operation and returns a value $[v_1]$ back to W_s. Thus, we can implement $\mathcal{F}^{\mathcal{E},\mathcal{E}'}_{\mathsf{Command}}$ by having W_s choose a path $p_{s,t} = (v_0 = s, v_1, \ldots, v_\ell = t)$ in G, send $[\mathsf{op}]$ to W_t through $p_{s,t}$, who sends back the returned value $[v_1]$ back to W_s through the reverse path. This in turn can be implemented by invoking $\mathcal{F}^{\mathcal{E},\mathcal{E}'}_{\mathsf{Command}}$ for each pair W_{v_i}, $W_{v_{i-1}}$ of neighboring committees along the path $p_{s,t}$ (with properly extended "operation set"). For clarity, we use $\mathcal{F}^{\mathcal{E},\mathcal{E}'}_{\mathsf{LocalCom}}$ to denote the command passing functionality, and say that each $\mathcal{F}^{\mathcal{E},\mathcal{E}'}_{\mathsf{Command}}$ is implemented by a sequence of $\mathcal{F}^{\mathcal{E},\mathcal{E}'}_{\mathsf{LocalCom}}$ invocations (along the path chosen by W_s). Note that now the $\mathcal{F}^{\mathcal{E},\mathcal{E}'}_{\mathsf{Command}}$ functionality is only invoked for neighbouring committees for the constant degree expander, and thus locality is achieved.

We introduce the notion of local load balanced communication network and present a simple construction in the next section, and prove Theorem 5.1 assuming input size $|x| = \mathsf{polylog}(n)$ in Section 5.1.2.

5.1.1 Local Load-Balanced Communication Network

In the next section, we introduce the notion of *local load-balanced communication network* that formally captures the load-balancing property and additional properties we need, and present a very simple construction based on Boolean hypercube graphs. Our construction is reminiscent to the work of Valiant and Brebner [VB81], which performs a very similar hybercube routing, but in a different context with a different analysis.

Definition 5.2. Let $G = \{G_n = ([n], E_n)\}_{n\in\mathbb{N}}$ be a graph family and A be a (uniform) routing algorithm A for G such that on input $n \in \mathbb{N}$ and a pair of nodes $s, t \in [n]$ outputs a path from s to t. We say that (G, A) form a *$d(n)$-local load-balanced communication network* if it satisfies the following properties.

Locality: For every $n \in \mathbb{N}$, the degree of G_n is $d(n)$.

Efficiency: For every input $n \in \mathbb{N}$ and $s, t \in [n]$, $A(n, s, t)$ has runtime $\mathsf{polylog}(n)$ and outputs a path of length $O(\log n)$.

Load Balancing: For every $n \in \mathbb{N}$ and $u, v \in [n]$,

$$E_{s,t\leftarrow[n]}[u \in A(n, s, t)] = E_{s,t\leftarrow[n]}[v \in A(n, s, t)].$$

We show that the family of Boolean hypercube graphs form a $\log(n)$-local load-balanced communication network. Specifically, for every $n = 2^d$, let $H_n = ([n], E_n)$ be the hypercube graph of dimension d. Namely, the vertex set $[n]$ is identified with $\{0,1\}^d$ and $(u, v) \in E_n$ iff the Hamming distance between u and v is 1. Let $H = \{H_n\}_{n=2^d}$ and consider the following natural routing algorithm A that routes s to t by taking a random path from s towards t as follows. On input $n = 2^d$, $s, t \in \{0,1\}^d$,

- Let $v_0 = s$ and let ℓ be the Hamming distance between s and t.
- For $i = 1, \ldots, \ell$, let $j \in [d]$ be a random coordinate such that v_{i-1} and t differs on the j-th bit, and set v_i to be v_{i-1} with j-th bit flipped. Namely, $v_i = v_{i-1} \oplus e_j$, where $e_j \in \{0,1\}^d$ is the string with j-th bit 1 and the remaining bits 0.

- Output $p_{s,t} = (v_0, \ldots, v_\ell)$.

Lemma 5.3. (H, A) *defined above is a* $\log(n)$*-local load-balanced communication network.*

Proof. Both locality and efficiency follow by inspection. Clearly, for every $n = 2^d$, H_n has degree $d = \log n$, and A has runtime $O(\mathsf{polylog} n)$ and outputs a path of length at most $\log n$. The load balancing property follows by observing that A satisfy a "shift-invariant" property that when both inputs s, t are shifted by some w, (the distribution of) A's output is simply shifted by w as well. Formally, for every shift $w \in \{0,1\}^d$, inputs $s, t \in \{0,1\}^d$ and output $p_{s,t} = (v_0, \ldots, v_\ell)$ of $A(n, s, t)$, let $s' = s \oplus w$, $t' = t \oplus w$, $v_i' = v_i \oplus w$ for $i \in [\ell]$ and $p_{s',t'} = (v_0', \ldots, v_\ell')$. It holds that $\Pr[p_{s,t} \leftarrow A(n, s, t)] = \Pr[p_{s',t'} \leftarrow A(n, s', t')]$. Therefore, for every $u, v \in \{0,1\}^d$, let $w = u \oplus v$, the shift-invariant property implies that

$$E_{s,t\leftarrow[n]}[u \in A(n,s,t)] = E_{s,t\leftarrow[n]}[(u \oplus w) \in A(n, s \oplus w, t \oplus w)] = E_{s,t\leftarrow[n]}[v \in A(n,s,t)].$$

□

5.1.2 Proof of Theorem 5.1 with $\mathsf{polylog}(n)$ Input Size

We now prove Theorem 5.1 assuming that per-party input size is $|x| = \mathsf{polylog}(n)$.

Proof (of Theorem 5.1 with $|x| = \mathsf{polylog}(n)$*).* We construct the desired protocol, called $\mathsf{LLB\text{-}MPC\text{-}Dyn}$, by appropriately modifying the $\mathsf{LB\text{-}MPC\text{-}Dyn}$ protocol from Section 4 as follows.

- By inspection, the Committee Setup Phase already achieves $\mathsf{polylog}(n)$ communication locality with the help of a single per-party broadcast, since the per-party communication complexity (excluding broadcast) is $\mathsf{polylog}(n)$. Therefore, no modification is needed for the Committee Setup Phase.
- For the Input Initialization Phase, note that the per-party complexity is $\tilde{O}(|x|)$. Since we assume $|x| = \mathsf{polylog}(n)$, it has $\mathsf{polylog}(n)$ communication locality as well and no modification is needed.
- For the dual ORAM program execution in the Computation Phase, we modify the implementation of the $\mathcal{F}^{\mathcal{E},\mathcal{E}'}_{\mathsf{Command}}$ ideal functionality in the way we discussed above using the $\log(N)$-local load-balanced communication network (H, A) constructed in Section 5.1.1. Namely, the sender committee W_s of $\mathcal{F}^{\mathcal{E},\mathcal{E}'}_{\mathsf{Command}}$ computes a path from W_s to the recipient committee W_t by first sample a random $u \leftarrow [N]$, invoke the routing algorithm A to obtain $p_{s,u} \leftarrow A(N, s, u)$ and $p_{u,t} \leftarrow A(N, u, t)$, and let $p_{s,t} = p_{s,u} \circ p_{u,t}$. Then W_s "sends" $[\mathsf{op}]$ through the path $p_{s,t}$ of length $O(\log N)$ to W_t, who performs the operation and "sends" back a value $[v_1]$ back to W_s through the same path, implemented using $\mathcal{F}^{\mathcal{E},\mathcal{E}'}_{\mathsf{LocalCom}}$ ideal functionality. We note that for load balancing and job-passing scheduling algorithm, we still use the number of invocations to (now virtual) $\mathcal{F}^{\mathcal{E},\mathcal{E}'}_{\mathsf{Command}}$ and $\mathcal{F}^{\mathcal{E}}_{\mathsf{Comp}}$ as our base cost metric for the scheduling algorithm, but ignoring the $\mathcal{F}^{\mathcal{E},\mathcal{E}'}_{\mathsf{LocalCom}}$.
- Finally, for output delivery, instead of relying on the output delivery procedure in Figure 9, we simply let the CPU worker disseminate the output through the hypercube underlying the local load-balanced communication network (H, A). More precisely, the output is delivered by a breadth-first search over the hypercube. Since we route the output through the hypercube, it clearly has $\mathsf{polylog}(n)$ communication locality. Additional, note that there is some fixed

$\mathsf{polylog}(n)$ upper bound on the work of the committee. Thus, we can preserve load balancing property by appropriated padding.

As argued above, by inspection, LLB-MPC-Dyn has $\mathsf{polylog}(n)$ locality (assuming the input has size upper-bounded by $\mathsf{polylog}(n)$). Also note that the total communication and computation complexity of LLB-MPC-Dyn is preserved, since the above implementation only blows up both complexities of $\mathcal{F}^{\mathcal{E},\mathcal{E}'}_{\mathsf{Command}}$ by a $\mathsf{polylog}(n)$ factor. The per-party memory is preserved since each committee only need $\mathsf{polylog}(n)$ extra temporary memory for handling with routing.

It remains to show that load balancing is preserved. Note that we only need to show that the *routing cost* incurred by the the $\mathcal{F}^{\mathcal{E},\mathcal{E}'}_{\mathsf{LocalCom}}$ ideal functionality is load balanced, since both the job-passing scheduling algorithm and the cost of the remaining part of the protocol remains unchanged. Let us focus on the computation complexity, and the same argument applies to the communication complexity. To see that the computation complexity of $\mathcal{F}^{\mathcal{E},\mathcal{E}'}_{\mathsf{LocalCom}}$ is load balanced, we define two non-typical cost metrics CP_s and CP_t that jointly capture the computational complexity exactly. At a high-level, we do not charge the cost of $\mathcal{F}^{\mathcal{E},\mathcal{E}'}_{\mathsf{LocalCom}}$ to the worker committee that performs the computation. Instead, we charge the cost to the jobs of the sender committee W_s and destination committee W_t of $\mathcal{F}^{\mathcal{E},\mathcal{E}'}_{\mathsf{Command}}$ that cause the invocation of $\mathcal{F}^{\mathcal{E},\mathcal{E}'}_{\mathsf{LocalCom}}$, where we split the cost into two halves, where the first half is the cost for routing between s and the middle random point u, which is charged to the job of W_s and captured by CP_s, and the second half is the cost for routing between u and t, which is charged to the job of W_t and captured by CP_t. Specifically, we define $\mathsf{CP}_s = (c^s, sc^s_i, \beta^s_i, \gamma^s_j)$ as follows. We let $c^s(i)$ be the total computation cost of J_i, when J_i is the job of the sender committee W_s, for routing between s and u (note that the cost is in fact independent of the worker W_s by the symmetric of the hypercube routing network). The switch cost is defined in an analogous way that measures the first half routing cost of switching for the sender. The crucial point here is that we define $\gamma^s_j(j')$ to be the random variable of the fraction of computation cost incurred by the worker W_j being the sender to the worker $W_{j'}$, and the load balancing property of (H, A) implies that CP_s is a *fair* cost metric. Note that the above definition indeed charges computation complexity of each worker committee accurately (for the routing cost they participate in the first half of the routing between s and u). It is not hard to check that CP_s is $\mathsf{polylog}(n)$-bounded by the base cost metric CP.

We can define CP_t in an analogous way to capture the second half of the routing cost, and by the same reason, CP_t is a fair cost metric $\mathsf{polylog}(n)$-bounded by CP. Note that now CP_s and CP_t jointly captures exactly the computation complexity incurred by $\mathcal{F}^{\mathcal{E},\mathcal{E}'}_{\mathsf{LocalCom}}$. We can now appeal to the robust load balancing property from Lemma 4.4 and conclude that the computation complexity of $\mathcal{F}^{\mathcal{E},\mathcal{E}'}_{\mathsf{LocalCom}}$ is load balanced (on the committee level). Finally, the same argument shows that the communication complexity of of $\mathcal{F}^{\mathcal{E},\mathcal{E}'}_{\mathsf{LocalCom}}$ is load balanced as well.

□

5.2 Handling Large Inputs

We now discuss how to further modify the Input Initialization Phase of LLB-MPC-Dyn to achieve $\mathsf{polylog}(n)$ communication locality even with large inputs, while preserving the load-balancing property and the protocol complexities. Let us first recall the high-level structure of the Input Initialization Phase.

Recall that in the Input Initialization Phase, each party P_j initially holds his input x_j of length $|x|$, and at the conclusion of the protocol, the inputs are entered into the Input tape ORAM structure, shared among the input tape ORAM memory node committees.

Let $\mathsf{size} \in \mathsf{polylog}(n)$ denote the memory block size of the ORAM. When the input size is large, P_j first splits his input into $|x|/\mathsf{size}$ blocks, and commits in parallel to each block to separate "parallel CPU" committees $C_{j'}$ via $\mathcal{F}_{\mathsf{Enc}}^{j,C_{j'}}$ ideal functionality (see Remark 3.17). Then the $L = n \cdot |x|/\mathsf{size} \in \tilde{O}(n \cdot |x|)$ committees jointly execute the $\mathsf{DistribInit}$ protocol, which in turn emulates the $\mathsf{ParallelInsert}$ procedure that initializes an ORAM structure with L blocks inserted in parallel.

In more detail, the $\mathsf{ParallelInsert}$ procedure starts with L parallel CPUs C_i, each holding one memory block, and an empty ORAM structure. Each C_i first samples a random position $\mathsf{Pos}(i)$ of its block, and then the L CPUs insert the L blocks directly to the $\ell = \log L$ level[21] of the ORAM tree according to $\mathsf{Pos}(i)$ in parallel through an L-to-L oblivious routing network. The routing network consists of $\log L$ rounds with deterministic communication pattern, where in each round, the CPUs group into $L/2$ pairs and perform $L/2$ pairwise communication in each pair to route the blocks. Then the CPUs merge the position map Pos into L/α blocks, and recursively enters the L/α blocks to the ORAM tree in the next recursive level (with L/α CPUs). At the conclusion of $\mathsf{ParallelInsert}$, a single CPU with $\mathsf{polylog}(L)$-size state and an ORAM structure with L inserted blocks are initialized.

Finally, for the purpose of load balancing, we append an Initial Worker Assignment protocol after $\mathsf{DistribInit}$, which assigns each job/role of CPU and ORAM memory node committee to a random worker.

When the input size is large, there are three reasons for poor communication locality:

- Input-committing: Each party P_j needs to commit to $|x|/\mathsf{size}$ input blocks to the same number of committees, which results in $(|x|/\mathsf{size}) \cdot \mathsf{polylog}(n) = \tilde{O}(|x|)$ communication locality.
- $\mathsf{DistribInit}$: Note that the CPUs in the $\mathsf{ParallelInsert}$ procedure have $\mathsf{polylog}(L)$ communication locality (since each performs only $\mathsf{polylog}(L)$ amount of work), which translates to $\mathsf{polylog}(L)$ communication locality in the $\mathsf{DistribInit}$ protocols on the *committee* level. However, note that there are L committees, each of which consists of a random set of $\mathsf{polylog}(n)$ parties. Therefore, each party participates in $\tilde{O}(|x|)$ committees on average, resulting in poor $\tilde{O}(|x|)$ communication locality.
- The final Initial Worker Assignment protocol involves $\tilde{O}(L)$ committees sending out their state to worker committees, which also results in poor communication locality.

We use the same ideas as in the previous case to achieve good communication locality. Namely, we elect a set of Worker committees, each of which plays multiple jobs/roles of parallel CPUs and ORAM memory nodes. Additionally, the pairwise communications between committees are routed through a low-degree network associated with the Worker committees.

As before, we need to be careful to maintain the load-balancing property. Here, the corresponding load-balancing problem is simpler, since we know an a priori $\tilde{O}(|x|)$ bound on the per-party complexity. Thus, it suffices to ensure that no party performs more than some fixed $\tilde{O}(|x|)$ amount of work, and apply appropriate padding to achieve load-balancing.

We rely on the following structure of the Input Initialization Phase summarized above, and a hypercube routing algorithm of Valiant and Brebner [VB81] to ensure load balancing.

- There are $\tilde{O}(L)$ jobs/roles of parallel CPUs and ORAM memory nodes.

[21]For simplicity, we assume L is a power of 2. The assumption can be made without loss of generality by padding dummy input blocks.

- The $\mathsf{DistribInit}$ protocol consists of $\mathsf{polylog}(L)$ parallel rounds with deterministic pairwise communication pattern per round.

Recall that we elect $N = n \cdot \mathsf{polylog}(n)$ worker committees. At a high level, we assign the $\tilde{O}(L)$ jobs to the workers (somewhat) evenly in a deterministic way so that each worker is assigned at most $\tilde{O}(|x|)$ jobs. This trivially implies that in each round of the $\mathsf{DistribInit}$ protocol, each worker participated in at most $\tilde{O}(|x|)$ number of pairwise communication. We then use the hypercube routing algorithm of [VB81] to route the communication, which has the property that no edge need to route more that $\tilde{O}(|x|)$ messages, except with negligible probability. This implies that the work performed by each worker for each round of the $\mathsf{DistribInit}$ protocol can be upper bounded by $\tilde{O}(|x|)$, which in turn implies that the overall complexity of each worker committee (and hence each party) is at most $\tilde{O}(|x|)$ in the $\mathsf{DistribInit}$ protocol. For the final Initial Worker Assignment protocol, we simply let each worker send each of his jobs to a random worker, and route the communication via the algorithm of [VB81], which again yields at most $\tilde{O}(|x|)$ per-party complexity.

Let H_d denote dimension-d Boolean hybercube graph with $D = 2^d$ nodes. The hypercube routing algorithm of [VB81] is a randomized oblivious routing algorithm R that on input $d \in \mathbb{N}$ and $s, t \in [D]$, outputs a path of length $O(d)$ from s to t in H_d with the following property.[22]

Theorem 5.4 ([VB81]). *For every $d \in \mathbb{N}$, and every set of routing request $\{(s_i, t_i)\}_i$ such that each node $v \in H_d$ appears in at most m pairs, it holds that for every edge $e \in H_d$,*

$$\Pr\left[|\{i : e \in R(d, s_i, t_i)\}| \geq m \cdot d^2\right] \leq \mathsf{negl}(D),$$

where the probability is over the randomness of the routing algorithm.

We proceed to prove Theorem 5.1 in its full generality.

Proof. (Proof of Theorem 5.1) We further modify the Input Initialization Phase of $\mathsf{LLB\text{-}MPC\text{-}Dyn}$ as follows.

1. We no longer use the CPU and ORAM memory node committees, but instead assign these jobs/roles to the worker committees (specified below).
2. (Input-committing:) Recall that we elect $N = n \cdot \mathsf{polylog}(n)$ worker committees in the Committee Setup Phase. For simplicity, we assume N is a power of 2 and associate it with a Boolean hybercube H. For each $j \in [n]$, let party P_j commit his whole input x_j to the corresponding Worker committee W_j (the remaining $N - n$ workers do not receive inputs).
3. For $j \in [n]$, each W_j internally splits x_j into $|x|/\mathsf{size}$ blocks, and plays the roles of corresponding parallel CPU in the $\mathsf{DistribInit}$ protocol. As such, each worker plays at most $|x|/\mathsf{size} \in \tilde{O}(|x|)$ parallel CPU roles (the remaining $N - n$ workers play no roles), and there are $L = n \cdot |x|/\mathsf{size}$ roles in total.
4. Note that the Input tape ORAM structure has $M = \tilde{O}(L)$ memory nodes in total (summing over all recursion levels). We assign the memory nodes to the workers evenly as follows. We label each memory node by an index in $[M]$, and assign nodes with index $\alpha N + j$ to worker W_j for every $j \in [N]$. Note that each worker is assigned $\tilde{O}(|x|)$ nodes.

[22][VB81] in fact proved a stronger property in the context of parallel communication schemes with tighter parameters, which implies Theorem 5.4.

5. ($\mathsf{DistribInit}$:) The worker committees jointly execute the $\mathsf{DistribInit}$ protocol with assigned roles. Recall that at each round, the roles group into pairs and perform pairwise communication with deterministic pattern. Thus, a worker W_s playing a role knows which worker W_t playing the corresponding role to talk to. W_s and W_t route the communication through H using the routing algorithm $R(\log N, s, t)$ of [VB81].
6. (Initial Worker Assignment:) Each role is assigned to a new random worker as follows.
   - For each role J_i played by a worker W_j, W_j samples a random new worker $j' \leftarrow [N]$ for J_i.
   - Recall that each role must maintain some link information for the CPU to traverse the ORAM trees. Each W_j sends the link information j' to the worker of the "parent" role of J_i (either the CPU for the root nodes, or the parent for internal ORAM tree nodes). The link information of each role is updated accordingly.
   - Then each worker sends the whole content of each role J_i (including the state and the link information) to its new worker $W_{j'}$.
   - Communication for both of the above two steps are again routed by the routing algorithm R of [VB81].
   - We initialize the Work tape ORAM structure in the same way as before. That is, the CPU worker samples random indices in $[N]$ for each root node of the ORAM Work tape, and appends the samples to its link vector link.

It is not hard to see by inspection that the modified protocol achieves $\mathsf{polylog}(n)$ communication locality and implements the same functionality. That is, at the conclusion of the protocol, a dual ORAM structure with all parties' input inserted is initialized, and each job/role of the ORAM structure is assigned to a random worker. The security of the modified protocol follows in an identical fashion. We proceed to show that the per-party computation complexity is upper bounded by a fixed $\tilde{O}(|x|)$ in the modified Input Initialization Phase protocol above, which implies load balancing with desired complexity by appropriately padding along the protocol.

As argued before, since there are $N = n \cdot \mathsf{polylog}(n)$ worker committees, each party participates in roughly the same number of $\mathsf{polylog}(n)$ committees (up to a $(1 \pm \delta)$ factor) except with negligible probability. Thus, it suffices to upper bound the complexity by $\tilde{O}(|x|)$ on the committee level. We analyze each step of the protocol as follows.

- Both Step 1. and 4. are conceptual steps with no cost.
- In Step 2., for $j \in [n]$, the input committing costs each party P_j and worker W_j complexity $\tilde{O}(|x|)$.
- In Step 3., for $j \in [n]$, each worker W_j performs $\tilde{O}(|x|)$ amount of work to split the input x_j.
- In Step 5., recall that in the original protocol, each pairwise communication via execution of the ideal functionality $\mathcal{F}^{\mathcal{E},\mathcal{E}'}_{\mathsf{Command}}$ has $\mathsf{polylog}(n)$ complexity. Thus, the messages that must be routed in the modified protocol have size at most $\mathsf{polylog}(n)$. At each round of $\mathsf{DistribInit}$, since each worker W_j only plays $\tilde{O}(|x|)$ roles, W_j participates in at most $\tilde{O}(|x|)$ pairwise communications. By Theorem 5.4, each edge of the hypercube only needs to route at most $\tilde{O}(|x|)$ messages, except with negligible probability. Together with the fact that the hypercube has $\log N$ degree, and that the $\mathsf{DistribInit}$ protocol has only $\mathsf{polylog}(n)$ parallel rounds, the total complexity of each worker in the $\mathsf{DistribInit}$ protocol is upper bounded by $\tilde{O}(|x|)$, except with negligible probability.

- In Step 6., recall that we randomly assign $\tilde{O}(L)$ jobs/roles to new workers. By a standard Chernoff bound, each worker is assigned $\tilde{O}(|x|)$ jobs except with negligible probability. Note that the protocol consists of only $\tilde{O}(1)$ parallel rounds, where in each round, all pairwise communication exchange $\mathsf{polylog}(n)$ size messages. By the same argument as above, the complexity of each worker in this step is upper bounded by $\tilde{O}(|x|)$ except with negligible probability.

□

References

[AJLA$^+$12] Gilad Asharov, Abhishek Jain, Adriana López-Alt, Eran Tromer, Vinod Vaikuntanathan, and Daniel Wichs. Multiparty computation with low communication, computation and interaction via threshold fhe. In *EUROCRYPT*, pages 483–501, 2012.

[BGJK12] Elette Boyle, Shafi Goldwasser, Abhishek Jain, and Yael Tauman Kalai. Multiparty computation secure against continual memory leakage. In *STOC*, pages 1235–1254, 2012.

[BGT13] Elette Boyle, Shafi Goldwasser, and Stefano Tessaro. Communication locality in secure multi-party computation - how to run sublinear algorithms in a distributed setting. In *TCC*, pages 356–376, 2013.

[BGW88] Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation (extended abstract). In *STOC*, pages 1–10, 1988.

[BR94] Mihir Bellare and Phillip Rogaway. Optimal asymmetric encryption. In *EUROCRYPT*, pages 92–111, 1994.

[BSGH13] Nicolas Braud-Santoni, Rachid Guerraoui, and Florian Huc. Fast byzantine agreement. In *PODC*, pages 57–64, 2013.

[CP13] Kai-Min Chung and Rafael Pass. A simple oram. Cryptology ePrint Archive, Report 2013/243, 2013.

[DI06] Ivan Damgård and Yuval Ishai. Scalable secure multiparty computation. In *CRYPTO*, pages 501–520, 2006.

[DIK$^+$08] Ivan Damgård, Yuval Ishai, Mikkel Krøigaard, Jesper Buus Nielsen, and Adam Smith. Scalable multiparty computation with nearly optimal work and resilience. In *CRYPTO*, pages 241–261, 2008.

[DIK10] Ivan Damgård, Yuval Ishai, and Mikkel Krøigaard. Perfectly secure multiparty computation and the computational overhead of cryptography. In *EUROCRYPT*, pages 445–465, 2010.

[DKMS12] Varsha Dani, Valerie King, Mahnush Movahedi, and Jared Saia. Breaking the o(nm) bit barrier: Secure multiparty computation with a static adversary. *CoRR*, abs/1203.0289, 2012.

[DMN11] Ivan Damgård, Sigurd Meldgaard, and Jesper Buus Nielsen. Perfectly secure oblivious ram without random oracles. In *TCC*, pages 144–163, 2011.

[DN07] Ivan Damgård and Jesper Buus Nielsen. Scalable and unconditionally secure multiparty computation. In *CRYPTO*, pages 572–590, 2007.

[Fei99] Uriel Feige. Noncryptographic selection protocols. In *Proceedings of the 40th Annual Symposium on Foundations of Computer Science*, 1999.

[Gen09] Craig Gentry. Fully homomorphic encryption using ideal lattices. In *STOC*, pages 169–178, 2009.

[GGH$^+$13] Craig Gentry, Kenny A. Goldman, Shai Halevi, Charanjit S. Jutla, Mariana Raykova, and Daniel Wichs. Optimizing oram and using it efficiently for secure computation. In *Privacy Enhancing Technologies*, pages 1–18, 2013.

[GKK$^+$11] S. Dov Gordon, Jonathan Katz, Vladimir Kolesnikov, Tal Malkin, Mariana Raykova, and Yevgeniy Vahlis. Secure computation with sublinear amortized work. *IACR Cryptology ePrint Archive*, 2011:482, 2011.

[GMW87] Oded Goldreich, Silvio Micali, and Avi Wigderson. How to play any mental game or a completeness theorem for protocols with honest majority. In *STOC*, pages 218–229, 1987.

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

[KLST11] Valerie King, Steven Lonargan, Jared Saia, and Amitabh Trehan. Load balanced scalable byzantine agreement through quorum building, with full information. In *ICDCN*, pages 203–214, 2011.

[KSSV06] Valerie King, Jared Saia, Vishal Sanwalani, and Erik Vee. Scalable leader election. In *SODA*, pages 990–999, 2006.

[LO13] Steve Lu and Rafail Ostrovsky. Distributed oblivious ram for secure two-party computation. In *TCC*, pages 377–396, 2013.

[MSS13] Steven Myers, Mona Sergi, and Abhi Shelat. Black-box proof of knowledge of plaintext and multiparty computation with low communication overhead. In *TCC*, pages 397–417, 2013.

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

[SCSL11] Elaine Shi, T.-H. Hubert Chan, Emil Stefanov, and Mingfei Li. Oblivious ram with o((logn)3) worst-case cost. In *ASIACRYPT*, pages 197–214, 2011.

[SvDS$^+$13] Emil Stefanov, Marten van Dijk, Elaine Shi, Christopher W. Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. Path oram: an extremely simple oblivious ram protocol. In *ACM Conference on Computer and Communications Security*, pages 299–310, 2013.

[VB81] Leslie G. Valiant and Gordon J. Brebner. Universal schemes for parallel communication. In *STOC*, pages 263–277, 1981.

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

[ZMS14] Mahdi Zamani, Mahnush Movahedi, and Jared Saia. Millions of millionaires: Multiparty computation in large networks. Cryptology ePrint Archive, Report 2014/149, 2014.

A Proof of Theorem 2.7

This section describes the underlying committee election and coin tossing protocol that is used as a tool within the setup phase of our MPC protocol, culminating in a proof of Theorem 2.7.

A.1 Memory-Efficient Committee Election

First, consider the task of reaching full agreement on a single "good" $\mathsf{polylog}(n)$-size committee, while maintaining only $\mathsf{polylog}(n)$ memory per party. We build upon the *almost-everywhere* committee election protocol of King *et al.* [KSSV06]. The main theorem of [KSSV06] focuses on electing a single party as leader, and guarantees almost-everywhere agreement on an honest leader with constant probability of success. We present a slightly modified variant of this theorem statement, in which their protocol is truncated to elect a committee of size $\mathsf{polylog}(n)$, and guarantees that this committee is 2/3 honest with overwhelming probability (assuming an original $(2/3 + \epsilon)$ majority of honest parties).

Theorem A.1 ([KSSV06] Almost-Everywhere Committee Election Protocol)**.** *Suppose there are n processors, at least $(2/3 + \epsilon)$ are honest for constant $\epsilon > 0$. Then for $K \in \mathsf{polylog}(n) \cap \omega(\log n)$, there exists an algorithm that elects, with overwhelming probability, a committee $C \subset [n]$ of size $|C| \in \Theta(K)$ satisfying the following properties:*

- *$1 - o(1)$ fraction of honest processors agree on C.*
- *$|C \cap H|/|C| \geq (2/3 + \epsilon/2)$, where $H \subset [n]$ denotes honest processors.*
- *Every honest processor sends and processes only $\mathsf{polylog}(n)$ bits.*
- *The number of rounds of communication is $\mathsf{polylog}(n)$.*

From almost-everywhere to everywhere. Now, given any protocol achieving almost-everywhere agreement on a good committee C, we may append a simple polling protocol to achieve *full* agreement on C, with the following per-party complexities, where m denotes the input message size (in our case, the description size of C, which is $\mathsf{polylog}(n)$), $\mathsf{BC}(m)$ denotes an execution of a broadcast channel with message m, and asymptotic notation is with respect to the number of parties n:

	Per-Party Memory	Rounds	Per-Party CC
$\mathsf{Poll}(m)$:	$\tilde{O}(\lvert m\rvert)$	$O(1)$	$\mathsf{BC}(m)$

See Figure 12 for a complete description of the protocol Poll.

Indeed, with overwhelming probability in n, every honest party P_i will select a subset $Q_i \subset [n]$ such that at least 2/3 of the parties in Q_i are honest parties that begin with the same (correct) value m (this holds by a straightforward Chernoff bound). Thus, each honest party P_i will receive a majority of correct responses, and will output in agreement. The memory requirement of each honest party consists of exactly $|Q_i| \cdot |m| \in \tilde{O}(|m|)$.

Combining Poll atop the protocol of [KSSV06], we thus have the following corollary. Note that a description of a $\mathsf{polylog}(n)$-size committee C requires $\mathsf{polylog}(n)$ bits; i.e., we may plug in $|m| = \mathsf{polylog}(n)$.

Corollary A.2. *Suppose there are n processors, at least $(2/3 + \epsilon)$ fraction are honest for constant $\epsilon > 0$. Then for $K \in \mathsf{polylog}(n) \cap \omega(\log n)$, there exists an protocol Elect such that with overwhelming*

Polling protocol Poll: Going from almost-everywhere to full agreement
Input: Each party P_i: m_i. There exists m s.t. $m_i = m$ for $(1 - o(1))n$ honest parties P_i.
Output: Each party P_i: m'_i, such that $m'_i = m$ for *all* honest P_i.

Each party P_i performs the following steps:

1. Sample a random query set $Q_i \subset [n]$ of size $|Q_i| = \log^2 n$.
2. Broadcast message m_i.
3. Ignore all incoming messages from parties $P_\ell \notin Q_i$. Output m'_i defined to be the most frequently occurring response m_j from $P_j \in Q_i$.

Figure 12: Protocol Poll to achieve full agreement on message m when starting with *almost-everywhere* agreement.

probability all honest parties *agree on a committee $C \subset [n]$ of size $|C| \in \Theta(K)$, for which $|C \cap H|/|C| \geq (2/3 + \epsilon/2)$ (for $H \subset [n]$ honest parties), with the following per-party complexities (where* BC *is a broadcast of* polylog(n) *bits):*

	Per-Party Memory	*Rounds*	*Per-Party CC*
Elect*:*	$\tilde{O}(1)$	$\tilde{O}(1)$	$\mathsf{BC} + \tilde{O}(1)$

A.2 Memory-Efficient Committee Election + Coin Tossing

Our MPC protocol will ultimately require electing a large number of good committees. As we show in Section 3.1, to generate such committees it suffices to sample a random polylog(n)-size seed s to a polylog(n)-wise independent function family $\{F_s\}$, taking the ith committee to be $C_i := F_s(i)$. One approach for generating and communicating such a seed is to start by electing a single good committee via Corollary A.2 as described above (reaching full agreement), and then have this committee collectively sample the string s and broadcast it to all parties. Indeed, this procedure follows in a black-box way from the almost-everywhere committee election protocol of [KSSV06] (Theorem A.1). However, this approach unfortunately requires *two* sequential implementations of broadcast: one per party to achieve full agreement on the identity of the first committee C (as in the Corollary A.2 protocol), and then a second broadcast per party in C to communicate the sampled seed s to all parties. Indeed, it is crucial for the remainder of the MPC protocol that *all* parties agree both on C and on the seed s.

Alternatively, we can combine these two sequential broadcasts into one, by "opening up the box" of the underlying almost-everywhere committee election protocol, and taking advantage of its specific structure. Loosely speaking, the [KSSV06] protocol achieves almost-everywhere agreement on a good committee C by constructing a tournament-style communication tree, where each node of the tree is populated dynamically during the course of the protocol. The final elected committee C corresponds to the parties elected to the root node of the tree. In their protocol, it holds that all honest parties in C are among the $1 - o(1)$ fraction of honest parties who agree on the correct committee composition, thus allowing them to run a protocol together amongst themselves (e.g., a small-scale standard MPC or coin tossing protocol). And most importantly, the communication tree structure allows the root node committee to "pass messages down the tree" in such a way that *almost everyone* will receive the message correctly. This structure was formalized and used in [BGT13].

Namely, in the notation of [BGT13]: (1) Protocol $\mathsf{BuildTree}$ corresponds to the [KSSV06] protocol, corresponding to Theorem A.1 in the present work, which yields the described communication tree structure; (2) Protocol $\mathsf{SendDownTree}$ (constructed in [BGT13]) provides a means for passing a message from the root node committee to (almost all) other parties (while incurring $\mathsf{polylog}(n)$ communication per party and $\mathsf{polylog}(n)$ sequential communication rounds).

Lemma A.3 ([BGT13])**.** *With overwhelming probability in n over the execution of* $\mathsf{BuildTree}$*, the following properties hold, in addition to the properties listed in Theorem A.1.*

- *All parties assigned to the elected committee C agree on the correct composition of C.*
- *If all honest parties assigned to the elected committee C initiate protocol* $\mathsf{SendDownTree}$ *with the same input s, then at its conclusion, $1 - o(1)$ fraction of all n honest parties agree on s.*

Proof. Follows from Theorem 3.2 and Lemma 3.3 of [BGT13] (corresponding to the cited theorem statement of the [KSSV06] result, and the $\mathsf{SendDownTree}$ protocol lemma). $\square$

In particular, this can be directly utilized to achieve a stronger notion of *almost-everywhere coin tossing*, where at the conclusion of the protocol $1 - o(1)$ fraction of honest parties agree on a (statistically close to) uniform string s.[23] Namely, the parties assigned to the elected committee C can communicate amongst themselves (since they all agree on the correct committee assignment) to perform a standard coin-tossing (or generic MPC, e.g. [BGW88]) protocol to collectively generate a uniform random string s, and then all honest parties in C will initiate the protocol $\mathsf{SendDownTree}$ to communicate s to almost all of the n total parties.

Note that we must be careful to ensure that the lack of full agreement on C does not unwittingly open the protocol to large unnecessary computation when generating the seed s. Indeed, in principle, parties in the $o(1)$ fraction of the misinformed may mistakenly believe they are part of the elected committee C. Such a mistake would incur additional wasted computation and memory on their behalf, while attempting to execute the coin tossing protocol together with a collection of other parties he believes to be elected to C. However, even in the worst case (e.g., if the party is tricked into believing that C consists of himself and a collection of other corrupted parties), the total of any such wasted computation and memory will be bounded by $\mathsf{polylog}(n)$ bits per party. This is because any party will only attempt to listen or speak to those parties he believes to be elected to C, which will necessarily be bounded in size by the parameter $K \in \mathsf{polylog}(n)$ (otherwise the party can be sure that he does not hold the correct value of C, and so must not be in the correct elected committee, in which case he can refrain from participating in this step).

We now put these pieces together.

Lemma A.4 (Almost-Everywhere Committee Election + Coin Tossing)**.** *Suppose there are n processors, at least $(2/3 + \epsilon)$ are honest for constant $\epsilon > 0$. Then for any $k \in \mathsf{polylog}(n)$, and $K \in \mathsf{polylog}(n) \cap \omega(\log n)$, there exists an algorithm that outputs, with overwhelming probability (in n), a string $s \in \{0,1\}^k$ and a committee $C \subset [n]$ of size $|C| \in \Theta(K)$, satisfying the following properties:*

- $1 - o(1)$ *fraction of honest processors agree on s and C.*
- *The distribution of s over the random coins of the honest parties is statistically close to uniform over $\{0,1\}^k$ (i.e., statistical distance negligible in n).*

[23] Note that a.e. agreement on a random string suffices to achieve a.e. agreement on a good committee.

- $|C \cap H|/|C| \geq (2/3 + \epsilon/2)$, *where* $H \subset [n]$ *denotes honest processors.*
- *Every honest processor sends and processes only* $\mathsf{polylog}(n)$ *bits.*
- *The number of rounds of communication is* $\mathsf{polylog}(n)$*.*

Proof. Follows from Theorem A.1 and Lemma A.3. □

Now, given this almost-everywhere committee election + coin tossing protocol, we can combine two earlier broadcasts from our MPC protocol into one: Namely, the parties will first reach almost-everywhere agreement on *both* the committee C and on a random string s; Then, as before, each party will make a *single* broadcast as described in the Poll protocol (Figure 12) to communicate their vote on the pair of values together, enabling full agreement to be reached on both.

Corollary A.5. *Suppose there are n processors, at least $(2/3 + \epsilon)$ are honest, for constant $\epsilon > 0$. Then, with the addition of a single broadcast of* $\mathsf{polylog}(n)$ *bits per party, there exists a protocol achieving the properties listed in Lemma A.4, for which, with overwhelming probability in n,* all *parties agree on s and C.*

This is precisely the desired guarantee, proving Theorem 2.7.