Differentially Private Access Patterns in Secure Computation

Sahar Mazloom[1] and S. Dov Gordon[1]

[1]George Mason University

October 17, 2017

Abstract

We explore a new security model for secure computation on large datasets. We assume that two servers have been employed to compute on private data that was collected from many users, and, in order to improve the efficiency of their computation, we establish a new tradeoff with privacy. Specifically, instead of claiming that the servers learn nothing about the input values, we claim that what they do learn from the computation preserves the differential privacy of the input. Leveraging this relaxation of the security model allows us to build a protocol that leaks some information in the form of access patterns to memory, while also providing a formal bound on what is learned from the leakage.

We then demonstrate that this leakage is useful in a broad class of computations. We show that computations such as histograms, PageRank and matrix factorization, which can be performed in common graph-parallel frameworks such as MapReduce or Pregel, benefit from our relaxation. We implement a protocol for securely executing graph-parallel computations, and evaluate the performance on the three examples just mentioned above. We demonstrate marked improvement over prior implementations for these computations.

1 Introduction

Privacy and utility in today's Internet is a tradeoff, and for most users, utility is the clear priority. We continue to contribute greater amounts of private data to an increasing number of entities in exchange for a wider variety of services. From a theoretical perspective, we can maintain privacy and utility if these entities are willing and able to compute on encrypted data. The theory of secure computation has been around since the earliest days of modern cryptography, but the practice of secure computation is relatively new, and still lags behind the advancements in data-mining and machine learning that have helped to create today's tradeoff.

Recently, we have seen some signs that the gap might be narrowing. The advancements in the field of secure computation have been tremendous in the last decade. The first implementations computed roughly 30 circuit gates per second, and today they compute as many as 6 million per second [1]. Scattered examples of live deployments have been referenced repeatedly, but most recently, in one of the more promising signs of change, Google has started using secure computation to help advertisers compute the value of their ads, and they will soon start using it to securely construct machine learning classifiers from mobile user data [2]. A separate, more recent line of research also offers promise: the theory and techniques of differential privacy give service providers new mechanisms for aggregating user data in a way that reasonably combines utility and privacy. The guarantee of these mechanisms is that, whatever can be learned from the aggregated data, the amount that it reveals about any single user input is minimal. The Chrome browser uses these techniques when aggregating crash reports [3], and Apple claims to be employing them for collecting usage information from mobile devices. In May, 2017, Senator Ron Wyden wrote an open letter to the commission on evidence-based policymaking urging that both secure computation and differential privacy be employed by "agencies and organizations that seek to draw public policy related insights from the private data of Americans [4]."

The common thread in these applications is large scale computation, run by big organizations, on data that has been collected from many individual users. To address this category of problems, we explore new improvements for two-party secure computation, carried out by two dedicated computational servers, over secret shares of user data. We use a novel approach: rather than attempting to improve on known generic constructions, or tailoring a new solution for a particular problem, we instead explore a trade-off between efficiency and privacy. Specifically, we propose a model of secure computation in which some small information is leaked to the computation servers, but this leakage is proven to preserve differential privacy for the users that have contributed data. More technically, the leakage is a random function of the input, revealed in the form of access patterns to memory, and the output of this function does not change "by too much" when one user's input is modified or removed.

The question of what is leaked by access patterns to input during a computation is central to secure computation. Although the circuit model of computation allows us to skirt the issue, because circuits are data oblivious, when computing on large data there are better ways of handling the problem, the most well-studied being the use of secure two-party ORAM [5, 6, 7, 8, 9]. However, when looking at very large data sets, it is often the case that both circuits and ORAM are too slow for practical requirements, and there is strong motivation to look for better approaches. In the area of encrypted search, cryptographers have frequently proposed access-pattern leakage as a tradeoff for efficiency [10, 11, 12, 13]. Unfortunately, analyzing and quantifying the leakage caused by the computation's access pattern is quite difficult, as it depends heavily on the specific computation, the particulars of the data, and even the auxiliary information of the adversary. Furthermore, recent progress on studying this leakage has mostly drawn negative conclusions, suggesting that a lot more is revealed than we might originally have hoped [14, 15, 16, 17, 18]. Employing the definition of differential privacy as a way to bound the leakage of our computation allows us to offer an efficiency / privacy tradeoff that cryptographers have been trying to provide, while quantifying, in a rigorous and meaningful way, precisely what we have leaked.

1.1 Graph-Parallel Computations

In designing our protocol, we aimed to strike another balance as well; one between generality and efficiency. Accordingly, we have identified a broad class of highly parallelizable computations that are amenable to the privacy tradeoff we propose. Recently, Nikolaenko et al. constructed a tailored secure computation for performing matrix factorization of sparse matrices, with application to building recommendation systems while user input remains hidden [19]. A generic circuit for matrix factorization would grow with the size of the matrix, rather with the number of entries in the matrix. When dealing with large, sparse matrices of size $n \times m$ but with only $O(n + m)$ entries, the quadratic growth in circuit size is prohibitive. By instead performing a sequence of generic computations, which includes several oblivious sorts over the secret-shared entries of the matrix, they are able to construct a computation that requires $O((n+m)\log^2(n+m))$ garbled AND gates, instead of $O(nm)$ AND gates. The $\log^2$ factor comes from the cost of performing oblivious sorting on the input data.

Generalizing this work, Nayak et al., recognized that a similar, though more elaborate, secure computation could be used to perform parallelizable computation on graph-structured data while the input remains encrypted [20]. When computing on plaintext data, frameworks such as MapReduce, Pregel, GraphLab and PowerGraph have very successfully enabled developers to leverage large networks of parallelized CPUs [21, 22, 23, 24]. The latter three mentioned systems are specifically designed to support computations on data that resides in a graph, either at the nodes or edges. The computation proceeds by iteratively gathering data from incoming edges to the nodes, performing some simple computation at the node, and pushing the data back to the outgoing edges. This simple iterative procedure captures many important computational tasks, including histogram, matrix factorization and page-rank, which we focus on here (as did Nayak et al.), as well as Markov random field parameter learning, parallelized Gibbs samplers, and name entity resolution, to name a few more. Nayak et al. built a parallelizable system for securely computing on graph-structured data. The complexity of their computation is similar to that of Nikolaenko et al., requiring $O(|E| + |V|)\log^2(|E| + |V|)$ garbled AND gates.

1.2 A Connection to Differential Privacy

It turns out that there is a natural connection between building differentially private access patterns, and these graph-parallel frameworks. The memory access pattern induced by this computation is easily described: during the gather stage, each edge is touched when fetching the data, and the identifier of the adjacent node is exposed when copying the data. A similar pattern is revealed during the scatter phase. (The computation performed during the apply phase is typically very simple, and can be executed in a circuit, which is memory oblivious.) In the computations we perform, each user is represented by a node in the graph, and provides the data that starts out on the edges adjacent to that node. For example, in a recommendation system, the graph is bipartite, each user is represented by one node on the left, each node on the right represents an item that users might review, and the edges are labeled with scores indicating the user's review of an item. The access pattern just described would reveal exactly which items every user reviewed! Our first observation is that if we use a secure computation to obliviously shuffle all of the edges in between the gather and apply phases, we break the correlation between the nodes, and the only thing revealed to the computing parties is a *histogram* of how many times each node is accessed – i.e. a count of each node's in-degree and out-degree. When building a recommendation system, this would reveal how many items each user reviewed, as well as how many times each item was reviewed. Fortunately, histograms are the canonical problem for differential privacy. Our second observation is that we can shuffle in dummy edges to help obscure this information, and by sampling the dummy edges from an appropriate distribution (which has to be done within a secure computation), we can claim that the degrees of each node remain differentially private.

Differentially Private Output: As is typical in secure computation, we are concerned here with *how* to securely compute some agreed upon function, rather than *what* function ought to be computed. In other words, we view the question of what the output itself might reveal about the input to be beyond scope of our work. Our concern is only that the process of computing that output does not reveal too much. Admittedly, if the parties that perform the computation will ultimately learn something that breaks differential privacy, it's not clear why we would insist that the process of performing that computation should preserve differential privacy. We could resolve this tension by insisting that the output of all computations preserve differential privacy, which, at least for the class of computations we support in this work, would not be that hard to do. Indeed, in the specific case of histograms, which we present as an example in Section 3, adding differentially private noise to the output is substantially *more efficient* than preserving an exact count.

Nevertheless, we take a different approach here. In all of our computations, the output of each server is a secret share of the desired output. The question of where to deliver these shares is left to the user, though we can imagine several scenarios in which the party receiving the shares might not require that the result preserve differential privacy. For example, it might be that the users of the system have entrusted their data to a single entity, such as a government agency, and that this entity is now outsourcing a complicated learning task to the computation servers, with the requirement that they not learn about the underlying data. We might even imagine that the shares are never reconstructed, but are used later inside another secure computation in order to make decisions that are driven by the output. The advantage of separating out the question of what is revealed by the output is that it allows us to compare apples to apples: we can isolate the question of what can be gained in performance when we employ our proposed tradeoff. In particular, as we described above, the class of computations that we support has been studied in prior work. In Section 5 we will give a direct comparison to the performance in that work, and demonstrate substantial improvement. Modifying the computational tasks would make such a comparison difficult.

1.3 Contributions and Related Work

Contributions. We make several new contributions, of both a theoretical and a practical nature.

Introducing the model. As cryptographers have attempted to support secure computation on increasingly large datasets, they have often allowed their protocols to leak some information to the computing parties in the form of access patterns to memory. This is especially true in the literature on encrypted search. The idea of bounding the leakage in a formal way, using the definitions from literature on differential privacy, is novel and important.

More efficient asymptotic analysis. The relaxation we introduce enables us to improve the asymptotic complexity of the target computations by a factor of $\log n$. While the more practical construction of Nayak et al. [20] has running time $O(n \log^2 n)$, if they instead use the best known asymptotic result for oblivious sorting, their protocol becomes less practical, but in fact runs in time $O(n \log n)$. In contrast, while our practical construction runs in time $O(n \log n)$, if we are willing to perform encryption and decryption inside a garbled circuit, we can modify our construction to achieve $O(n)$ run-time. We will provide the details of this improvement in the full version of this paper.

An implementation. We demonstrate that the asymptotic improvements lead to tangible gains. We have implemented our system, and compared the results to the system of Nayak et al. [20]. We demonstrate up to a 20X factor improvement in the number of garbled AND gates required in the computation, while preserving differential privacy with strong parameters: $\epsilon = .3$ and $\delta = 2^{-40}$.

Related Work. Nayak et al. [20] were the first to consider parallelizing the secure computation of graph-structured data, and we use their work as the basis for evaluating the efficiency of our own construction. Their construction generalized the protocol designed by Nikolaenko for matrix factorization of sparse matrices. However, since the protocol of Nayak et al. out-performs theirs, we only compare to the former. In both works, the constructions are fully secure, in contrast to our own construction that intentionally leverages some bounded leakage in exchange for improved efficiency.

Concurrent with our own work, Papadimitriou et al. [25] also build a system for secure computation of graph-structured data, and they even offer differential privacy of the output. However, as we mention above, we view this property as being orthogonal to the question of whether the protocol itself leaks differentially private information. Indeed, their construction is fully-secure, so they do not leverage differential privacy in order to construct a more efficient protocol, as we do. Their model also differs from our own (and those of Nayak et al. and Nikolaenko et al.) in that they consider a set of parties that are unwilling to entrust their input to computation servers. Instead, each party holds their own piece of the graph-structured data, and the authors construct a multi-party protocol where the communication patterns hide the structure of the graph. This setting is more challenging, and the result is much less efficient; the authors do not provide a comparison of their performance to that of Nayak et al., so we do not provide one here.

In an unpublished work (which pre-dates our own), Kellaris et al. construct differentially private storage systems that allow a client to outsource their data to an untrusted server while supporting arbitrary queries to the data [26]. They define a model in which the access pattern to storage leaks information to the server, but prove that the leakage preserves the differential privacy of the users. Communication with the authors about their work helped to inspire our own ideas. The primary difference between their work and ours is that we explore secure computation on the data, rather than search. Since database queries can be viewed as a particular instance of secure computation, one could view the idea of leveraging differentially private leakage in secure computation as a generalization of their idea to leverage such leakage in the setting of encrypted search. In practical terms, though, this is mostly inaccurate, because of many other differences in the model: they assume a single computation server, a client that provides all of the (pre-processed) data, and require computation times that are sub-linear. Our protocol is not meant to capture arbitrary computation, and, in particular, it runs in time super-linear in the input size, so it only provides benefit when compared with protocols that are super-linear.

In another unpublished work (which also pre-dates our own), Wagh et al. define and construct differentially private ORAM [27]. This is an oblivious memory structure that guarantees that two "neighboring access patterns" are indistinguishable. This is an extremely interesting relaxation of the standard definition for ORAM, in which all access patterns must be indistinguishable. Furthermore, as the authors point out, their construction composes: allowing for a degradation in the privacy parameter, they can provide differential privacy for any two access patterns of bounded distance from one another. Although their construction is in the client/server model, in which all of the data is known to the client, and security / privacy is only guaranteed with respect to the server, using standard techniques we could execute their ORAM in a two-party computation to achieve precisely the privacy / efficiency tradeoff we have proposed, for all RAM-model computations. However, the authors did not address this question, and in particular did not define secure computation with differentially private leakage, which we view as one of our contributions. More importantly, the resulting construction, while more general than our own, would also be much less efficient than ours. The primary savings in Root ORAM (as compared to

Path ORAM [28]) stems from a modification to the read/write operation: instead of always assigning the last touched memory item to a new leaf node in the binary tree, in Root ORAM the mapping is occasionally left untouched, which allows for operating with a smaller stash. We have not attempted to implement it, and a good estimation is hard to make, but, using $\epsilon = .3$ with Root ORAM, for about 3 in 10,000 memory accesses, a data item would not be re-mapped to a new leaf node in the ORAM. On a computation involving 10^6 edges and 4000 nodes, this amounts to 300 data lookups (on average) out of 10^6. Although the authors do not give a direct analysis of the required stash size for preventing an over-flow event (and, in particular, they do not compare the needed stash size with that of Path ORAM), we are quite certain that the overhead of implementing garbled ORAM would make this far less efficient than our own construction, in which there are no hidden constants in the asymptotic notation. (Indeed, even if the stash were eliminated entirely, we think it is very unlikely that garbled ORAM would outperform our own construction.)

2 Definitions and Notation

Throughout the paper, we use the following notations. We view a database as a multi-set of elements drawn from some fixed set S. We represent the database by a function $D : S \to \mathbb{N}$, and we use $|D|$ in the natural way to mean $\sum_{i \in S} D(i)$. We use $\mathcal{DB}_i$ to denote the set of all databases of size i, and $\mathcal{DB} = \bigcup_i \mathcal{DB}_i$. We consider two databases D_1 and D_2 to be adjacent if the two multi-sets differ in exactly one element. Technically, $|D_1 \setminus D_2| = 1$ and $|D_2 \setminus D_1| = 0$. For simplicity, we will sometimes denote this by $|D_1 - D_2| = 1$. For example, $D_1 = \{A, A, B, B, B, C, C\}$ and $D_2 = \{A, A, B, B, C, C\}$ have distance 1.

We let $\langle x \rangle$ denote a variable which is XOR secret-shared between parties. Arrays have a public length and are accessed via public indices; we use $\langle x \rangle_i$ to specify element i within a shared array, and $\langle x \rangle_{i:j}$ to indicate a specific portion of the array containing elements i through j, inclusive. When we write $\langle x \rangle \leftarrow c$, we mean that both users should fix their shares of x (using some agreed upon manner) to ensure that $x = c$. For example, one party might set his share to be c while the other sets his share to 0.

2.1 Differential Privacy

We use the definition that appears in [29].

Definition 1 *A randomized algorithm $\mathcal{F} : \mathcal{D} \to \mathcal{R}_{\mathcal{F}}$, with an input domain $\mathcal{D}$ that is the set of all databases and output $\mathcal{R}_{\mathcal{F}} \subset \{0,1\}^*$ is (ϵ, δ)-differentially private if for all $T \subseteq \mathcal{R}_{\mathcal{F}}$ and $\forall D_1, D_2 \in \mathcal{D}$ such that $|D_1 - D_2| \leq 1$:*

$$\Pr[\mathcal{F}(D_1) \in T] \leq e^{\epsilon} \Pr[\mathcal{F}(D_2) \in T] + \delta$$

where the probability space is over the coin flips of the mechanism $\mathcal{F}$.

Given a database $\mathcal{D}$ over a set V, defined as above, we define our mechanism $\mathcal{F}_{\epsilon,\delta}(\mathcal{D})$ to output a "noisy" database, $\widehat{\mathcal{D}}$, where for each $i \in V$, $\widehat{\mathcal{D}}(i) = \mathcal{D}(i) + \gamma_i$, and each γ_i is drawn independently from a shifted geometric distribution, parameterized by a probability p, and denoted by $\mathfrak{D}_p$. The shift ensures that negative values are negligible likely to occur. This is necessary because the noisy set will determine our access pattern to memory, and we cannot accommodate a negative number of accesses. More specifically, we will define below a "shift function" $\alpha : \mathbb{R} \times \mathbb{R} \to \mathbb{N}$ that maps every (ϵ, δ) pair to a natural number. When ϵ and δ are fixed, we will simply use α to denote $\alpha(\epsilon, \delta)$, and $\mathcal{F}$ to denote $\mathcal{F}_{\epsilon,\delta}$. Intuitively, we sample γ_i by flipping a biased coin p until it comes up heads. We flip one more unbiased coin to determine the sign of the noise, and then add the result to α. We will determine p based on ϵ and δ. Formally, γ_i is sampled as follows:

$$\Pr[\gamma_i = \alpha] = \frac{p}{2}$$

$$\forall k \in \mathbb{N}, k \neq 0 : \Pr[\gamma_i = \alpha + k] = \frac{1}{2}(1 - \frac{p}{2})p(1-p)^{|k|-1}.$$

As just previously described, we view p as the stopping probability. However, in the first coin flip, we stop with probability $p/2$. We note that this is a slight modification to the normalized 2-sided geometric distribution, which would typically be written as $\Pr[\gamma_i = \alpha + k] = \frac{1}{2-p}p(1-p)^{|k|}$. The advantage of the distribution as it is written above is that it is very easy to sample in a garbled circuit, so long as p is an inverse power of 2; normalizing by $\frac{1}{2-p}$ introduces problems of finite precision and greatly complicates the sampling circuit. An analysis of this mechanism, including concrete settings of the parameters, and a brief description of how we implement it in a garbled circuit, appears in Section 4.

We note that with some probability that is dependent on the choice of α, for $\widehat{\mathcal{D}} = \mathcal{F}(\mathcal{D})$, $\exists i \in V, \widehat{\mathcal{D}}(i) < 0$, which leaves us with a bad representation of a multi-set. We therefore modify the definition of $\mathcal{F}$ to output $\emptyset$ whenever this occurs, and we always choose α so that this occurs with probability bound by $\delta = 2^{-40}$.

2.2 Secure computation with differentially private access patterns

Input model: We try to keep the definitions general, as we expect they will find application beyond the space of graph-structured data. However, we use notation that is suggestive of computation on graphs, in order to keep our notation consistent with the later sections. We assume that two computation servers have been entrusted to compute on behalf of a large set of users, $\mathcal{V}$, with $|\mathcal{V}| = n$, and having sequential identifiers, $1, \ldots, n$. Each user i contributes data v_i. They might each entrust their data to one of the two servers (we call this the *disjoint collection setting*), or they might each secret-share their input with the two-servers (*joint collection setting*). In the latter case, we note that both servers learn the size of each v_i but neither learns the input values; in the former case, each server learns a subset of the input values, but learns nothing about the remaining inputs (other than the sum of their sizes).[1] Below we will define two variant security notions that capture these two scenarios.

In all computations that we consider in our constructions, the input is represented by a graph. In every case, each user is represented as a node in this graph, and each user input is a set of weighted, directed edges that originate at their node. In some applications, the graph is bipartite, with user nodes on the left, and some distinct set of item nodes on the right: in this case, all edges go from user nodes to item nodes. In other applications, there are only user nodes, and every edge is from one user to another. In the joint collection setting, we can leak the out-degree of each node, which is the same as the user input size, but must hide (among other things) the in-degree of each node. In the disjoint collection setting, the protocol has to hide both the in-degree and out-degree of each node. In the case of a bipartite graph, it is publicly known that the in-degree of every user is 0 (i.e. items have no input). In the joint collection setting, this knowledge allows for some improvement in efficiency that we will leverage in Section 5.

Secure computation with leakage: In this section, we define secure computation with differentially private leakage. For simplicity, we start with a standard definition of semi-honest security[2], but make two important changes. The first change is that we allow certain leakage in the ideal world, in order to reflect what is learned by the adversary in the real world through the observed access pattern on memory. The leakage function is a randomized function of the inputs. The second change is an additional requirement that this leakage function be proven to preserve the differential privacy for the users that contribute data. Our ideal world experiment is as follows. There are two parties, P_1 and P_2, and an adversary $\mathcal{S}$ that corrupts one of them. The parties are given input, as described above; we use V_1 and V_2 to denote the inputs of the computing parties, regardless of whether we are in the joint collection setting or the disjoint collection setting, and we let $V = \{v_1, \ldots, v_n\}$ denote the user input. Technically, in the joint collection setting, $V = V_1 \oplus V_2$, while in the disjoint collection setting, $V = V_1 \cup V_2$. Each computing party submits their input to the ideal functionality, unchanged. The ideal functionality reconstructs the n user inputs, $v_1, \ldots, v_n$, either by taking the union of the inputs submitted by the computation servers in the disjoint collection setting, or by reconstructing the input set from the provided secret shares in the joint collection setting. The ideal functionality then outputs $f_1(v_1, \ldots, v_n)$ to P_1 and

---

[1] We note that the disjoint collection setting corresponds to the "standard" setting for secure computation where each computing party contributes one set of inputs. Just as in that setting, each of the two computing parties could pad their inputs to some maximum size, hiding even the sum of the user input sizes. In fact, we could have them pad their inputs using a randomized mechanism that preserves differential privacy, possibly leading to smaller padding sizes, depending on what the maximum and average input sizes are. We don't explore this option further in this work.

[2] We stress that our use of differentially private leakage leads to gains in the *circuit construction*, so we could use *any* generic secure computation of Boolean circuits, including those that are maliciously secure, and benefit from the same gains. See more details below.

$f_2(v_1, \ldots, v_n)$ to P_2. These outputs might be correlated, and, in particular, in our own use-cases, each party receives a secret share of a single function evaluation: $\langle f(v_1, \ldots, v_n)\rangle_1, \langle f(v_1, \ldots, v_n)\rangle_2$. The ideal functionality also computes $\mathcal{L}(D)$ and provides this, along with $\sum_{i\in\mathcal{V}} |v_i|$ to $\mathcal{S}$.[3] Additionally, depending on the choice of security definition, the ideal functionality might or might not give the simulator, $\forall i \in \mathcal{V}$, $|v_i|$.

Our protocols are described in a *hybrid world*, in which the parties are given access to several secure, ideal functionalities. In our implementation, these are replaced using generic constructions of secure computation (i.e. garbled circuits). Relying on a classic result of Canetti [30], when proving security, it suffices to treat these as calls to a trusted functionality. In the definitions that follow, we let $\mathcal{G}$ denote an appropriate collection of ideal functionalities.

As is conventionally done in the literature on secure computation, we let $\textsc{hybrid}^{\mathcal{G}}_{\pi,\mathcal{A}(z)}(V_1, V_2, \kappa)$ denote a joint distribution over the output of the honest party and the view of the adversary $\mathcal{A}$ with auxiliary input $z \in \{0,1\}^*$, when the parties interact in the hybrid protocol $\pi^{\mathcal{G}}$ on inputs V_1 and V_2, each held by one of the two parties, and computational security parameter κ. We let $\textsc{ideal}_{\mathcal{F},\mathcal{S}(z,\mathcal{L}(V),\forall i\in V:|v_i|)}(V_1, V_2, \kappa)$ denote the joint distribution over the output of the honest party and the view output by the simulator $\mathcal{S}$ with auxiliary input $z \in \{0,1\}^*$, when the parties interact with an ideal functionality $\mathcal{F}$ on inputs V_1 and V_2, each submitted by one of the two parties, and security parameters κ. We define the joint distribution $\textsc{ideal}_{\mathcal{F},\mathcal{S}(z,\mathcal{L}(V),\sum_{i\in V}|v_i|)}(V_1, V_2, \kappa)$ in a similar way, the only difference being that the simulator is given the sum of the input sizes and not the value of each input size.

Definition 2 *Let $\mathcal{F}$ be some functionality, and let π be a two-party protocol for computing $\mathcal{F}$, while making calls to an ideal functionality $\mathcal{G}$. π is said to* securely compute $\mathcal{F}$ in the $\mathcal{G}$-hybrid model with $\mathcal{L}$ leakage, known input sizes, and $(\kappa, \epsilon, \delta)$-security *if $\mathcal{L}$ is (ϵ, δ)-differentially private, and, for every PPT, semi-honest, non-uniform adversary $\mathcal{A}$ corrupting a party in the $\mathcal{G}$-hybrid model, there exists a PPT, non-uniform adversary $\mathcal{S}$ corrupting the same party in the ideal model, such that, on any valid inputs V_1 and V_2*

$$\left\{\textsc{hybrid}^{\mathcal{G}}_{\pi,\mathcal{A}(z)}(V_1, V_2, \kappa)\right\}_{z\in\{0,1\}^*,\kappa\in\mathbb{N}} \stackrel{c}{\equiv}$$
$$\left\{\textsc{ideal}^{(1)}_{\mathcal{F},\mathcal{S}(z,\mathcal{L}(V),\forall i\in V:|v_i|)}(V_1, V_2, \kappa)\right\}_{z\in\{0,1\}^*,\kappa\in\mathbb{N}} \qquad (1)$$

The above definition is the one that we use in our implementations. However, in Section 4 we also describe a modified protocol that achieves the stronger security definition that follows, where the adversary does not learn the sizes of individual inputs. This property might be desirable (or maybe even essential) in the disjoint collection model, where users have not entrusted one of the two computing parties with their inputs, or even the sizes of their inputs.

---

[3] In the joint collection setting, the simulator can infer this value from the size of the input that was submitted to the ideal functionality. But it simplifies things to give it to him explicitly.

Definition 3 *Let $\mathcal{F}$ be some functionality, and let π be a two-party protocol for computing $\mathcal{F}$, while making calls to an ideal functionality $\mathcal{G}$. π is said to* securely compute $\mathcal{F}$ in the $\mathcal{G}$-hybrid model with $\mathcal{L}$ leakage, and $(\kappa, \epsilon, \delta)$-security *if $\mathcal{L}$ is (ϵ, δ)-differentially private, and, for every PPT, semi-honest, non-uniform adversary $\mathcal{A}$ corrupting a party in the $\mathcal{G}$-hybrid model, there exists a PPT, non-uniform adversary $\mathcal{S}$ corrupting the same party in the ideal model, such that, on any valid inputs V_1 and V_2*

$$\left\{\mathrm{HYBRID}^{\mathcal{G}}_{\pi,\mathcal{A}(z)}\left(V_1, V_2, \kappa\right)\right\}_{z\in\{0,1\}^*,\kappa\in\mathbb{N}} \overset{c}{\equiv}$$
$$\left\{\mathrm{IDEAL}^{(2)}_{\mathcal{F},\mathcal{S}(z,\mathcal{L}(V),\sum_{i\in V}|v_i|)}(V_1, V_2, \kappa)\right\}_{z\in\{0,1\}^*,\kappa\in\mathbb{N}} \tag{2}$$

Extensions to other models: Extending these definitions to other common models, including those with malicious adversaries and/or multi-party computation is straightforward, so we do not provide redundant detail. However, we stress that our improvements over prior work are *at the circuit level*

3 A Differentially Private Protocol for Computing Histograms

To illustrate our main idea, we describe an algorithm that computes the data histogram (counting or data frequency) with differentially private access patterns. Although this computation can be formalized in the context of our general framework, it is instructive to demonstrate some of the main technical ideas with this simple example before considering how they generalize (which we do in Section 4). We defer a discussion about security until we present the more general protocol.

In this computation, we assume that each user in the system contributes a single input value, $x_i \in S$, where we call the set S the set of *types*. The computation servers (parties) each begin the computation with secret shares of the input array, denoted by $\langle real \rangle$ in output a secret share of $|S|$ counters, where each counter contains the exact count of the number of inputs of the corresponding type. The full protocol specification appears in Figure 1.

The two parties begin by generating some number of dummy inputs. The functionality for this is described in the left of Figure 2, and it is realized using a generic secure two-party computation. As part of this computation, the parties have to securely sample the distribution $\mathfrak{D}_p$, which is also done using a generic protocol for secure computation: we implemented one of the circuits described by Dwork et al. [31]. The output of $DumGen_{p,\alpha}$ is a secret sharing of values in $S \cup \{\perp\}$: the size of the output is $2\alpha|S|$, where α is some constant determined by the desired privacy values ϵ and δ (see Section 4). The number of dummy items of each type is random, and neither party should learn this value; shares of $\perp$ are used to pad the number of dummy items of each type until they total 2α.

Each party locally concatenates their share of the real input array with their share of the dummy values. They also initialize shares of an array of flags, which will be used to keep track of which items are real and which are dummy. They then shuffle the real and dummy items together using an oblivious shuffle. We implement this using two sequential, generic secure computations of the Waksman permutation network, with each party randomly choose one of the two permutations. The same permutations are used to shuffle the flags, ensuring that the flags are "moved around with" the items. We note that all secret shares are updated during the process of shuffling, so while the parties knew which items and flags were real and which were not before the shuffle, they have no way of knowing this after they receive fresh shares of the shuffled items and flags.

The parties now open their shares of the data types, while leaving the flag values unknown. This is where our protocol leaks some information: revealing the data types allows the parties to see a noisy sum of the number inputs of each type. On the other hand, this is also where we gain in efficiency: the remainder of the protocol requires only a linear scan over the data array, with a small secure computation for each element in order to update the appropriate counter value. More specifically, the parties iterate through the shuffled array. On data type i, they fetch their shares of the counter for type i from memory and perform a secure computation that adds the (reconstructed) flag value to the (reconstructed) counter. If the item was a real item, the parties receive fresh shares of the incremented counter value, while if it was dummy, they receive fresh shares of the counter's prior value; in either case, they never learn whether they fetched that counter from memory because of a real input value, or because of a dummy value.

Simple extensions: In Section 4 we show how to generalize this protocol to the wider function class. However, we note that in this specific case, if we did want to add noise to the output, we could simply instruct the servers to count the number of times each counter is accessed. They would not need to update the counter values through a secure computation, so this would be a (slightly) faster protocol. The output would have one-sided noise, but they could simply subtract off α from each counter at the end to get a more accurate estimate of the counts. We also note that the protocol in Figure 1 can be applied to other similar computations such as taking averages or sums over r values of $|S|$ types. For example, if each user contributed a salary value and a zip-code, we could use the above method for computing the average salary in each zip-code: instead of incrementing the counter by 1 when we encounter a real item, we simply increment it by the value of the secret-shared input. In this case, though, the access pattern alone does not suffice for creating noisy output. If that is desired for these computations, the noise would have to be generated independently, through a secure computation for sampling the desired distribution, and then added obliviously to the output.

Differentially Private Histogram Protocol

Input: Each party, P_1 and P_2, receives a secret-share of real items $\langle \mathsf{real} \rangle$

Computation:

```
$\langle \mathsf{Counter} \rangle_{1:|S|} \leftarrow 0$
$\langle \mathsf{dummy} \rangle_{1:d} \leftarrow DumGen_{p,\alpha}$
$\langle \mathsf{flag}_0 \rangle_{1:d} \leftarrow 0$ ,   $\langle \mathsf{flag}_1 \rangle_{1:r} \leftarrow 1$
$\langle \mathsf{data} \rangle_{1:(r+d)} = \langle \mathsf{real} \rangle || \langle \mathsf{dummy} \rangle$
$\langle \mathsf{flag} \rangle_{1:(r+d)} = \langle \mathsf{flag}_1 \rangle || \langle \mathsf{flag}_0 \rangle$
$\langle \widehat{\mathsf{data}} \rangle \leftarrow \mathcal{F}_{\mathsf{Shuffle}}(\langle \mathsf{data} \rangle, \langle \rho \rangle)$
$\langle \widehat{\mathsf{flag}} \rangle \leftarrow \mathcal{F}_{\mathsf{Shuffle}}(\langle \mathsf{flag} \rangle, \langle \rho \rangle)$
$\widehat{\mathsf{data}} \leftarrow \mathsf{Open}(\langle \widehat{\mathsf{data}} \rangle)$
for $i = 1 \ldots (r+d)$
    $\mathcal{F}_{\mathsf{add}}(\langle \mathsf{flag} \rangle_i, \langle \mathsf{Counter} \rangle_t)$ where $t = \widehat{\mathsf{data}}_i$
```

Output: $\langle \mathsf{Counter} \rangle$

Figure 1: A protocol for two parties to compute a histogram on secret-shared data with an access pattern that preserves differential privacy.

4 OblivGraph: Differentially Private protocol for Secure Graph-Parallel Computation

When considering how the protocol from the previous section might be generalized, it is helpful to recognize the essential property of the computation's access pattern that we were leveraging. When computing a histogram, the access pattern to memory exactly leaks a histogram of the input! This might sound like a trivial observation, but it is in fact fairly important, as histograms are the canonical example in the field of differential privacy, and finding other computations where the access pattern reveals a histogram of the input will allow us to broadly apply our techniques.

With that in mind, we extend our techniques to graph structured data, and the graph-parallel frameworks that support highly parallelized computation. There are several frameworks of this type, including MapReduce, Pregel, GraphLab and others [21, 23, 22]. We describe the framework by Gonzalez et al. [24] called PowerGraph since it combines the best features from both Pregel and GraphLab. PowerGraph is a graph-parallel abstraction that consists of a sparse graph that encodes computation as *vertex-programs*, which run in parallel and interact along edges in the graph. While the implementation of vertex-programs in Pregel and GraphLab differ in how they collect and disseminate information, they share a common structure called the GAS model of graph computation. The GAS model represents three conceptual phases of a vertex-program: Gather, Apply, and Scatter. The computation proceeds in

```
$DumGen_{p,\alpha}$

Input: None.

Computation:
  $d = 2\alpha|V|$
  $dummy_{1:d} \leftarrow \perp$
  for $i = 0 \ldots |V| - 1$
      $j = 2\alpha i$
      $\gamma_i \leftarrow \mathfrak{D}_p$
      $k = \gamma_i + j$
      $dummy_{j:k} = i$

Output: $\langle dummy \rangle$
```

```
$DumGen_{p,\alpha}$

Input: None.

Computation:
  $d = 2\alpha|V|$
  $\mathsf{dummyEdges}_{1:d} \leftarrow \perp$
  for $i = 0 \ldots |V| - 1$
      $j = 2\alpha i$
      $\gamma_i \leftarrow \mathfrak{D}_p$
      $k = \gamma_i + j$
      $\mathsf{dummyEdges}_{j:k}.v = i$

Output: $\langle \mathsf{dummyEdges} \rangle$
```

```
$DumGen_{p,\alpha}$

Input: None.

Computation:
  $d = 2\alpha|V|$
  $\mathsf{dummyEdges}_{1:d} \leftarrow \perp$
  for $i = 0 \ldots |V| - 1$
      $j = 2\alpha i$
      $\gamma_i \leftarrow \mathfrak{D}_p$
      $\delta_i \leftarrow \mathfrak{D}_p$
      $k = \gamma_i + j$
      $\ell = \delta_i + j$
      $\mathsf{dummyEdges}_{j:k}.v = i$
      $\mathsf{dummyEdges}_{j:\ell}.u = i$

Output: $\langle \mathsf{dummyEdges} \rangle$
```

Figure 2: Three variations on the Ideal functionality, $DumGen_{p,\alpha}$. Each is parameterized by α, p. The leftmost functionality is used in the histogram protocol described in Section 3. The middle definition is the one used in our implementation. The right-most adds differential privacy to out-degrees, which is needed in the disjoint collection model (i.e. when hiding the input sizes for all users).

```
$\mathcal{F}_{\mathsf{gas}}$
GAS Model Operations

Inputs: $\langle\mathsf{Vertices}\rangle, \langle\mathsf{Edges}\rangle$
Gather
    for $e(u, v) \in \mathsf{Edges}$
        $v.data \leftarrow \mathsf{copy}(e.data)$
Apply$_f$
    $v.data \leftarrow f(v)$
Scatter
    for $e(u, v) \in \mathsf{Edges}$
        $e.data \leftarrow \mathsf{copy}(u.data)$
Outputs: $\langle\mathsf{Vertices}\rangle, \langle\mathsf{Edges}\rangle$
```

Figure 3: Ideal functionality for a single iteration of the GAS model operations

```
$\pi_{\mathsf{gas}}$
Secure Graph-Parallel Computation with Differentially Private Access Patterns

Input: $\langle\mathsf{Vertices}\rangle$, $\langle\mathsf{realEdges}\rangle$
Computation:
  $\langle\mathsf{dummyEdges}\rangle_{1:d} \leftarrow DumGen_{p,\alpha}$
  $\langle\mathsf{dummyEdges}.flag\rangle \leftarrow 0$
  $\langle\mathsf{realEdges}.flag\rangle \leftarrow 1$
  $\langle\mathsf{Edges}\rangle_{1:(r+d)} \leftarrow \langle\mathsf{realEdges}\rangle || \langle\mathsf{dummyEdges}\rangle$
  $\langle\mathsf{Edges}\rangle \leftarrow \mathcal{F}_{\mathsf{Shuffle}}(\langle\mathsf{Edges}\rangle)$

  Gather
  $\langle\mathsf{Edges}\rangle \leftarrow \mathcal{F}_{\mathsf{Shuffle}}(\langle\mathsf{Edges}\rangle)$
  for $i = 1 \ldots r + d$
    $\mathsf{Open}(\mathsf{Edges}_i.u)$
    $\langle D_u\rangle \leftarrow \mathsf{copy}(\langle\mathsf{Edges}_i.D_u\rangle)$

  Apply
  $\langle\mathsf{Vertices}\rangle \leftarrow \mathcal{F}_{\mathsf{func}}(\langle\mathsf{Vertices}\rangle)$

  Scatter
  $\langle\mathsf{Edges}\rangle \leftarrow \mathcal{F}_{\mathsf{Shuffle}}(\langle\mathsf{Edges}\rangle)$
  for $i = 1 \ldots r + d$
    $\mathsf{Open}(\mathsf{Edges}_i.v)$
    $\langle\mathsf{Edges}_i.D_v\rangle \leftarrow \mathsf{copy}(\langle D_v\rangle)$
Output: $\langle\mathsf{Vertices}\rangle$ $\langle\mathsf{realEdges}\rangle$
```

Figure 4: A protocol for two parties to compute a single iteration of the GAS model operation on secret-shared data. This protocol realizes the ideal functionality described in Figure 3.

iterations, and in each iteration, every node gathers data from their incoming edges, applies some simple computation to the data, and then scatters the result to their outgoing edges. Viewing each node as a CPU (or by assigning multiple nodes to each CPU), the apply step, which constitutes the bulk of the computational work, is easily parallelized. The framework is quite general, and captures computations such as gradient descent, which is used in matrix factorization for recommendation systems, as well PageRank, histograms, and many other computations. Taking matrix factorization as an example, an edge (u, v) indicates that user u reviewed item v, and the data stored on the edge indicates the value of the user's review.

When computing in this manner, the access pattern to memory reveals the edges between nodes. However, because we touch only the left node of every edge during the gather, and only the right node of every edge during the scatter, by adding an oblivious shuffle of the edges between these two steps, we can hide the connection between neighboring nodes. The leakage of the computation is then reduced to two histograms: the in-degrees of all nodes, and the out-degrees of all nodes! We preserve deferential privacy by adding noise to these two histograms, just as we did in the previous section. Details follow below, the formal protocol specification appears in Figure 4, and the ideal functionality for the PowerGraph framework appears in Figure 3.

We denote the data graph by $G = (V, E)$. We denote the data associated with each vertex $v \in V$ and each edge $e \in E$, with $v.data$, and $e.data$, respectively. As in Section 3, our protocol is in a hybrid model where we assume we have access to three ideal functionalities: $DumGen_{p,\alpha}$, $\mathcal{F}_{\mathsf{Shuffle}}$, $\mathcal{F}_{\mathsf{func}}$. As compared to Section 3, here we have dropped an explicit specification of the permutation used in the $\mathcal{F}_{\mathsf{Shuffle}}$. In all instances, we use a random permutation. (Since the dummy flags are now included inside the edge structure, we no longer need to specify that they are shuffled using the same permutation as the data elements.) We note that only the shuffle operations inside Gather and Scatter are repeated in each iteration.

In the protocol, Apply makes a call to an ideal functionality, $\mathcal{F}_{\mathsf{func}}$. This functionality takes secret shares of all vertices, applies the specified function to the data at each vertex, and returns fresh secret shares of the aggregated data. In our protocol, we implement this ideal functionality using generic secure computation. We don't focus on the details here, as they have been described elsewhere (e.g. [20, 19]).

The ideal functionality for $DumGen_{p,\alpha}$ appears in the middle column of Figure 2. The only difference between the functionality described there and the one in the left portion of the figure (which was used in Section 3) is that our "types" are now node identifiers, and they are stored within edge structures. However, the reader should note that only the right node in each edge is assigned a dummy value, while the left nodes all remain $\perp$. This design choice is for efficiency, and comes at the cost of leaking the exact histogram defined by the out-degrees of the graph nodes. For example, when computing gradient descent for matrix factorization, this reveals the number of reviews written by each user, while ensuring that the number of reviews received by each item remains differentially private. In particular, then, the noise hides whether any given user reviewed any specific item. This suffices for achieving security with known input sizes, as defined in Definition 2. This is the protocol that we use in our implementation, but we also include a third variant of $DumGen_{p,\alpha}$ on the right side of the Figure. In that variant, separate noise is added to the left node of each edge as well, which provides security according to Definition 3. We do not implement or analyze the security of this variant. However, it is not hard to see that this doubles the "sensitivity" of the "query", and that ϵ will have to be cut in half in order to provide the same security guarantee. This roughly amounts to doubling the number of dummy edges in the system. The impact this would have on performance depends on the ratio of real edges to dummy edges in the system, which itself depends on the data set and the number of vertices in the graph. See Section 5 for a general sense of how these parameters impact performance.

In some computations, the graph is known to be bipartite, with all edges starting in the left vertex set and ending in the right vertex set (again, recommendation systems are a natural example). In this case, since it is known that all nodes in the left vertex set have in-degree 0, we do not need to add dummy edges containing nodes. This cuts down on the number of dummies required, and we take advantage of this when implementing matrix factorization.

In our implementation of $DumGen_{p,\alpha}$, we instantiate $\mathfrak{D}_p$ with the distribution described in Section 2. Intuitively, as outlined in the work of Dwork et al. [31], we sample this distribution by repeatedly flipping a coin until it comes up heads, and letting the number of coin flips determine the number of dummy items. The bias of the coin is $p \in [0, 1]$, and we assume $p = 1/2^{\ell}$ for some integer ℓ. (The exact value depends on the choice of ϵ, as described in the next subsection.) Each party inputs a random string, and we let the XOR of these strings define the random tape for flipping the biased coins. If the first ℓ bits of the random tape are 1, the first coin is set to heads, and otherwise it set to tails: this is computed with a single ℓ-input AND gate. We iterate through the random tape, ℓ bits at a time, determining the value of each coin, and setting the dummy elements appropriately. Recall that $\mathfrak{D}_p$ is a two-sided distribution, so we also use one bit from the random tape to determine the sign of the sample. To prevent a negative number of dummy items, we add α dummies to the result of $\mathfrak{D}_p$. Therefore, when setting values based on the coin flip, we either add or subtract dummy values from the initial α, based on the sign of the sample drawn from $\mathfrak{D}_p$. Note that the output length is fixed, regardless of this random tape, so after we set the appropriate number of dummy items based on our coin flips, the remaining output values are set to $\perp$.

The cost of this implementation of $DumGen_{p,\alpha}$ is $O(V)$, though this hides a dependence on ϵ and δ: an exact accounting for various values can be found in Section 5. This cost is small relative to the cost of the oblivious shuffle, but we did first consider a much simpler protocol for $DumGen_{p,\alpha}$ that is worth describing. Instead of performing a coin flip inside a secure computation, by choosing a different distribution, we can implement $DumGen_{p,\alpha}$ without any interaction at all! To do this, we have each party choose d random values from $\{1, \ldots, |V|\}$, and view them as additive shares (modulo $|V|$) of each dummy item. Note that this distribution is already one-sided, so we do not need to worry about α, and it already has fixed length output, so we do not need to worry about padding the dummy array with $\perp$ values. Intuitively, this can be viewed as $|V|$ correlated samples from the binomial distribution, where the bias of the coin is $1/|V|$. Unfortunately, the binomial distribution performs far worse than the geometric distribution, and in concrete terms, for the same values of ϵ and δ, this protocol resulted in 250X more dummy items. The savings from avoiding the secure computation of $DumGen_{p,\alpha}$ were easily washed away by the cost of shuffling so many additional items. It is interesting to note that in the "standard" settings where differential privacy is employed, additional noise affects the accuracy of the result, whereas here it costs us in terms of performance.

4.1 Proof of security

We begin by describing the leakage function $\mathcal{L}(\mathsf{Vertices}, \mathsf{Edges})$. For each $v \in V$, we let $\mathsf{out\text{-}deg}(v)$ denote the nodes out-degree, and we use $\mathsf{out\text{-}deg}(V)$ to denote the set of integers, $\{\mathsf{out\text{-}deg}(v)\}_{v \in V}$. We use $\mathsf{in\text{-}deg}()$ analogously. Note that $|V|$ and $|E|$ are both determined by $\mathsf{out\text{-}deg}(V)$, and these values will be used by the simulator as well. Recall that the edges are formatted as $D_e :=< u, v, e, isReal, D_u, D_v >$. We define a database $\mathcal{DB}_L$ by taking every edge $D_e \in \mathsf{Edges}$, and adding $D_e.v$ to $\mathcal{DB}_L$. Intuitively, this is a multi-set over the node identifiers from the input graph, with each node identifier v appearing k times if $\mathsf{in\text{-}deg}(v) = k$. The leakage function is $\mathcal{L}(\mathsf{Vertices}, \mathsf{Edges}) = (\mathcal{F}_{\epsilon,\delta}(\mathcal{DB}_L), \mathsf{out\text{-}deg}(V))$ (where $\mathcal{F}_{\epsilon,\delta}$ is the mechanism defined in Section 2). We note that $\mathsf{out\text{-}deg}(V)$ can both be modeled as auxiliary information about $\mathcal{DB}_L$ – intuitively, it can be viewed as the number of rows that each user contributed to the database – so the proof that $\mathcal{L}$ preserves differential privacy follows from the fact that the mechanism $\mathcal{F}_{\epsilon,\delta}$ is differentially private. It is well known that similar methods of generating 1-sided noise preserve differential privacy, but, for completeness, we prove it below for our modified distribution, which is much simpler to execute in a garbled circuit.

Analyzing our mechanism: We remind the reader that we use the following distribution for sampling noise:

$$\Pr[\gamma_i = \alpha] = \frac{p}{2}$$

$$\forall k \in \mathbb{Z}, k \neq 0 : \Pr[\gamma_i = \alpha + k] = \frac{1}{2}(1 - \frac{p}{2})p(1-p)^{|k|-1}.$$

Consider any two neighboring databases, D_1, D_2, and some fixed $\widehat{\mathcal{D}} \in \mathcal{R}_{\mathcal{F}}$, $\widehat{\mathcal{D}} \neq \emptyset$ for $\mathcal{F}$ as just defined. Let $\widehat{\mathcal{D}}_1 = \mathcal{F}(D_1)$, let $\widehat{\mathcal{D}}_2 = \mathcal{F}(D_2)$, and let i be the value for which $D_1(i) = D_2(i) + 1$. By the definition of $\mathcal{F}$, for $j \neq i$, $\Pr[\widehat{\mathcal{D}}_1(j) = \widehat{\mathcal{D}}(j)] = \Pr[\widehat{\mathcal{D}}_2(j) = \widehat{\mathcal{D}}(j)]$. Furthermore, for $k \neq j, k \neq i$, $b \in \{1, 2\}$,

$\widehat{\mathcal{D}}_b(k)$ and $\widehat{\mathcal{D}}_b(j)$ are sampled independently. Therefore,

$$\frac{\Pr[\widehat{\mathcal{D}}_1 = \widehat{\mathcal{D}}]}{\Pr[\widehat{\mathcal{D}}_2 = \widehat{\mathcal{D}}]} = \frac{\Pr[\widehat{\mathcal{D}}_1(i) = \widehat{\mathcal{D}}(i)]}{\Pr[\widehat{\mathcal{D}}_2(i) = \widehat{\mathcal{D}}(i)]} \leq \frac{1}{(1-p)}$$

(Note that the case $|\widehat{\mathcal{D}}(i)| = |\widehat{\mathcal{D}}_1(i)|$ – i.e. where there is no noise of type i added to the first dataset – $\frac{\Pr[\widehat{\mathcal{D}}_1=\widehat{\mathcal{D}}]}{\Pr[\widehat{\mathcal{D}}_2=\widehat{\mathcal{D}}]} \leq \frac{1}{1-p/2} < \frac{1}{1-p}$.) By choosing $1-p = e^{-\epsilon}$, we achieve the desired bound. Then, for any $T_{\mathsf{g}} \subseteq \mathcal{F}_R \setminus \{\emptyset\}$,

$$\begin{aligned}\Pr[\mathcal{F}(D_1) \in T_{\mathsf{g}}] &= \sum_{D \in T_{\mathsf{g}}} \Pr[\mathcal{F}(D_1) = D] \\ &\leq \sum_{D \in T_{\mathsf{g}}} e^{\epsilon} \Pr[\mathcal{F}(D_2) = D] \\ &= e^{\epsilon} \Pr[\mathcal{F}(D_2) \in T_{\mathsf{g}}]\end{aligned}$$

We now consider the probability that $\mathcal{F}(D) = \emptyset$. Recall, this is exactly the probability that for some $i \in V$, $\gamma_i < 0$, which grows as a negligible function in α. We choose α such that this probability is δ. (We will derive the exact function below, and demonstrate some sample parameters.) Then, for any $T \subseteq \mathcal{F}_R$, letting $T_{\mathsf{g}} = T \setminus \{\emptyset\}$,

$$\begin{aligned}\Pr[\mathcal{F}(D_1) \in T] &= \Pr[\mathcal{F}(D_1) \in T_{\mathsf{g}}] + \Pr[\mathcal{F}(D_1) = \emptyset] \\ &\leq e^{\epsilon} \Pr[\mathcal{F}(D_2) \in T_{\mathsf{g}}] + \delta \\ &\leq e^{\epsilon} \Pr[\mathcal{F}(D_2) \in T] + \delta\end{aligned}$$

Setting the parameters We set $\delta = 2^{-40}$, and show how to calculate α; this allows us to give the expected size of $\widehat{\mathcal{D}}$ as a function of ϵ and δ. We first fix some $i \in V$ and calculate $\Pr[\gamma_i < 0]$, and then we take a union bound over $|V|$.

$$\begin{aligned}\Pr[\gamma_i < 0] &= \sum_{k=\alpha+1}^{\infty} \frac{1}{2}(1-\frac{p}{2})p(1-p)^{k-1} \\ &= \frac{p}{2}(1-\frac{p}{2}) \sum_{k=0}^{\infty} (1-p)^{\alpha}(1-p)^{k} \\ &= \frac{p}{2}(1-\frac{p}{2})(1-p)^{\alpha} \frac{1}{1-(1-p)} \\ &= \frac{1}{2}(1-\frac{p}{2})(1-p)^{\alpha}\end{aligned}$$

After taking a union bound over $|V|$, we have $\Pr[\mathcal{F}(D) = \emptyset] \leq 2^{-40}$ when $\alpha > \frac{-40-\log(\frac{1}{2}-\frac{p}{4})-\log(|V|)}{\log(1-p)}$. Recall that $(1-p) = e^{-\epsilon}$. So, as an example, setting $\epsilon = .3$ and $|V| = 2^{12}$, we have $\alpha = 118$, and $\mathbb{E}(|\mathcal{F}(D)|) = 118|V| + |D|$.

Theorem 1 *The protocol π_{gas} defined in Figure 4 securely computes $\mathcal{F}_{\mathsf{gas}}$ with $\mathcal{L}$ leakage in the $(\mathcal{F}_{\mathsf{func}}, \mathcal{F}_{\mathsf{Shuffle}}, DumGen_{p,\alpha})$-hybrid model.*

Proof: (sketch.) We construct a simulator for a semi-honest P_1. For all three ideal functionalities, the output is simply and XOR secret sharing of some computed value. The output of all calls to these functionalities can be simply simulated using random binary strings of the appropriate length. Let $\mathsf{simEdges}_1$ denote the random string used to simulate the output of $\mathcal{F}_{\mathsf{Shuffle}}$ the first time the functionality is called, and let $\mathsf{simEdges}_2$ denote the random string used to simulate the output on the second call. Let $\mathsf{simEdges}_1.u$ denote the restriction of $\mathsf{simEdges}_1$ to the bits that make up the sharings $\mathsf{Edges}.u$, and let $\mathsf{simEdges}_2.v$ be defined similarly.

There are only two remaining messages to simulate: $\mathsf{Open}(\mathsf{Edges}.u)$, and $\mathsf{Open}(\mathsf{Edges}.v)$. Recall that there are $|E|+2\alpha|V|$ edges in the Edges: the original $|E|$ real edges, and the $2\alpha|V|$ dummy edges generated in $DumGen_{p,\alpha}$. To simulate the message sent when opening $\mathsf{Edges}.u$, the simulator uses the values $|V|$ and $\mathsf{out\text{-}deg}(V)$ to create a bit string representing a random shuffling of the following array of size $|E| + 2\alpha|V|$. For each $u \in V$, the array contains the identifier of u exactly $\mathsf{out\text{-}deg}(u)$ times. This accounts for $|E| = \sum_u \mathsf{out\text{-}deg}(u)$ positions of the array; the remaining $2\alpha|V|$ positions are set to $\perp$, consistent with the left nodes output by $DumGen_{p,\alpha}$. Letting r denote the resulting bit-string, the simulator sends $r \oplus \mathsf{simEdges}_1.u$ to the adversary.

To simulate $\mathsf{simEdges}_2.v$, the simulator creates another bit-string representing a random shuffling of the following array, again of size $|E| + 2\alpha|V|$. Letting $\widehat{\mathcal{D}} = \mathcal{F}(\mathcal{DB}_L)$ denote the first element output by the leakage $\mathcal{L}$, the simulator.adds the node identifiers in $\widehat{\mathcal{D}}$ to the array. In the remaining $|E|+2\alpha|V|-|\widehat{\mathcal{D}}|$ positions of the array, he adds $\perp$. Letting r denote the resulting bit-string, the simulator sends $r \oplus \mathsf{simEdges}_2.v$ to the adversary.

■

Hiding the out-degree of each node. We don't formally prove that using the third variant of $DumGen_{p,\alpha}$ suffices for achieving security as described in Definition 3. We instead provide a brief intuition for the argument. For a graph $G = (E, V)$, it is helpful to think of the edge set as defining two databases of elements over V: for each (directed) edge (u, v), we will view u as an element in database E_L and v as an element in database E_R. Because the oblivious shuffle hides the edges between these two databases, the access pattern can be fully simulated from two noisy histograms (one for each database). Because differential privacy composes, the added noisy information has the affect of cutting ϵ in half.

Hiding a user's full edge set. The leakage function described above provide edge privacy to each contributing party. That is, we have defined two databases to be neighboring when they differ in a single edge. We could also define two neighboring databases as differing in a single node. This would require more noise: if the maximum degree of each node is d, it would have the affect of scaling ϵ by d. In our experiments, we have included some smaller values of ϵ to help the reader evaluate how this additional noise would impact performance.

5 Implementation and Evaluation

In this section, we describe and evaluate the implementation of our proposed framework called OblivGraph.

We implemented OblivGraph using the FlexSC multi-party computation framework, which executes Yao's Garbled Circuits protocol with a Java-based garbled circuit implementation. We measured the performance of our framework on a set of micro-benchmarks in order to evaluate our design. These micro-benchmarks consist of histogram, PageRank and matrix factorization problems which are commonly used for evaluating highly-parallelizable frameworks.

5.1 Implementation

Using the OblivGraph framework, the histogram and matrix factorization problems can be represented as directed bipartite graphs, and PageRank as a directed non-bipartite graph. When we are computing on bipartite graphs, if we consider the Definition 2.2 where we aim to hide the in-degree of the nodes (nodes on the left have in-degree 0), the growth rate of dummy edges is linear with the number of nodes on the right and it is independent of the real edges or users. However, considering the stronger Definition 2.3, the growth rate of dummy edges is linear with max(users, items).

Histogram: In histogram, left vertices represent data elements, right vertices are the counters for each type of data element, and existence of an edge indicates that the data element on the left has the type on the right.

Matrix Factorization: In matrix factorization, left vertices represent the users, right vertices are items (e.g. movies in movie recommendation systems), the edges show if a user ranked that item, and the weight of the edge represents the rating value.

PageRank: In PageRank, each vertex corresponds to a webpage and each edge is the link between two webpages. The vertex data comprises of two real values, one for the PageRank of the vertex and the other for the number of its outgoing edges. Edge data is a real value corresponding to the weighted contribution of PageRank of the source vertex to the PageRank of the sink vertex.

Vertex and Edge representation: In all scenarios, vertices are identified using 16-bit integers and 1 bit is used to indicate if the edge is real or dummy. For Histograms, besides 16 bits for the vertex id of data elements, we use 20 bits to represent the counter values. In PageRank, we represent the PageRank value using a 40-bit fixed-point representation, with 20-bit for the fractional part. In our matrix factorization experiments, we used synthetic data with variable number of users, variable number of items, a dimension of 10 for the user and item profiles, each with 20 bits for the fractional part of the 40-bit fixed-point representation. We chose these values to be consistent with GraphSC representation.

System setting: We run each benchmark on a pair of processors, one as garbler, and the other as evaluator, on a machine with 2.30GHz 24-processor

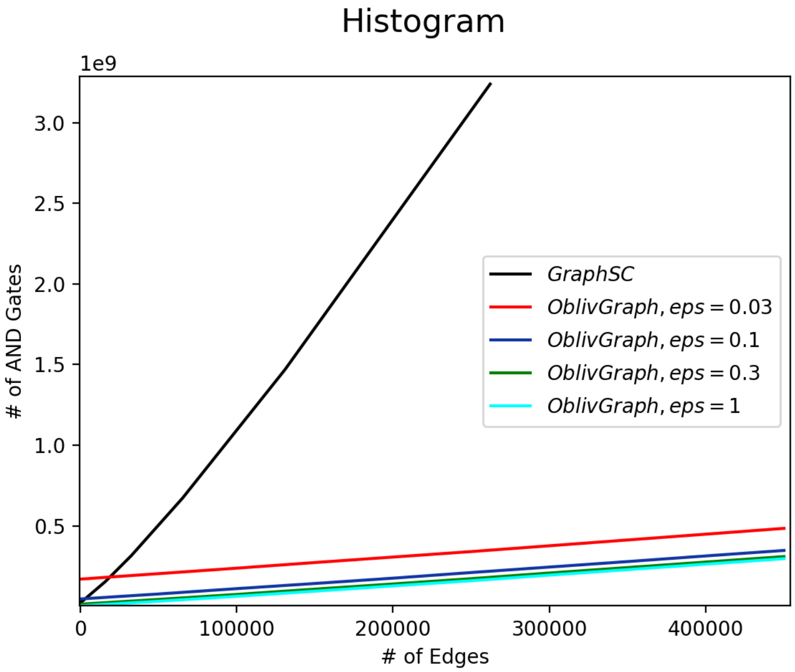

Figure 5: Histogram with 4000 users and 128 types and varying ϵ

and 60GB RAM.

5.2 Evaluation

We used circuit complexity as the main metric to study the performance of our system. We report the total number of AND gates generated in the garbled system. This metric helps us to have a more fair comparison with other frameworks, since it is independent of the hardware configuration and of the chosen secure computation implementation. We run all the benchmarks with the same set of parameters that have been used in the GraphSC framework. In our histogram and matrix factorization experiments, we run the experiments for 4000 users and 128 items. The number of nodes in our PageRank experiment is set to be 2000.

Histogram: To assess the performance of OblivGraph, first we show the results for the histogram example that we used to explain our main construction. Figure 5 demonstrates the number of AND gates for histogram in both the GraphSC and OblivGraph frameworks. In Histogram, with 2000 data elements and 128 data types, we always do better than GraphSC when $\epsilon >= 0.3$. When $\epsilon = 0.1$, we start outperforming GraphSC when there are at least 3400 edges.

Matrix Factorization: We run the same set of experiments for the matrix factorization problem and provide the results in Figure 7. We consider a scenario with 4000 users and a movie set of size 128 movies; we use the (batch) gradient decent method for generating the recommendation model, as in [19, 20]. In MF, with 2000 users, 128 items, and $\epsilon = 0.3$, we outperform GraphSC once there are at least 15000 edges. When $\epsilon = 0.1$, we start outperforming them on 54000 edges. We always do better than GraphSC when the $\epsilon = 1$ or higher.

PageRank: Figure 6 provides the result of running PageRank in our framework with 2000 nodes and different values of ϵ. With $\epsilon = 0.3$, we outperform GraphSC when the number of edges are about 400000, and with $\epsilon = 1$ we outperform them on just 130000 edges. In both cases, the graph is quite sparse, compared to a complete graph of 2 million edges. Note, though, that our comparison is slightly less favorable for this computation. Recall, the number of dummy edges grow with the number of nodes in the graph, and, when hiding only in-degree in a bipartite graph, this amounts to growing only with the number of nodes on the right. In contrast, the runtime of GraphSC grows equivalently with any increase in users, items, or edges, because their protocol hides any distinction between these data types. We therefore compare best with them when there are more users than items. When looking at a non-bipartite graph, such as PageRank, our protocol grows with any increase in the size of the singular set of nodes, just as theirs does. If we increase the number of items in matrix factorization to 2000, or decrease the number of nodes in PageRank to 128, the comparison to GraphSC in the resulting experiments would look similar. We let the reader extrapolate, and avoid the redundancy of adding such experiments.

DumGen Procedure: Figure 8 shows the number of AND gates in the DumGen procedure for different algorithms and varying values of ϵ. Due to the nature of the DumGen procedure, the number of items (or nodes in the case of PageRank) affects the number of dummy edges. Therefore the number of AND gates in PageRank is higher than in histogram and matrix factorization, and the number of AND gates for histogram and matrix factorization are the same due to having the same number of items. By relaxing the privacy notion and increasing the value of ϵ (recall the value of δ is always fixed), the number of required dummy edges will decrease, and consequently the size of the DumGen procedure will shrink. More specifically, increasing ϵ increases the value of p in our geometric distribution, which hastens the halting probability and creates fewer dummy edges on average. We do not include the cost of DumGen in our comparison to GraphSC because it is a one time overhead and we want to capture the cost of one iteration, and note that they have a much more expensive sort that we do not include.

Optimization using Compaction: It is important to note that the measured circuit sizes in our OblivGraph experiments correspond to the worst-case scenario in which the number of dummy edges are equal to $d = 2\alpha|V|$, which is the maximum number of dummies per type. Consequently the time for OblivShuffle is its maximum value. However, looking at the geothermic distribution used in the DumGen procedure, the expected number of dummy edges is $\alpha|V|$, so half of

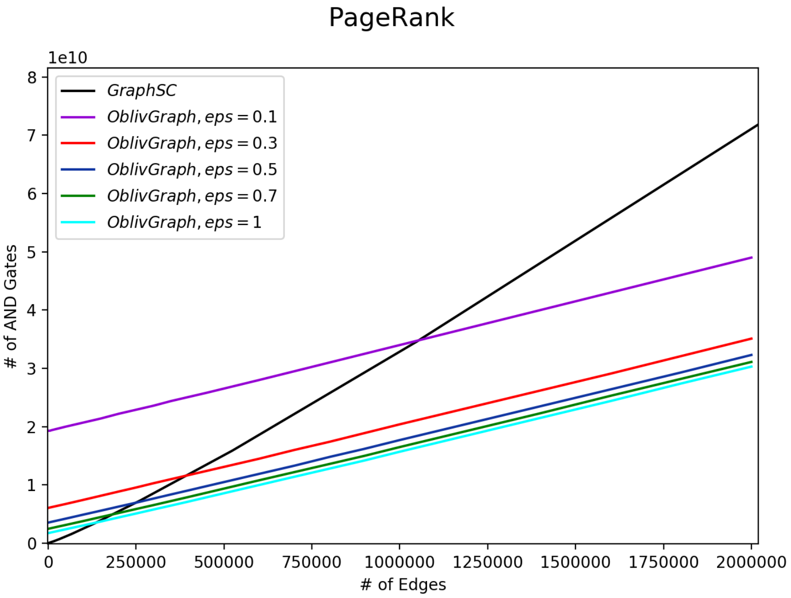

Figure 6: PageRank with 2000 nodes and varying ϵ

the dummy items are unnecessary. Removing these extra dummy items during DumGen is non-trivial, because, while it is safe to reveal the total number of dummy items in the system, revealing the number of dummy items of each type would violate differential privacy. After the first iteration of the computation, once the dummy items are shuffled in with the real items, an extra flag marking the excessive dummy items can be used to safely remove them from the system; this optimization can significantly reduce the shuffling time (roughly by half) in the following iterations. However, our graphs are showing only the first iteration of the algorithm and they do not reflect this simple optimization.

Oblivious Shuffle: We use an Oblivious Shuffle in our OblivGraph framework which has a factor of $\log(n)$ less overhead than the Bitonic sort used in GraphSC. We designed the Oblivious Shuffle operation based on the Waksman network [32]. The cost of shuffling is approximately $BW(n)$ using a Waksman network, where $W(n) = n \log n - n + 1$ is the number of oblivious swaps required to permute n input elements, and B indicates the size of the elements being shuffled. In the original Waksman switching network, the size of the input, n, is assumed to be a power of two. However, in order to have an Oblivious Shuffle for arbitrary sized input, we must use an improved version of the Waksman network proposed in [33] which is called AS-Waksman (Arbitrary-Sized Waksman). The number of necessary swapper gates in AS-Waksman can be calculated using the following formula:

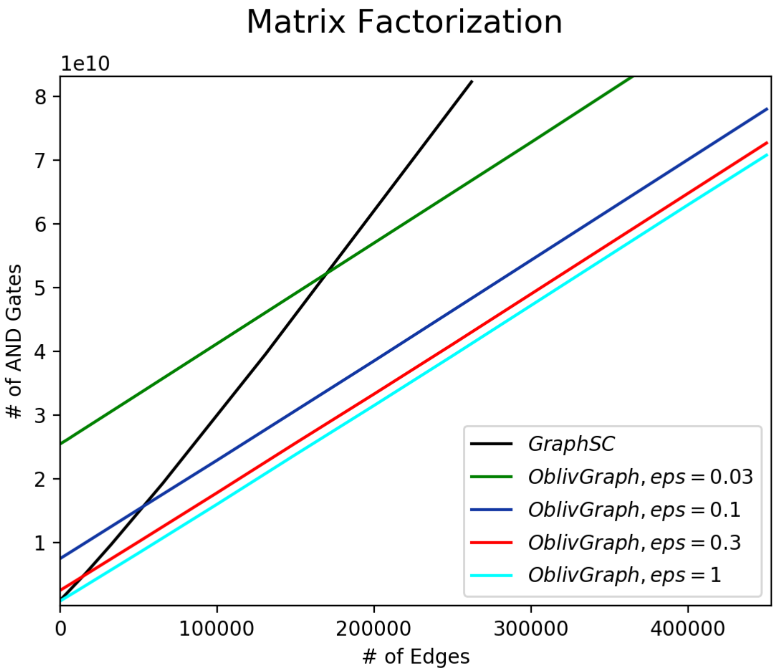

Figure 7: Matrix factorization with 4000 users and 128 movies and varying ϵ

$$W(n) = W\left(\left\lceil \frac{n}{2} \right\rceil\right) + W\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + n - 1 = \sum_{i=1}^{n} \left\lceil \log(i) \right\rceil \tag{3}$$

In our current set of experiments, we have only implemented the original version of the Waksman network and have not implemented AS-Waksman. However, we used the aforementioned formula for AS-Waksman to calculate the number of necessary swapper gates in our Oblivious Shuffle when using arbitrary sized input, and we report these projected values in the graphs. For example in Figure 9 the green line shows the values that we obtained from the Waksman implementation where the input size must be power of two, and the red dotted line represents the projection values we computed with the AS-Waksman formula for any arbitrary sized inputs. Since all of the operations in ObliveGraph, including DumGen, Gather, Apply and Scatter, can work with arbitrary sized inputs, and the only limitation of our current implementation is imposed by using the conventional Waksman switching network, in all graphs we used the estimated circuit size of the OblivShuffle for non-power of two inputs and the exact circuit size of the OblivShuffle for power of two inputs. (To avoid confusion, we did not show the step function in the rest of the graphs.)

In order to understand how expensive the DumGen and ObliveShuffle proce-

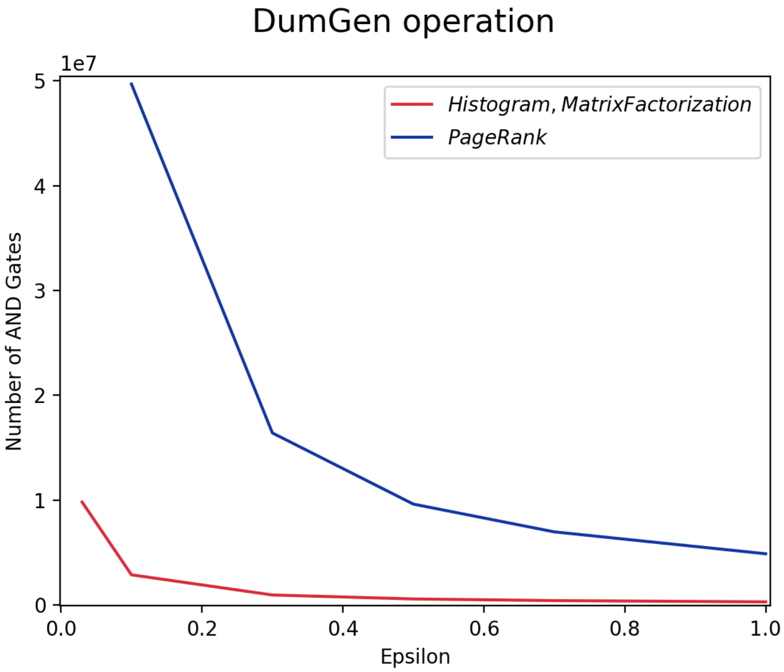

Figure 8: DumGen for different algorithms with different values of ϵ. Histogram and Matrix Factorization with 4000 users and 128 types and PageRank with 2000 number of nodes.

dures are, as compared to other GAS model operations, we show the number of AND gates for each of these procedures in Table 1. A single (and only) iteration in the Histogram computation includes one OblivShuffle. PageRank and Matrix Factorization have two OblivShuffle in a single iteration. The results shown in this table are for Histogram and Matrix Factorization with 4000 users and 128 types and PageRank with 2000 number of nodes. In all experiments, we used $\epsilon = 0.3$, and the number of real edges is 250000. The last column estimates the run time of the framework in one single iteration, assuming, conservatively, that 5 million gates can be processed per second. We demonstrate the result for GraphSC framework with similar parameters in Table 2. In all of these experiments we eliminate the effect of parallelization by running the computations on a single machine. As demonstrated in Table 3 and discussed below, the only overhead in parallelizing our protocol lies in the communication cost; roughly, the estimated times reported in Table 1 can be reduced by a factor of P by using P processors.

Effect of Parallelization: Table 3 shows the effect of parallelization in our framework as compared to GraphSC. Adding more processors in GraphSC in-

Computation	DumGen	Shuffle	GAS Operations in a Single Iteration	Total AND Gates in a Single Iteration	Estimated
Histogram	$5.02E+05$	$1.61E+08$	$1.11E+07$	$1.72E+08$	
PageRank	$1.64E+07$	$2.97E+09$	$3.62E+09$	$9.56E+09$	1
Matrix Factorization	$5.02E+05$	$4.49E+09$	$3.22E+10$	$4.11E+10$	8

Table 1: Cost of DumGen and ObliveShuffle versus GAS operation for a single iteration in OblivGraph

Computation	Total AND Gates in a Single Iteration	Estimated Run Time (s) in a Single Iteration
Histogram	$3.24E+09$	647
PageRank	$7.34E+09$	1468
Matrix Factorization	$8.23E+10$	16463

Table 2: Cost of all operations in a single iteration in GraphSC

creases the total number of AND Gates by some small amount. However, the size of the circuit generated in OblivGraph framework is constant in the number of processors: parallelization does not affect the number of AND gates in the OblivGraph GAS operations, DumGen, or the OblivShuffle procedures. It does, of course, increase the total communication in the network, but, as already demonstrated in GraphSC, this has small impact on the value of parallelizing (and is even smaller in our own protocol).

Processors	GraphSC		OblivGraph	
	$\|E\| = 8192$	24576	$\|E\| = 8192$	24576
1	$4.047E+09$	$1.035E+10$	$2.018E+09$	$4.480E+09$
2	$4.055E+09$	$1.039E+10$	$2.018E+09$	$4.480E+09$
4	$4.070E+09$	$1.046E+10$	$2.018E+09$	$4.480E+09$
8	$4.092E+09$	$1.057E+10$	$2.018E+09$	$4.480E+09$

Table 3: Cost of Parallelization on OblivGraph vs. GraphSC in computing Matrix Factorization

6 Conclusion and Open Problems

We have established a new tradeoff between privacy and efficiency in secure computation by defining a new security model in which the adversary is provided some leakage that is proven to preserve differential privacy. We show that this leakage allows us to construct a more efficient protocol for a broad class of computations: those that can be computed in graph-parallel frameworks such

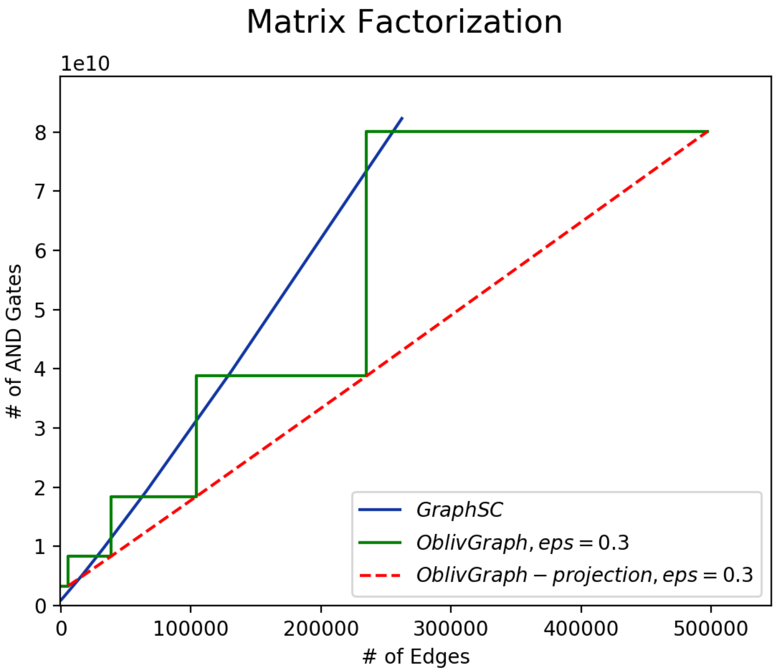

Figure 9: The effect of using Waksman network vs. AS-Waksman network in OblivShuffle procedure in Matrix factorization with 4000 users, 128 movies and $\epsilon = 0.3$

as MapReduce. We have evaluated the impact of our relaxation by comparing the performance of our protocol with the best prior implementation of secure computation for graph-parallel frameworks.

Our work demonstrates that differentially private leakage is useful, in that it provides opportunity for more efficient protocols. The protocol we present has broad applicability, but we leave open the very interesting question of determining, more precisely, for which class of computations this leakage might be help. Graph-parallel algorithms have the property that the access pattern to memory can be easily reduced to revealing only a histogram of the memory that is accessed, and histograms are the canonical example in the differential privacy literature. Looking at other algorithms will likely introduce very interesting leakage functions that are new to the differential privacy literature, and security might not naturally follow from known mechanisms in that space. A wonderful example is the stable matching problem, which is another large-scale computation that has been the focus of some research in secure computation.

References

[1] S. Zahur and D. Evans, "Obliv-c: A language for extensible data-oblivious computation," Cryptology ePrint Archive, Report 2015/1153, 2015, http://eprint.iacr.org/2015/1153.

[2] B. Kreuter, "Secure multiparty computation at google," https://www.youtube.com/watch?v=ee7oRsDnNNc, 2017, real World Crypto.

[3] Ú. Erlingsson, V. Pihur, and A. Korolova, "RAPPOR: Randomized aggregatable privacy-preserving ordinal response," in *ACM CCS 14*, G.-J. Ahn, M. Yung, and N. Li, Eds. ACM Press, Nov. 2014, pp. 1054–1067.

[4] R. Wyden, "Letter to commission on evidence-based policymaking," https://www.wyden.senate.gov/download/?id=B10146F5-EDEB-4A2C-AD5E-812B363EE0DC&download=1, 2017, u.S. Senate.

[5] S. D. Gordon, J. Katz, V. Kolesnikov, F. Krell, T. Malkin, M. Raykova, and Y. Vahlis, "Secure two-party computation in sublinear (amortized) time," in *ACM CCS 12*, T. Yu, G. Danezis, and V. D. Gligor, Eds. ACM Press, Oct. 2012, pp. 513–524.

[6] X. S. Wang, Y. Huang, T.-H. H. Chan, A. Shelat, and E. Shi, "SCORAM: Oblivious RAM for secure computation," in *ACM CCS 14*, G.-J. Ahn, M. Yung, and N. Li, Eds. ACM Press, Nov. 2014, pp. 191–202.

[7] C. Liu, X. S. Wang, K. Nayak, Y. Huang, and E. Shi, "ObliVM: A programming framework for secure computation," in *2015 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, May 2015, pp. 359–376.

[8] S. Zahur, X. S. Wang, M. Raykova, A. Gascón, J. Doerner, D. Evans, and J. Katz, "Revisiting square-root ORAM: Efficient random access in multiparty computation," in *2016 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, May 2016, pp. 218–234.

[9] S. Zahur and D. Evans, "Obliv-C: A language for extensible data-oblivious computation," Cryptology ePrint Archive, Report 2015/1153, 2015, http://eprint.iacr.org/2015/1153.

[10] D. Cash, S. Jarecki, C. S. Jutla, H. Krawczyk, M.-C. Rosu, and M. Steiner, "Highly-scalable searchable symmetric encryption with support for Boolean queries," in *CRYPTO 2013, Part I*, ser. LNCS, R. Canetti and J. A. Garay, Eds., vol. 8042. Springer, Heidelberg, Aug. 2013, pp. 353–373.

[11] D. Cash, J. Jaeger, S. Jarecki, C. S. Jutla, H. Krawczyk, M.-C. Rosu, and M. Steiner, "Dynamic searchable encryption in very-large databases: Data structures and implementation," in *NDSS 2014*. The Internet Society, Feb. 2014.

[12] V. Pappas, F. Krell, B. Vo, V. Kolesnikov, T. Malkin, S. G. Choi, W. George, A. D. Keromytis, and S. Bellovin, “Blind seer: A scalable private DBMS,” in *2014 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, May 2014, pp. 359–374.

[13] S. Kamara and T. Moataz, “Boolean searchable symmetric encryption with worst-case sub-linear complexity,” in *Advances in Cryptology - EUROCRYPT 2017 - 36th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Paris, France, April 30 - May 4, 2017, Proceedings, Part III*, ser. Lecture Notes in Computer Science, J. Coron and J. B. Nielsen, Eds., vol. 10212, 2017, pp. 94–124. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-56617-7_4

[14] M. S. Islam, M. Kuzu, and M. Kantarcioglu, “Access pattern disclosure on searchable encryption: Ramification, attack and mitigation,” in *NDSS 2012*. The Internet Society, Feb. 2012.

[15] M. Naveed, S. Kamara, and C. V. Wright, “Inference attacks on property-preserving encrypted databases,” in *ACM CCS 15*, I. Ray, N. Li, and C. Kruegel:, Eds. ACM Press, Oct. 2015, pp. 644–655.

[16] D. Cash, P. Grubbs, J. Perry, and T. Ristenpart, “Leakage-abuse attacks against searchable encryption,” in *ACM CCS 15*, I. Ray, N. Li, and C. Kruegel:, Eds. ACM Press, Oct. 2015, pp. 668–679.

[17] G. Kellaris, G. Kollios, K. Nissim, and A. O’Neill, “Generic attacks on secure outsourced databases,” in *ACM CCS 16*, E. R. Weippl, S. Katzenbeisser, C. Kruegel, A. C. Myers, and S. Halevi, Eds. ACM Press, Oct. 2016, pp. 1329–1340.

[18] F. B. Durak, T. M. DuBuisson, and D. Cash, “What else is revealed by order-revealing encryption?” in *ACM CCS 16*, E. R. Weippl, S. Katzenbeisser, C. Kruegel, A. C. Myers, and S. Halevi, Eds. ACM Press, Oct. 2016, pp. 1155–1166.

[19] V. Nikolaenko, S. Ioannidis, U. Weinsberg, M. Joye, N. Taft, and D. Boneh, “Privacy-preserving matrix factorization,” in *ACM CCS 13*, A.-R. Sadeghi, V. D. Gligor, and M. Yung, Eds. ACM Press, Nov. 2013, pp. 801–812.

[20] K. Nayak, X. S. Wang, S. Ioannidis, U. Weinsberg, N. Taft, and E. Shi, “GraphSC: Parallel secure computation made easy,” in *2015 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, May 2015, pp. 377–394.

[21] J. Dean and S. Ghemawat, “Mapreduce: Simplified data processing on large clusters,” in *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, ser. OSDI’04. Berkeley, CA, USA: USENIX Association, 2004, pp. 10–10. [Online]. Available: http://dl.acm.org/citation.cfm?id=1251254.1251264

[22] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: A system for large-scale graph processing," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '10. New York, NY, USA: ACM, 2010, pp. 135–146. [Online]. Available: http://doi.acm.org/10.1145/1807167.1807184

[23] Y. Low, J. E. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein, "Graphlab: A new framework for parallel machine learning," *CoRR*, vol. abs/1408.2041, 2014. [Online]. Available: http://arxiv.org/abs/1408.2041

[24] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "Powergraph: Distributed graph-parallel computation on natural graphs," in *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*. Hollywood, CA: USENIX, 2012, pp. 17–30. [Online]. Available: https://www.usenix.org/conference/osdi12/technical-sessions/presentation/gonzalez

[25] A. Papadimitriou, A. Narayan, and A. Haeberlen, "Dstress: Efficient differentially private computations on distributed data," in *Proceedings of the Twelfth European Conference on Computer Systems, EuroSys 2017, Belgrade, Serbia, April 23-26, 2017*, G. Alonso, R. Bianchini, and M. Vukolic, Eds. ACM, 2017, pp. 560–574. [Online]. Available: http://doi.acm.org/10.1145/3064176.3064218

[26] G. Kellaris, G. Kollios, K. Nissim, and A. O'Neill, "Accessing data while preserving privacy," https://www.youtube.com/watch?v=u9LIU4Frce8, 2017, communication with the authors.

[27] S. Wagh, P. Cuff, and P. Mittal, "Root ORAM: A tunable differentially private oblivious RAM," *CoRR*, vol. abs/1601.03378, 2016. [Online]. Available: http://arxiv.org/abs/1601.03378

[28] E. Stefanov, M. van Dijk, E. Shi, C. W. Fletcher, L. Ren, X. Yu, and S. Devadas, "Path ORAM: an extremely simple oblivious RAM protocol," in *ACM CCS 13*, A.-R. Sadeghi, V. D. Gligor, and M. Yung, Eds. ACM Press, Nov. 2013, pp. 299–310.

[29] C. Dwork and A. Roth, "The algorithmic foundations of differential privacy," *Foundations and Trends in Theoretical Computer Science*, vol. 9, no. 3-4, pp. 211–407, 2014. [Online]. Available: http://dx.doi.org/10.1561/0400000042

[30] R. Canetti, "Security and composition of multiparty cryptographic protocols," *Journal of Cryptology*, vol. 13, no. 1, pp. 143–202, 2000.

[31] C. Dwork, K. Kenthapadi, F. McSherry, I. Mironov, and M. Naor, "Our data, ourselves: Privacy via distributed noise generation," in *EUROCRYPT 2006*, ser. LNCS, S. Vaudenay, Ed., vol. 4004. Springer, Heidelberg, May / Jun. 2006, pp. 486–503.

[32] A. Waksman, "A permutation network," *Journal of the ACM (JACM)*, vol. 15, no. 1, pp. 159–163, 1968.

[33] B. Beauquier and É. Darrot, "On arbitrary size waksman networks and their vulnerability," *Parallel Processing Letters*, vol. 12, no. 03n04, pp. 287–296, 2002.