Large-Scale Secure Computation: Multi-party Computation for (Parallel) RAM Programs

Elette Boyle*
Technion Israel
eboyle@alum.mit.edu, Kai-Min Chung
Academica Sinica
kmchung@iis.sinica.edu.tw, and Rafael Pass**
Cornell University
rafael@cs.cornell.edu

No Institute Given

Abstract. We present the first efficient (i.e., polylogarithmic overhead) method for securely and privately processing large data sets over multiple parties with *parallel, distributed algorithms*. More specifically, we demonstrate load-balanced, statistically secure computation protocols for computing Parallel RAM (PRAM) programs, handling $(1/3-\epsilon)$ fraction malicious players, while preserving up to polylogarithmic factors the computation and memory complexities of the PRAM program, aside from a one-time execution of a broadcast protocol per party. Additionally, our protocol has $\mathsf{polylog}$ communication locality—that is, each of the n parties speaks only with $\mathsf{polylog}(n)$ other parties.

1 Introduction

Large data sets, such as medical data, genetic data, transaction data, the web and web access logs, and network traffic data, are now in abundance. Much of the data is stored or made accessible in a distributed fashion, having necessitated the development of efficient distributed protocols that compute over such data. In particular, novel programming models for processing large data sets with *parallel, distributed algorithms*, such

* 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.

as MapReduce (and its implementation Hadoop) are emerging as crucial tools for leveraging this data in important ways.

But these methods require that the data itself is revealed to the participating servers performing the computation—and thus blatantly violate the privacy of potentially sensitive data. As a consequence, such methods cannot be used in many critical applications (e.g., discovery of causes or treatments of diseases using genetic or medical data).

In contrast, methods such as secure multi-party computation (MPC), introduced in the seminal works of Yao [Yao86] and Goldreich, Micali and Wigderson [GMW87], enable securely and privately performing any computation on individuals private inputs (assuming some fraction of the parties are honest). However, despite great progress in developing these techniques, there are no MPC protocols whose efficiency and communication requirements scale to the modern regime of large-scale distributed, parallel data processing.

We are concerned with merging these two approaches. In particular,

We seek MPC protocols that efficiently *(technically, with polylogarithmic overhead) enable secure and private processing of large data sets with parallel, distributed algorithms.*

Explicitly, in this large-scale regime, the following properties are paramount:

1. *Exploiting Random Access.* Computations on large data sets are frequently "lightweight": accessing a small number of dynamically chosen data items, relying on conditional branching, and/or maintaining small memory. This means that converting a program first into a circuit to enable its secure computation, which immediately obliterates these gains, will not be a feasible option.
2. *Exploiting Parallelism.* In fact, as mentioned, to effectively solve large-scale problems, modern programming models heavily leverage parallelism. The notion of a Parallel RAM (PRAM) better captures such computing models. In the PRAM model of computation, several (polynomially many) CPUs run simultaneously, potentially communicating with one another, while accessing the same shared external memory. We consider a PRAM model with a variable number of CPUs but with a fixed activation structure (i.e., what processors are activated at which time steps is fixed). Note that such a model simultanously captures RAMs (a single CPU) and circuits (the circuit topology dictates the CPU activation structure).

Additionally, the following desiderata are often of importance:

3. *Load balancing.* When the data set contains tens or hundreds of thousands of users' data, it is often unreasonable to assume that any single user can provide memory, computation, or communication resources on the order of the data of *all users*. Rather, we would like to *balance* the load across nodes.
4. *Communication Locality.* In many cases, establishing a secure communication channel with a large number of distinct parties may be costly, and thus we would like to minimize the *locality of communication* [BGT13]: that is, the number of total parties that each party must send and receive message to during the course of the protocol.

To date, no existing work addresses secure computation of Parallel RAM programs. Indeed, nearly all results in MPC require a *circuit* model for the function being evaluated (including the line of work on scalable MPC [DI06,DIK+08,DKMS12,ZMS14]), and thus inherit resource requirements that are linear in the circuit size. Even for (sequential) RAM, the only known protocols either only handle two parties [OS97,GKK+11,LO13,GGH+13], or in the context of multi-party computation require all parties to store *all inputs* [DMN11], rendering the protocol useless in a large-scale setting (even forgetting about computation load balancing and locality).

1.1 Our Results

We present a statistically secure MPC for (any sequence of) PRAMs handling $(1/3-\epsilon)$ fraction static corruptions in a synchronous communication network, with secure point-to-point channels. In addition, our protocol is strongly *load balanced* and *communication local* (i.e., $\mathsf{polylog}(n)$ locality). We state our theorem assuming each party itself is a k-processor PRAM, for parameter k.

Theorem 1 (Informal – Main Theorem). *For any constant $\epsilon > 0$ and polynomial parallelism parameter $k = k(n)$, there exists an n-party statistically secure (with error negligible in n) protocol for computing any adaptively chosen sequence of PRAM programs Π_j with fixed CPU activation structures (and that may have bounded shared state), handling $(1/3 - \epsilon)$ fraction static corruptions with the following complexities, where each party is a k-processor PRAM (and where $|x|, |y|$ denote per-party input and output size,*[1] $\mathsf{space}(\Pi)$, $\mathsf{comp}(\Pi)$, *and* $\mathsf{time}(\Pi)$ *denote the worst-case space, computation, and (parallel) runtime of Π, and $CPUs(\Pi)$ denotes the number of CPUs of Π):*

- *Computation per party, per Π_j:* $\tilde{O}\big(\mathsf{comp}(\Pi_j)/n + |y|\big)$.
- *Time steps, per Π_j:* $\tilde{O}\left(\mathsf{time}(\Pi_j) \cdot \max\left\{1, \frac{CPUs(\Pi)}{nk}\right\}\right)$.
- *Memory per party:* $\tilde{O}\left(|x| + |y| + \max_{j=1}^{N} \mathsf{space}(\Pi_j)/n\right)$.
- *Communication Locality:* $\tilde{O}(1)$.

given a one-time preprocessing phase with complexity:

- *Computation per party:* $\tilde{O}(|x|)$, *plus single broadcast of* $\tilde{O}(1)$ *bits.*
- *Time steps:* $\tilde{O}\left(\max\left\{1, \frac{|x|}{k}\right\}\right)$.

Additionally, our protocol achieves a strong "online" load-balancing guarantee: at all times *during the protocol, all parties' communication and computation loads vary by at most a constant multiplicative factor (up to a $\mathsf{polylog}(n)$ additive term).*

Remark 1 (Round complexity). As is the case with all general MPC protocols in the information-theoretic setting to date, the round complexity of our protocol corresponds directly with the time complexity (as when restricted to circuits, parallel complexity corresponds to circuit depth). That is, for each evaluated PRAM program Π_j, the protocol runs in $\tilde{O}(\mathsf{time}(\Pi_j))$ sequential communication rounds to securely evaluate Π_j.

[1] For simplicity of exposition, we assume all parties have the same input size and receive the same output.

Remark 2 (On the achieved parameters). Note that in terms of memory, each party only stores her input, output, and her "fair" share of the required space complexity, up to polylogarithmic factors. In terms of computation (up to polylogarithmic factors), each party does her "fair" share of the computation, receives her outputs, and in addition is required to read her entire input at an initial preprocessing stage (even though the computations may only involve a subset of the input bits; this additional overhead of "touching" the whole input once is necessary to achieve security).[2] Finally, the time complexity corresponds to the parallel complexity of the PRAM being computed, as long as the combined number of available processors nk from all parties matches or exceeds the number of required parallel processes of the program (and degrades with the corresponding deficit).

Remark 3 (Instantiating the single-use broadcast). The broadcast channel can be instantiated either by the $O(\sqrt{n})$-locality broadcast protocol of King *et al.* [KSSV06], or the $\mathsf{polylog}(n)$-*average* locality protocol of [BSGH13] at the expense of a cost of a one-time per-party computational cost of $O(\sqrt{n})$, or average cost of $\mathsf{polylog}(n)$, 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.[3]

1.2 Construction Overview

Our starting point is an *Oblivious PRAM (OPRAM)* compiler [BCP14,GO96], a tool that compiles any PRAM program into one whose memory access patterns are independent of the data (i.e., "oblivious"). Such a compiler (with polylogarithmic overhead) was recently attained by [BCP14].
Indeed, it is no surprise that such a tool will be useful toward our goal. It has been demonstrated in the sequential setting that Oblivious (sequential) RAM (ORAM) compilers can be used to builds secure 2-party protocols for RAM programs [OS97,GKK$^+$11,LO13,GGH$^+$13]. Taking a similar approach, building upon the OPRAM compiler of [BCP14] directly yields 2-party protocols for PRAMs.
However, OPRAM on its own does not directly provide a solution for *multi*-party computation (when there are many parties). While this approach gives protocols whose complexities scale well with the RAM (or

[2] For general secure computation, and even if we restrict to functionalities that only access a few parties' inputs, and only a few bits of their data, essentially all parties must perform computation at least $\Omega(|x|)$. To see this, consider secure computation of a "multi-party Private Information Retrieval (PIR)" functionality: each party $i > 1$ has as input some "big data" x_i, and party 1 has as input a party index i and an index j into their data x_i. The functionality returns $x_i[j]$ (i.e., the j'th bit of party i's data) to party 1 and nothing to everyone else. We claim that each party $i > 1$ 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.

[3] For instance, it remains open to achieve statistically secure broadcast with worst-case $\mathsf{polylog}(n)$ locality.

PRAM) complexity of the programs, the complexities grow poorly with the number of parties. Indeed, the only current technique for securely evaluating a RAM program on *multiple* parties' inputs [DMN11] is for all parties to hold secret shares of all parties' inputs, and then jointly execute (using standard MPC for circuits) the trusted CPU instructions of the ORAM-compiled version of the program. This means 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.

One may attempt to improve the situation by first electing a small $\mathsf{polylog}(n)$-size representative committee of parties, and then only performing the above steps within this committee. This approach drops the total communication and computation of the protocol to reasonable levels. However, this approach does not save the subset of elected parties from carrying the burden of the entire computation. In particular, each elected party must memory storage equal to the size of *all parties' inputs combined*, making the protocol unusable for "large-scale" computation.

In this paper, we provide a new approach for dealing with this issue. We show how to use an OPRAM in a way that achieves balancing of memory, computation, and communication across all parties.

Our MPC construction proceeds in the following steps:

1. **From OPRAM to MPC.** Given an OPRAM, we begin by considering MPC in a "benign" adversarial setting, which we refer to as *oblivious multi-party computation*, where all parties are assumed to be honest, and we only require that an external attacker that views communication and activation (including memory and computation usages) patterns does not learn anything about the inputs. We show:
   (a) OPRAM yields efficient *memory-balanced* oblivious MPC for PRAM.
   (b) Using committee election techniques (à la [KLST11,DKMS12,BGT13]), any oblivious multi-party computation can be compiled into a standard secure MPC with only $\mathsf{polylog}$ overhead (and a one-time use of a broadcast channel per party).
2. **Load Balancing & Communication Locality.** We next show semi-generic compilers for "nice" (formally defined) *oblivious multi-party* protocols, each introducing only $\mathsf{polylog}(n)$ overhead:
   (a) From any "nice" protocol to one whose computation and communication are *load-balanced*.
   (b) From any "nice" protocol to one that is both *load-balanced* and *communication local* (i.e., $\mathsf{polylog}(n)$ locality).

Our final result is obtained by combining the above steps and observing that Step 1(b) preserves load-balancing and communication locality (and thus can be applied *after* Step 2). Let us mention that *just* Step 1 (together with existing construction of ORAMs) already yields the first MPC protocol for (sequential) RAM programs in which no party must store all parties' inputs. Additionally, *just* Step 1 (together with the OPRAM construction of [BCP14]) yields the first MPC for PRAMs.

We now expand upon each of these steps.

MPC from OPRAM Recall that our construction proceeds via an intermediate notion of *oblivious* security, in which we do not require security against corrupted parties, but rather against an external adversary who sees the activation patterns (i.e., accessed memory addresses and computation times) and communication patterns (i.e., sender/receiver ids and message lengths) of parties throughout the protocol.

Oblivious MPC from OPRAM. At a high level, our protocol will emulate a *distributed* OPRAM[4] structure, where the CPUs and memory cells in the OPRAM are *each associated with parties*. (Recall that we need only achieve "oblivious" security, and thus can trust individual parties with these tasks). The "CPU" parties will control the evaluation flow of the (OPRAM-compiled) program, communicating with the parties emulating the role of the appropriate memory cells for each address to be accessed in the (OPRAM-compiled) database.
The distributed OPRAM 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 and input committees to direct program execution and distributedly store parties' inputs. The distributed OPRAM 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.

Compiling from "Oblivious" Security to Malicious Security. We next present a general compiler taking an oblivious protocol to one that is secure against $(1/3 - \epsilon)n$ statically corrupted malicious parties. (This step can be viewed as a refinement and formalization of ideas from [KLST11,DKMS12,BGT13].) We ensure the compiler tightly preserves the computation, memory, load-balancing, and communication locality of the original protocol, up to $\mathsf{polylog}(n)$ factors (modulo a one-time broadcast per party). This enables us to apply the transformation to any of the oblivious protocols resulting from the intermediate steps in our progression.
At a high level, the compiler takes the following form: (1) First, the parties collectively elect a large number of "good" committees, each of size $\mathsf{polylog}(n)$, where "good" means each committee is composed of at least 2/3 honest parties, and that parties are spread roughly evenly across committees. (2) Each party will verifiably secret share his input among the corresponding committee C_i. (3) From this point on, the role of each party P_i in the original protocol will be emulated by the corresponding

[4] We remark that the term "distributed ORAM" was used with a different meaning in [LO13], in regard to an ORAM that was split across two users.

committee C_i. That is, each local P_i computation will be executed via a small-scale MPC among C_i, and each communication from P_i to P_j will be performed via an MPC among committees C_i and C_j.
The primary challenge in this step is how to elect such committees while incurring only $\mathsf{polylog}(n)$ locality and computation per party. To do so, we build atop the "almost-everywhere" scalable committee election protocol of King *et al.* [KSSV06] to elect a single good committee, and then show that one may use a $\mathsf{polylog}(n)$-wise independent function family $\{F_s\}_{s\in S}$ to elect the remaining committees with small description size (in the fashion of [KLST11,BGT13], for the case of combinatorial samplers and computational pseudorandom functions), with committee i defined as $C_i := F_s(i)$ for fixed random seed s.
We remark that, aside from the one-time broadcast, this compiler preserves load balancing and $\mathsf{polylog}(n)$ locality. Indeed, load balancing is maintained since the committee setup procedure is computationally inexpensive, and each party appears in roughly the same number of "worker" committees. The locality of the resulting protocol increases by an additive $\mathsf{polylog}(n)$ for the committee setup, and a multiplicative $\mathsf{polylog}(n)$ term since all communications are now performed among $\mathsf{polylog}(n)$-size *committees* instead of individual parties.

Load Balancing Distributed Protocols

Load-balancing (Without Locality). We now show how to modify our protocol such that the total computational complexity and memory balancing are preserved, while additionally achieving a strong *computation* 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 computation and communication.[5]
We present and analyze our load-balancing solution in the intermediate *oblivious* MPC security setting (recall that one can then apply the compiler from Step 2(b) above to obtain malicious MPC with analogous load-balancing). Let us mention that there is a huge literature on "load-balanced distributed computation" (e.g., [ACMR95,MPS02,MR98,AAK08]): As far as we can tell, our setting differs from the typical studied scenarios in that we must load balance an underlying distributed protocol, as opposed to a collection of independent "non-communicating jobs". Indeed, the main challenge in our setting is to deal with the fact that "jobs" talk to one another, and this communication must remain efficient also be made load balanced. Furthermore, we seek a load-balanced solution with communication locality.
We consider a large class of arbitrary (potentially load-unbalanced and large-locality) distributed protocols Π, where we view each party in this underlying protocol as a "job". Our goal is to load-balance Π by passing

[5] Note that while our current protocol is memory balanced, it is currently rather imbalanced in computation: e.g., the parties emulating OPRAM CPUs are required to perform computation that is proportional to the whole PRAM computation.

"jobs" between "workers" (which will be the actual parties in the new protocols). More precisely, we start off with any protocol Π that satisfies the following (natural) "nice" properties:

- Each "job" has $\mathsf{polylog}(n)$ size state;
- In each round, each "job" performs at most $\mathsf{polylog}(n)$ computation and communication;
- In each round, each "job" communicates (either sending or receiving a message) to at most one other "job".

It can be verified that these properties hold for our oblivious MPC for PRAM protocol.

Our load-balanced version of such a protocol first randomly[6] efficiently assigns "workers" (i.e., parties) to "jobs". Next, whenever a worker W has performed "enough" work for a particular job J, it randomly selects a replacement worker W' and passes the job over to it (that is, it passes over the state of the job J—which is "small" by assumption). The key obstacle in our setting is that the job J may later communicate with many other jobs, and all the workers responsible for those jobs need to be informed of the switch (and in particular, who the new worker responsible for the job J is). Since the number of jobs is $\Omega(n)$, workers cannot afford to store a complete directory of which worker is currently responsible for each job.

We overcome this obstacle by first modifying Π to ensure that it has small locality—this enables each job to only maintain a short list of the workers currently responsible for the *"neighboring"* jobs. We achieve this locality by requiring that parties (i.e., jobs) in the original protocol Π route their messages along the hypercube. Now, whenever a worker W for a job J is being replaced by some worker W', W informs all J's neighboring jobs (i.e., the workers responsible for them) of this change. We use the Valiant-Brebner [VB81] routing procedure to implement the hypercube routing because it ensures a desirable "low-congestion property," which in our setting translates to ensuring that the overhead of routing is not too high for any individual worker.

The above description has not yet mentioned what it means for a worker to have done "enough" work for a job J. Each round a job is active (i.e., performing some computation), its "cost" increases by 1—we refer to this as an *emulation cost*. Additionally, each time a worker W is switched out from a job J, then J's and each of J's neighboring jobs' costs are increased by 1—we refer to this as a *switch cost*. Finally, once a job's (total) cost has reached a particular threshold τ, its cost is reset to 1 and the worker responsible for the job is switched out. The threshold τ is set to $2 \log M + 1$ where M is the number of jobs.

We show: (1) This switching does not introduce too much overhead. We, in fact, show that the total induced switching cost is bounded above by the emulation cost. (2) The resulting total work is load balanced across workers—we show this by first demonstrating that the protocol is load-balanced in expectation, and then using concentration to argue our stronger online load-balancing property.

[6] In the actual analysis, we show that it also suffices to use $\mathsf{polylog}(n)$-wise independent randomness to pick this and subsequent assignments.

Finally, note that although communication between *jobs* is being routed through the hypercube, and thus the job communication protocol has small locality, the final load-balanced protocol, being run by *workers*, does *not* have small locality. This is because workers are assigned the role of many different jobs over time, and may possibly speak to a new set of neighbors for each position. (Indeed, over time, each worker will eventually need to speak to every other worker). We next show how to modify this protocol to achieve *locality*, while preserving load-balancing.

Achieving Both Load-Balancing and Locality. In our final step, we show how to modify the above-mentioned protocol to also achieve locality. We modify the protocol to also let *workers* route messages through a low-degree network (on top of the routing in the previous step). This immediately ensures locality. But, we must be careful to ensure that the additional message passing does not break load-balancing.

A natural idea is to again simply pass messages between *workers* along a low-degree hypercube network via Valiant-Brebner (VB) routing [VB81]. Indeed, the low-congestion property will ensure (as before) that routing does not incur too large an overhead for each worker.

However, when analyzing the overall load balance (for workers), we see an inherent distinction between this case and the previous. Previously, the nodes of the hypercube corresponded to *jobs*, each emulated by workers who swap in and out over time. When the underlying jobs protocol required job s to send a message to job t, the resulting message routing induced a cost along a path of neighboring jobs (that is, the workers emulating them), *independent of which workers are currently emulating them.* This independence, together with the fact that a worker passes his job after performing "enough" work for it, enabled us to obtain concentration bounds on overall load balancing over the random assignment of workers to jobs.

Now, the nodes correspond directly to *workers.* When the underlying jobs protocol requires a message transferred from job s to job t, routing along the workers' graph must traverse a path from the *worker currently emulating job* s to the *worker currently emulating job* t, removing the crucial independence property from above. Even worse, workers along the routing path can now incur costs *even if they are not assigned to any job.* In this case, it is not even clear that job passing in of itself will be sufficient to ensure balancing.

To get around these issues, we add an extra step in the VB routing procedure (itself inspired by [VB81]) to break potential bad correlations. The idea is as follows: To route from the worker W_s emulating job s to the worker W_t emulating job t, we first route (as usual) from W_s to a *random* worker W_u, and then from W_u to W_t; i.e., travel from W_s to W_t by "walking into the woods" and back. We may now partition the cost of routing into these two sub-parts, each associated with a single active job (s or t). Now, although workers along the worker-routing path will still incur costs from this routing (even though their jobs may be completely unrelated), the *distribution* of these costs on workers depends only on the identity of the initiating worker (W_s or W_t). We may thus

generalize the previous analysis to argue that if the expectation of work is load-balanced, then it still has concentration in this case.
For a modular analysis, we formalize the required properties of the underlying communication network and routing algorithm (to be used for the s-to-u and u-to-t routing) as a *local load-balanced routing network*, and show that the hypercube network together with VB routing satisfies these conditions.

1.3 Discussion and Future Work

With the explosive growth of data made available in a distributed fashion, and the growth of efficient parallel, distributed algorithms (such as those enabled by MapReduce) to compute on this data, ensuring privacy and security in such large-scale parallel settings is of fundamental importance. We have taken the first steps in addressing this problem by presenting the first protocols for secure multi-party computation, that with only polylogarithmic overhead, enable evaluating PRAM programs on a (large) number of parties' inputs. Our work leaves open several interesting open problems:

Honest Majority. We have assumed that 2/3 of the players are honest. In the absence of a broadcast channel,[7] it is known that this is optimal. But if we assume the existence of a broadcast channel, it may suffice to assume 1/2 fraction honest players.

Asynchrony. Our protocol assumes a synchronous communication network. We leave open the handling of asynchronous communication.

Trading efficiency for security. An interesting avenue to pursue are various tradeoffs between boosted efficiency and partial sacrifices in security. For example, in some settings, it is not detrimental to leak which parties' inputs were used within the computation; in such scenarios, one could then hope to remove the one-time $\Theta(n|x|)$ input preprocessing cost. Similarly, it may be acceptable to reveal the input-specific resources (runtime, space) required by the program on parties inputs; in such cases, we may modify the protocol to take only input-specific runtime and use input-specific memory.
In this work we focus only on achieving standard "full" security. However, we remark that our protocol can serve as a solid basis for achieving such tradeoffs (e.g., a straightforward tweak to our protocol results in input-specific resource use).

Communication complexity. As with all existing generic multi-party computation protocols in the information-theoretic setting, the communication complexity of our protocol is equal to its computation complexity. In contrast, in the computational setting (based on cryptographic assumptions), protocols with communication complexity below the complexity of the evaluated function have been constructed by relying on *fully homomorphic encryption (FHE)* [Gen09] (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.

[7] While the statement of our result makes use of a broadcast channel, as we mention, this channel can also be instantiated with known protocols.

1.4 Overview of the Paper

Section 2 contains preliminaries. In Section 3 we provide our ultimate final theorem, which is built up in the remaining sections. In Section **??** we show a general technique for converting a memory-oblivious program (e.g., an OPRAM) into an oblivious multi-party protocol while maintaining balanced memory requirements across parties. In Sections **??** and **??**, we present compilers for achieving load balancing and communication locality in distributed protocols. Finally, in Section **??**, we describe the compiler taking an oblivious n-party protocol to one secure against $(1/3 - \epsilon)n$ (statically corrupted) malicious parties (while preserving the necessary complexity, load balancing, and locality properties).

2 Preliminaries

2.1 Multi-party Computation (MPC)

Protocol Syntax. We model parties as (parallel) RAM machines. An n-party protocol Φ is described as a collection of n (parallel) RAM programs $(P_i)_{i\in[n]}$, to be executed by the respective parties, which contain the standard instructions in addition to communication instructions $\mathsf{Comm}(i, \mathsf{msg})$, indicating for the executing party to send message msg to party i.
The per-party space, computation, and time complexities of the protocol $\Phi = (P_i)_{i\in[n]}$ are defined directly with respect to the corresponding party's PRAM program P_i, where each Comm is charged as a single computation time step. (See Remark 4 within Section 2.2 for a definition of $\mathsf{space}(P), \mathsf{comp}(P), \mathsf{time}(P)$ for PRAM P). The analogous total protocol complexities are defined as expected: Namely, $\mathsf{space}(\Phi)$ and $\mathsf{comp}(\Phi)$ are the *sums*, $\mathsf{space}(\Phi) = \sum_{i\in[n]} \mathsf{space}(P_i)$, $\mathsf{comp}(\Phi) = \sum_{i\in[n]} \mathsf{comp}(P_i)$, and $\mathsf{time}(\Phi)$ is the *maximum*, $\mathsf{time}(\Phi) = \max_{i\in[n]} \mathsf{time}(P_i)$.

MPC Security. We consider the standard notion of (statistical) MPC security. We refer the reader to e.g. [BGW88] for a more complete description of MPC security within this setting.

2.2 Parallel RAM (PRAM) Programs

A Concurrent Read Concurrent Write (CRCW) m-processor *parallel random-access machine (PRAM)* with memory size n consists of numbered processors $CPU_1, \ldots, CPU_m$, each with local memory registers of size $\log n$, which operate synchronously in parallel and can make access to shared "external" memory of size n.
A PRAM program Π (given m, n, and some input x stored in shared memory) provides CPU-specific execution instructions, which can access the shared data via commands $\mathsf{Access}(r, v)$, where $r \in [n]$ is an index to a memory location, and v is a word (of size $\log n$) or $\perp$. Each $\mathsf{Access}(r, v)$ instruction is executed as:

1. **Read** from shared memory cell address r; denote value by v_{old}.
2. **Write** value $v \neq \perp$ to address r (if $v = \perp$, then take no action).
3. **Return** v_{old}.

In the case that two or more processors simultaneously initiate $\mathsf{Access}(r, v_i)$ with the same address r, then all requesting processors receive the previously existing memory value v_{old}, and the memory is rewritten with the value v_i corresponding to the lowest-numbered CPU i for which $v_i \neq \perp$. We more generally support PRAM programs with a dynamic number of processors (i.e., m_i processors required for each time step i of the computation), as long as this sequence of processor numbers $m_1, m_2, \ldots$ is fixed, public information. The complexity of our OPRAM solution will scale with the number of required processors in each round, instead of the maximum number of required processors.
The *(parallel) time complexity* of a PRAM program Π is the maximum number of time steps taken by any processor to evaluate Π, where each Access execution is charged as a single step. The PRAM complexity of a function f is defined as the minimal parallel time complexity of any PRAM program which evaluates f. We remark that the PRAM complexity of any function f is bounded above by its circuit depth complexity.

Formal PRAM Syntax. In our setting, where we consider evaluating a sequence of programs, it will be convenient to denote the external memory as composed of two separate parts: one part containing *remnant* memory that is maintained from one program to the next, and a second part containing program-specific partial computation data that will be "deallocated" from one program to the next. We will occasionally refer to the first part as the "input database" and the second as the "work database," and addresses etc in the second database will be denoted with superscript "Work" (e.g., $\mathsf{addr}^{\mathsf{Work}}$).
We assume that PRAM programs maintain a regular pattern of computation and data accesses: namely, we assume the program makes precisely one data access to the input database and one access to the Work database after every fixed-size computation step. (Note that this can be forced with only a $\mathsf{polylog}(n)$ multiplicative blowup in complexity, since each data access incurs $\mathsf{polylog}(n)$ complexity cost). This program structure will be important when evaluating PRAMs via secure protocols. In addition, we allow for a subset of CPUs to be *inactive* in certain rounds, in which case their instructions for this round are simply $\emptyset$. However, we restrict our attention to only those PRAMs whose CPU activation schedules (i.e., the identities of which CPUs are active in each round) are fixed and public information.
Combining these observations, we have the syntax in Remark 4. Note that subscripts correspond to timesteps, and superscripts in parentheses identify CPU id.

Remark 4 (Syntax of PRAMs). We assume the following syntax of parallel RAM programs, consisting of a sequence of sub-computations and data accesses for each CPU.

Parallel m-Processor Data Program Π:

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

where each (possibly random) subcomputation $\Pi_t^{(i)}$ has equal and input-independent running time (or is publicly $\emptyset$, indicating that CPU i is inactive in timestep t), all $\Pi_t^{(i)}$ (respectively, $\mathsf{Access}_t^{(i)}, \mathsf{Access}_1^{(i)\mathsf{Work}}$) are executed simultaneously in parallel during timestep t, and $\Pi_t^{(i)}$, $\mathsf{Access}_t^{(i)}$, and $\mathsf{Access}_i^{(t)\mathsf{Work}}$ have the following form:

- $\big(\mathsf{state}, (\mathsf{op}, \mathsf{addr}), (\mathsf{op}^{\mathsf{Work}}, \mathsf{addr}^{\mathsf{Work}}), \mathsf{output}\big) \leftarrow \Pi_t^{(i)}(\mathsf{state}, v, v^{\mathsf{Work}})$.
  On input a CPU state and *two* data values (corresponding to the previous Input and Work Tape data accesses), $\Pi_t^{(i)}$ outputs: an updated state, *two sets* $((\mathsf{op}, \mathsf{addr}), (\mathsf{op}^{\mathsf{Work}}, \mathsf{addr}^{\mathsf{Work}}))$ of data-access operation instruction (denoted "op") and data address values. Note that $\mathsf{op}, \mathsf{op}^{\mathsf{Work}}$ will correspond to one data access in each of the Input and Work memory structures. At the final time step q', the computation will also yield the final output value, output (before this step, $\mathsf{output} = \perp$).
- $(v_1, v_2) \leftarrow \mathsf{Access}_t^{(i)}(\mathsf{op}, v)$.
  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 1.
- $(v_1^{\mathsf{Work}}, v_2^{\mathsf{Work}}) \leftarrow \mathsf{Access}_t^{(i)\mathsf{Work}}(\mathsf{op}^{\mathsf{Work}}, v^{\mathsf{Work}})$.
  Identical to that above, except taking place instead in the Work database.

We define $\mathsf{space}(\Pi)$, $\mathsf{comp}(\Pi)$, and $\mathsf{time}(\Pi)$ as the worst-case space, computation, and (parallel) runtime of the program Π over different inputs. That is, $\mathsf{space}(\Pi)$ is equal to the sum of the largest accessed address in the Input Database and in the Work Database. $\mathsf{comp}(\Pi)$ is equal to the total sum of all computation steps of active CPUs (which, for programs with fixed activation schedules as we consider, is a fixed value). $\mathsf{time}(\Pi)$ is equal to the maximum number of time steps taken by any processor to evaluate Π (where each Access is charged as a single step).

2.3 Oblivious (Parallel) RAM

A machine is said to be *oblivious* if the sequences of memory accesses made by the machine for two inputs with the same running time are identically (or close to identically) distributed. Oblivious RAM compilers—compilers that turn any RAM program Π into a oblivious RAM Π', while only incurring a "small", polylogarithmic, slow-down—were first

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}.

Fig. 1: The function Access, used to implement a Read or Write command held within the CPU as op.

introduced by Goldreich and Ostrovsky [GO96]. The notion of Oblivious *Parallel* RAM (OPRAM) compilers (which perform the same functionality for *parallel* RAM programs) was recently introduced and achieved by Boyle, Chung, and Pass [BCP14].

Definition 1 (Oblivious Parallel RAM). *A polynomial-time algorithm O is an* Oblivious Parallel RAM (OPRAM) *compiler with computational overhead* $\mathsf{comp}(\cdot)$ *and memory overhead* $\mathsf{mem}(\cdot)$*, if for any polynomial $m(n) \in n^{O(1)}$, the compiler O given input $n \in \mathbb{N}$, and a deterministic m-processor PRAM program Π with memory size n, outputs a program Π' with memory size $\mathsf{mem}(n) \cdot n$ such that for any input x, the parallel running time of $\Pi'(m, n, x)$ is bounded by $\mathsf{comp}(n) \cdot T$, where T is the parallel runtime of $\Pi(m, n, x)$, and there exists a negligible function μ such that the following properties hold:*

- **Correctness:** *For any $m, n \in \mathbb{N}$ and any string $x \in \{0,1\}^*$, with probability at least $1 - \mu(n)$, it holds that $\Pi(m, n, x) = \Pi'(m, n, x)$.*
- **Obliviousness:** *For any two PRAM programs Π_1, Π_2, any $m, n \in \mathbb{N}$, and any two inputs $x_1, x_2 \in \{0,1\}^*$, if $|\Pi_1(m, n, x_1)| = |\Pi_2(m, n, x_2)|$, then $\tilde{\Pi}'_1(m, n, x_1)$ is μ-close to $\tilde{\Pi}'_2(m, n, x_2)$ in statistical distance, where $\Pi'_i \leftarrow O(m, n, \Pi_i)$ for $i \in \{1, 2\}$.*

Remark 5 (Collision-freeness). For purposes of load balancing our eventual MPC protocol, it will be useful to use an OPRAM that is *collision-free*, in the sense that in every timestep of an OPRAM-compiled program, different CPUs access distinct memory cells. We remark that the OPRAM construction of [BCP14] satisfies this property. (And, in fact, the techniques of [BCP14] provide a means for converting any OPRAM into one for which collision-freeness holds with $\mathsf{polylog}(n)$ overhead).

Theorem 2 (OPRAM [BCP14]). *There exists a collision-free OPRAM compiler with worst-case computational overhead* $\mathsf{comp}(n) \in \mathsf{polylog}(n)$ *and memory overhead* $\mathsf{mem}(n) \in O(f(n))$ *for any $f \in \omega(\log n)$, for memory size n.*

3 Final Result: Local, Load-Balanced MPC for PRAM

Ultimately, we construct a protocol that securely realizes the ideal functionality $\mathcal{F}_{\mathsf{PRAMs}}$, as in Figure 2. For simplicity of exposition, we assume the total remnant state from one program execution to the next

is bounded in size by the combined input size of all parties. (To support larger shared state, the memory requirements of the protocol must grow accordingly).

Theorem 3 (Main Theorem). *For any constant $\epsilon > 0$ and polynomial parallelism parameter $k = k(n)$, there exists an n-party statistically secure (with error negligible in n) protocol realizing the functionality $\mathcal{F}_{\mathsf{PRAMs}}$, handling $(1/3 - \epsilon)$ fraction static corruptions with the following complexities, where each party is a k-processor PRAM (and where $|x|, |y|$ denote per-party input and output size, $\mathsf{space}(\Pi)$, $\mathsf{comp}(\Pi)$, and $\mathsf{time}(\Pi)$ denote the worst-case space, computation, and (parallel) runtime of Π, and $CPUs(\Pi)$ denotes the number of CPUs of Π):*

- *Computation per party, per Π_j: $\tilde{O}\big(\mathsf{comp}(\Pi_j)/n + |y|\big)$.*
- *Time steps, per Π_j: $\tilde{O}\left(\mathsf{time}(\Pi_j) \cdot \max\left\{1, \frac{CPUs(\Pi)}{nk}\right\}\right)$.*
- *Memory per party: $\tilde{O}\left(|x| + |y| + \max_{j=1}^{N} \mathsf{space}(\Pi_j)/n\right)$.*
- *Communication Locality: $\tilde{O}(1)$.*

given a one-time preprocessing phase with complexity:

- *Computation per party: $\tilde{O}(|x|)$, plus single broadcast of $\tilde{O}(1)$ bits.*
- *Time steps: $\tilde{O}\left(\max\left\{1, \frac{|x|}{k}\right\}\right)$.*

Additionally, the protocol achieves $\mathsf{polylog}(n)$ communication locality, and a strong "online" load-balancing guarantee:

Online Load Balancing: *For every constant $\delta > 0$, with all but negligible probability in n, the following holds at* all times *during the protocol: Let cc and $\mathsf{cc}(W_j)$ denote the total communication complexity and communication complexity of party P_j, comp and $\mathsf{comp}(P_j)$ denote the total computation complexity and computation complexity of party P_j, we have*

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

Oblivious Security. We achieve our result via an intermediate step of (weaker) "*oblivious*" security. Intuitively, an adversary in the oblivious model is not allowed to corrupt any parties, and instead is restricted to seeing the "externally measurable" properties of the protocol (e.g., party response times, communication patterns, etc).

Definition 2 (Oblivious secure MPC). *Secure realization of a functionality F by a protocol in the* oblivious model *is defined by the following real-ideal world scenario:*

Ideal World: Same as standard MPC without corrupted parties. That is, the adversary learns only public outputs of the functionality F evaluated on honest-party inputs.

Real World: Instead of corrupting parties, viewing their states, and controlling their actions (as in the (standard) malicious adversarial setting), the adversary is now limited as an external observer, and is given access only to the following information from the protocol execution:

Ideal Functionality $\mathcal{F}_{\mathsf{PRAMs}}$:
$\mathcal{F}_{\mathsf{PRAMs}}$ running with parties $P_1, \ldots, P_n$ and an adversary proceeds as follows. The functionality maintains longterm storage of parties' inputs $\{x_i\}_{i\in[n]}$ (each of equal size $|x|$), per-CPU state information state_i, and remnant memory $\mathsf{data}^{\mathsf{Remnant}}$ of total size $\mathsf{space}^{\mathsf{Remnant}} \in O(n \cdot |x|)$ transferred from computation to computation.

- Initialize $\mathsf{data}^{\mathsf{Remnant}} \leftarrow \emptyset$ and $\mathsf{state}_i \leftarrow \emptyset$ for each processor $i \in [m]$.
- Input Submission: Upon receiving an input $(commit, sid, input, x_i)$ from party P_i, record the value x_i as the input of P_i.
- Computation: Upon receiving a tuple $(compute, sid, \Pi, \mathsf{space}, \mathsf{time})$ consisting of an m-processor PRAM program Π, a space bound space, and a time bound time, execute Π as $(\mathsf{output}, \mathsf{state}_1, \ldots, \mathsf{state}_m, \mathsf{data}^{\mathsf{Remnant}}) \leftarrow \Pi(x_1, \ldots, x_n, \mathsf{state}_1, \ldots, \mathsf{state}_m, \mathsf{data}^{\mathsf{Remnant}})$ with the current value of state_i for each CPU $i \in [m]$. Send output to all parties.

Fig. 2: The ideal functionality $\mathcal{F}_{\mathsf{PRAMs}}$, corresponding to secure computation of a sequence of adaptively chosen PRAMs on parties' inputs.

- *Activation Patterns: Complete list of tuples of the following form.*
  - $(\mathsf{timestep}, \mathsf{party\text{-}id}, \mathsf{compute\text{-}time})$*: Specifying all local computation times of parties.*
  - $(\mathsf{timestep}, \mathsf{party\text{-}id}, \mathsf{local\text{-}mem\text{-}addr})$*: Specifying all memory access patterns of parties.*
- *Communication Patterns: Complete list of tuples of the following form.*
  - $(\mathsf{timestep}, \mathsf{sndr\text{-}id}, \mathsf{rcvr\text{-}id}, \mathsf{msg\text{-}len})$*: Specifying all sender-receiver pairs, in addition to the corresponding communicated message bit-length.*

The output of the real-world experiment consists of the outputs of the (honest) parties, in addition to an arbitrary PPT function of the adversary's view at the conclusion of the protocol.

Security: As usual, (statistical) security is defined by requiring for every PPT adversary $\mathcal{A}$ in the real-world execution, there exists a PPT ideal-world adversary $\mathcal{S}$ for which for every environment $\mathcal{Z}$, it holds that

$$\mathsf{output}_{\mathsf{Real}}(1^k, \mathcal{A}, \mathcal{Z}) \stackrel{s}{\cong} \mathsf{output}_{\mathsf{Ideal}}(1^k, \mathcal{S}, \mathcal{Z}).$$

References

[AAK08] Baruch Awerbuch, Yossi Azar, and Rohit Khandekar. Fast load balancing via bounded best response. In *Proceedings of the Nineteenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2008, San Francisco, California, USA, January 20-22, 2008*, pages 314–322, 2008.

[ACMR95] Micah Adler, Soumen Chakrabarti, Michael Mitzenmacher, and Lars Eilstrup Rasmussen. Parallel randomized load balancing (preliminary version). In *Proceedings of the Twenty-Seventh Annual ACM Symposium on Theory of Computing, 29 May-1 June 1995, Las Vegas, Nevada, USA*, pages 238–247, 1995.

[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.

[BCP14] Elette Boyle, Kai-Min Chung, and Rafael Pass. Oblivious parallel ram. Cryptology ePrint Archive, Report 2014/594, 2014.

[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.

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

[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.

[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.

[MPS02] Michael Mitzenmacher, Balaji Prabhakar, and Devavrat Shah. Load balancing with memory. In *43rd Symposium on Foundations of Computer Science (FOCS 2002), 16-19 November 2002, Vancouver, BC, Canada, Proceedings*, pages 799–808, 2002.

[MR98] S. Muthukrishnan and Rajmohan Rajaraman. An adversarial model for distributed dynamic load balancing. In *Proceedings of the Tenth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '98, pages 47–54, 1998.

[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.

[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.