Observing and Preventing Leakage in MapReduce*

Olga Ohrimenko
Microsoft Research
oohrim@microsoft.com

Manuel Costa
Microsoft Research
manuelc@microsoft.com

Cédric Fournet
Microsoft Research
fournet@microsoft.com

Christos Gkantsidis
Microsoft Research
christos.gkantsidis@microsoft.com

Markulf Kohlweiss
Microsoft Research
markulf@microsoft.com

Divya Sharma†
Carnegie Mellon University
divyasharma@cmu.edu

ABSTRACT

The use of public cloud infrastructure for storing and processing large datasets raises new security concerns. Current solutions propose encrypting all data, and accessing it in plaintext only within secure hardware. Nonetheless, the distributed processing of large amounts of data still involves intensive encrypted communications between different processing and network storage units, and those communications patterns may leak sensitive information.

We consider secure implementation of MapReduce jobs, and analyze their intermediate traffic between mappers and reducers. Using datasets that include personal and geographical data, we show how an adversary that observes the runs of typical jobs can infer precise information about their input. We give a new definition of data privacy for MapReduce, and describe two provably-secure, practical solutions. We implement our solutions on top of VC3, a secure implementation of Hadoop, and evaluate their performance.

Categories and Subject Descriptors

K.6.5 [**Management of Computing and Information Systems**]: Security and Protection.

Keywords

Map-reduce; traffic analysis; oblivious shuffle; oblivious load balancing.

1. INTRODUCTION

The use of shared cloud infrastructure for storing and processing large structured datasets has gained widespread prominence. In particular, the MapReduce framework is routinely used to outsource such tasks in a simple, scalable, and cost-effective manner. As can be expected, reliance on a cloud provider for processing sensitive data entails new integrity and privacy risks.

Several recent works explore different trade-offs between performance, security, and (partial) trust in the cloud. Most proposals involve protecting data at rest—using some form of authenticated encryption—and protecting data in use with either advanced cryptography or secure hardware. Although homomorphic encryption [12] may address our privacy concerns, it remains impractical for general processing of large data, in particular when they involve complex, dynamic intermediate data. Conversely, limited trust assumptions on the cloud infrastructure may lead to efficient solutions, but their actual security guarantees are less clear.

As a concrete example, VC3 [26] recently showed that, by relying on the new Intel SGX infrastructure [19] to protect local mapper and reducer processing, one can adapt the popular Hadoop framework [2] and achieve strong integrity and confidentiality for large MapReduce tasks with a small performance overhead. All data is systematically AES-GCM-encrypted, except when processed within hardware-protected, remotely-attested enclaves that include just the code for mapping and reducing data, whereas the rest of the Hadoop distributed infrastructure need not be trusted. They report an average 4% performance overhead for typical MapReduce jobs. Trusting Intel's CPUs may be adequate for many commercial applications, and yields a much smaller TCB than the whole cloud infrastructure. Similar practical solutions may rely instead, for instance, on a hypervisor and simple virtual machines dedicated to the MapReduce job.

Even if we assume perfect encryption for all data and perfect isolation for all local processing, mappers and reducers still need to access shared resources (the memory, the data store, the network) thereby opening as many side channels. Their access patterns to memory, storage, and network—such as for instance the data volume of map and reduce jobs—are visible to the cloud provider and, to a lesser extents, to its other tenants. Revealing this information may be justified by the practical performance one gains in return. However, there are circumstances where observing the encrypted traffic of MapReduce jobs on a sensitive dataset reveals much more information than may be expected.

A first, important insight is that observing access to intermediate data structures is more informative than just observing inputs and outputs. In the case of MapReduce, for instance, observing and correlating intermediate key-value pairs exchanged between every mapper and every reducer for a series of typical jobs, each using a different field of the input records as key for mapping—say the age, the place of birth, and the place of work—enables us to label each input record with precise values for *all* these fields. What we learn from such a 'job composition' attack is much more detailed than what we would learn even by accessing the job results in plaintext. This may come as a surprise for data owners, who reason about which MapReduce job to authorize, and which results to declassify, but usually not about leakage in their distributed execution.

*This is an extended version of the work to appear in the proceedings of the 22nd ACM Conference on Computer and Communications Security (CCS 2015).

†Work done at Microsoft Research.

To support our claim, we demonstrate information leaked from two sample datasets (900MB and 24GB, respectively) that include personal and geographical attributes. We assume an honest-but-curious adversary that observes the volume of encrypted communications between long-term storage, mappers, and reducers. (Other lower-level side channels may be available, such as precise timings, page faults, cache misses, etc, but they seem harder to exploit in general, and may not reveal much more information on the input datasets.) Our attacks suggest that, even with the use of encryption and secure hardware, stronger methods are required to avoid leakage through traffic analysis.

To remedy this problem, we propose a new definition of data privacy for MapReduce—essentially, that observable I/O should look independent of the input dataset—and we describe two practical solutions that meet this definition, and thus prevent our attacks.

As a trivial solution, we may pad all accesses and communications to their potential maximal lengths. Similarly, we may apply generic oblivious RAM techniques [13] and oblivious sorting on top of a MapReduce implementation. However, such solutions would incur a polylogarithmic overhead, and they would preclude any speedups enabled by the parallel nature of MapReduce. We further discuss related baseline solutions in §9.

Intuitively, many existing mechanisms already in place in MapReduce frameworks to achieve good performance should also help us for privacy. Mappers and reducers often use large I/O buffers, making it harder to track individual records within large, encrypted batches of data. Similarly, for jobs with adequate load balancing, one would expect mappers and reducers to exchange roughly the same amount of data with one another, thereby limiting that side channel. Our solutions take advantage of these mechanisms to provide strong, provable security, while respecting the simple, uniform, parallel data flow in MapReduce jobs.

In summary, we contribute:

1. an empirical analysis of 'MapReduce leakage' on two sample datasets—a sample of the 1990 US Census and a log of taxi rides in New York—showing that one can reliably infer (or even extract) precise information about these datasets by simply observing the volume of encrypted communications between the mappers and reducers that perform a few typical jobs on their data;

2. a model of information leakage in MapReduce jobs, against an adversary that observes all encrypted intermediate traffic between Map and Reduce nodes;

3. two practical solutions that provably limit leakage to essentially the total volume of their I/O, relying on shuffling, sampling, and a carefully-chosen amount of padding, with different performance trade-offs.

4. an implementation of these solutions, embedded as auxiliary MapReduce jobs for Hadoop and VC3, and their performance evaluation on our sample datasets. Our results suggest that information security can be achieved for typical MapReduce jobs with a reasonable overhead (7% on average for our most secure solution, depending on the distribution of intermediate key-value pairs) and in some cases can outperform the baseline solution due to internal grouping of key-value pairs.

Although we focus on MapReduce, and sometimes on the details of the VC3 secure implementation of Hadoop, our results may be of interest for a larger class of data-intensive applications, such as SQL and semi-structured databases.

2. PRELIMINARIES

Notations. We use $A = \langle a_1, a_2, \ldots \rangle$ to denote a list with records a_1, a_2 etc, and $A \| B$ to denote list concatenation.

If record a_i is a key-value pair then $a_i.\mathsf{key}$ and $a_i.\mathsf{val}$ denote the key and value of the pair, respectively. We let $A_{|k}$ denote the pairs of A with the same key, k, i.e., $A_{|k} = \langle a_i \mid a_i.\mathsf{key} = k \rangle$. For a set K of keys, we write $\|_{k \in K}$ to denote concatenation of lists indexed by k, e.g., $\|_{k \in K} A_{|k}$ denotes the list of records of A ordered and grouped by $k \in K$. We also sometimes split A into M batches A_m such that $A = \|_{m \in [1,M]} A_m$. $\mathsf{M}(a_i)$ applies an operation M on a_i, while $\mathsf{M}(A)$ applies the operation element-wise on A.

Cryptographic Primitives. Our solutions rely on semantically secure encryption, pseudo-random permutations, and pseudo-random functions. We use the usual cryptographic notions of negligibility and indistinguishability. As we treat cryptographic schemes as abstract building blocks, we avoid committing to either an asymptotic or a concrete security model. Similarly we keep cryptographic keys and their sizes implicit.

Semantically secure encryption [14] guarantees that every encryption of the same message is very likely to map to a different ciphertext. That is, given two ciphertexts the adversary cannot distinguish whether they correspond to two encryptions of the same message or encryptions of two different messages. This strong security property is possible due to a probabilistic encryption algorithm that uses a random nonce every time an encryption algorithm is invoked. We use $[p]$ to denote a semantically secure encryption of a plaintext p. We sometimes overload this notation, using $[D]$ to denote an encrypted dataset D, where each record may be encrypted separately.

The second primitive, a pseudo-random permutation π [16], is an efficiently computable keyed permutation function. Its security property is expressed as indistinguishability from a truly random permutation. That is, if an adversary observes an output of π and a truly random permutation, he is not able to distinguish the two. We use $\pi(i)$ to denote the location of the ith record according to π and, again overload notations, use $\pi(D)$ to denote a dataset that contains the records of D permuted according to π.

The third primitive, a pseudo-random function f [16], is a keyed cryptographic primitive that is indistinguishable from a truly random function with the same domain and range.

Secure Regions/Hardware. Our solutions rely on the ability to protect local processing inside secure regions. Secure regions are trusted containers of code and data that are isolated from other code in a system. Secure regions may be implemented as trusted physical machines, trusted virtual machines, or other forms of trusted execution environments such as SGX enclaves [19]. While our solutions apply independently of the specific implementation of secure regions, we outline an implementation based on SGX processors and used in our experiments. In this case, secure regions are im-

plemented as ranges of virtual memory addresses that are protected by secure SGX processors using three mechanisms.

First, processors control memory accesses to the secure regions. Code inside the region may be invoked only through a call-gate mechanism that transfers control to an entry point inside the region. Code inside the region has full access to the data inside the region, but external read, write, and execute accesses to the memory region are blocked by the processor, even if they originate from code running at a high level of privilege. Thus, the software TCB in our solutions is just the code inside secure regions and, in particular, does not include the operating system or the hypervisor.

Second, processors encrypt and protect the integrity of cache lines when they are evicted to system memory (RAM). This guarantees that the data in the regions is never in the clear outside the physical processor package. This removes a broad class of hardware attacks, such as cold boot attacks, and limits our hardware TCB to only the processor.

Finally, processors support remote attestation. When a region is created, the processor computes a cryptographic digest of the region and signs it with a secret key available only to the processor. This allows an external entity to verify that data originated from a specific secure region. We use this mechanism to establish secure channels between regions and remote systems.

3. MAPREDUCE

3.1 MapReduce

Let D be a dataset that contains n records of equal size. Let (M, R) be a pair of *map* and *reduce* functions defining a MapReduce job on D. Let X be a list of intermediate key-value pairs and O be the output of the MapReduce job executed on D, as explained below. Lower-case subscript for each of the datasets denotes a record of the dataset in the corresponding position (e.g., d_i is the ith record of D).

- M takes a record d_i as input and outputs a list of key-value pairs X_i. Let X collect the key-value pairs produced by M on every record of D: $X = \|_{i \in 1,\ldots,|D|} \mathsf{M}(d_i)$.
- R takes as input records $X_{|k}$ with key k, and outputs a list of values. Hence, the output of MapReduce is $O = \|_{k \in K} \mathsf{R}(X_{|k})$ where K is the set of keys in X.

In cloud deployments, a user uploads D to a cloud datacenter and later requests that the cloud provider executes jobs on D by sending pairs (M, R). The MapReduce framework (e.g., Hadoop) is responsible for invoking M on every record of D in parallel and obtain X as a result; grouping key-value pairs in X by keys; and calling a reduce function R on each resulting group.

3.2 MapReduce on Secure Hardware

We now describe adaptations to the MapReduce framework when executed on secure hardware. The high-level idea of such systems (e.g., VC3) is to store the data, intermediate key-value pairs, and output in encrypted form, and run the map and reduce functions within secure regions (see e.g., §2).

The system is set up with the following changes. The user uploads an encrypted dataset $[D]$ to the cloud. Whenever she needs to request a run of a MapReduce job, she uses a secure channel with the secure regions to provide binaries for functions M and R, numbers M and R, and keys for protecting the data as well as for evaluating a pseudo-random function $f : K \to [1, R]$. The key for accessing the dataset is reused across computations but crucially every MapReduce job uses a fresh pseudo-random function.

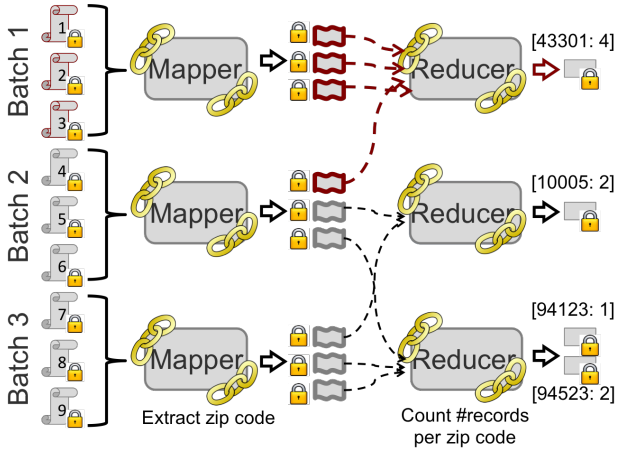

Figure 1: Example of MapReduce on secure hardware (see §3.2) with three Mappers and three Reducers. Dashed lines indicate intermediate traffic observable by an adversary (see §4).

The MapReduce framework then executes the job by invoking M Mappers and R Reducers. At a high level, mappers and reducers execute M and R within a secure region, respectively. We describe their functionality in more detail.

The MapReduce framework splits the records of D into M batches; we write D_m for the mth batch. Each batch is processed as follows: given the mth encrypted batch $[D_m]$, a mapper decrypts it and executes M on each of its records. For every key-value pair x_j with key $x_j.\mathsf{key} = k$ produced by M, Mapper outputs a tuple $(r, [x_j])$, where $r = f(k)$ is the index of a reducer. Every mapper evaluates the same pseudo-random function and thus assigns a record with the same key to the same value of r. Hence, more than one key may be assigned to the same r. The input of the rth Reducer is $(r, [X_{|r}])$ for $X_{|r} = \|_{k \in f^{-1}(r)} X_{|k}$. The index r hides the exact $x_j.\mathsf{key}$ and the ordering of the x_j, but still allows the MapReduce framework to group tuples by keys.

The rth Reducer is responsible for decrypting key-value pairs it receives, grouping them according to keys, executing R on each group, and outputting the encrypted results $[O_r]$.

In the rest of the paper, we refer to the implementation of MapReduce based on secure hardware described in this section, simply as MapReduce framework. Figure 1 shows an example of the system with mappers processing batches of three records and three reducers, where two keys happened to be mapped to the last reducer.

The batch size an adversary can observe varies from system to system and may be deliberately chosen by its designers or may depend on the contingent nature of its performance characteristics. For instance, when running Hadoop unmodified in a secure region, an adversary can delay inputs to mappers to observe batches of a single record. On the other hand VC3 enforces a pre-configured batch size.

4. TRAFFIC ANALYSIS

In the previous section, we described MapReduce on secure hardware using the storage and compute infrastructure of an untrusted cloud provider. By encrypting data stored or exchanged between mappers and reducers, and by executing mappers and reducers within secure regions, it makes an important step towards protecting the privacy of outsourced data and MapReduce computation.

However, there are many aspects of this system that are still observable in the environment where MapReduce is run and, as we show, lead to information leakage.

4.1 What's the adversary?

We first consider a wide range of attacks, and then narrow it down to the main channels we consider in the rest of the paper. (These channels suffice for our attacks, and they are simple enough to yield analytical results for our solutions.) Basically, *our adversary observes runs of MapReduce jobs:*

At the system level, he may record the exchange of encrypted data, either between every node in the system (network traffic analysis) or between every node and storage (storage traffic analysis). The volume of data exchanged may be measured in bytes, pages, packets, or records. Some batching and padding may blur this side channel.

In our examples, we suppose that the number of records can be accurately observed (or deduced) from network traffic. Within one job, as further explained in §4.2, the granularity of each observation may range from individual records to large batches. Conversely, we do not consider cache, timing and other low-level side channels against local runs of the map and reduce functions; devising countermeasures for them is an independent problem.

Instead we focus on higher-level side-channels at the level of the MapReduce framework, namely on network interaction with the outside environment, e.g., reading/writing from/to the long-term storage provided by the cloud data center, or receiving/sending packets from/to another secure region.

At the application level, the adversary may have background knowledge about the job, its input, and its output. Information about the job itself may be readily available, or may be inferred from the shape of the traffic data. (In VC3, for instance, the code for M and R is encrypted; still, the adversary may use, e.g., their data-exchange profile, binary code size, and runtimes to guess the job being executed.)

Statistical information about the input and output data may also be common knowledge, e.g., the adversary may know the distribution of marital status.

In our attacks, unless explicitly mentioned, we assume that the adversary knows the job and some statistics about the data, but not the content of its input and output—except for their total sizes. (Such aggregate information is in general hard to hide.)

In our security definition, we model arbitrary background knowledge by letting the adversary choose two input datasets for which the background knowledge is the same while data contents may differ; the adversary is then challenged to infer which dataset was computed on. This more demanding definition lets us capture that the adversary cannot learn anything besides his background knowledge by observing traffic.

Our adversary may also actively interfere with job runs:

At the system level, an active adversary may control resources and scheduling, e.g., feeding a Mapper one record at a time. However, as discussed in §2, we assume he cannot directly alter encrypted traffic or break into secure regions.

At the application level, he may partly choose his own input, or even his own jobs, to mount adaptive attacks. Our security definition reflects such capabilities by letting the adversary adaptively choose the jobs as it is trying to use traffic analysis to learn information about the dataset.

Our adversary may observe a sequence of jobs, on the same datasets, or related datasets. MapReduce implementations re-use inputs for multiple jobs, inasmuch as they co-locate the input batches and the mappers on the same nodes. They also avoid costly re-encryption of data between jobs. As a qualitative example, assume the adversary observes runs of a job before and after adding a single, target record in the dataset. If the job splits on an attribute with few keys and a known distribution, then the attribute for the new record can be precisely inferred.

In practice, the more selective the jobs are, the more precise information we can extract (assuming we know, or we can guess, what the job is). In the following, we mostly focus on observing jobs that split all input records depending on some of their attributes. More generally, MapReduce jobs may first filter out parts of the input records before splitting. Our attacks would similarly apply to those jobs (in particular to multiple jobs with the same filter), except that we would learn information specific to the smaller, filtered collection of inputs, rather than the whole dataset.

Intermediate Traffic. Our examples primarily target traffic from mappers to reducers and how it can be combined with background information about the data to infer the content of the data. Observing the reducer outputs can also be informative, in particular for selective jobs known to the adversary (e.g. "Does the dataset contain this record?"). However, reducers must usually wait till they have seen all intermediate key-value pairs for any given key, thereby leaking relatively little traffic information.

Example: Attacking VC3. In VC3, an adversary may gain full control of the Hadoop scheduler, enabling it, for example, to send the same input batch to multiple mappers, or to send a single input batch to a mapper; it may even cause mappers to fail or restart, enabling it to improve the accuracy of his network traffic measurements. On the other hand, the input batches (and their sizes) are GCM-protected, so the adversary cannot change them. VC3 reducers also incorporate integrity checks against any input-batch replication: a reducer that receives intermediate key-value pairs from different mappers processing the same input batch will detect the attack and safely stop processing data.

Hadoop tries to keep nodes stateless, hence they rarely delay sending data between batches. In VC3, mapper-reducer communications rely on stateful secure channels for the whole job; however, the adversary may send input batches one at a time, and measure how many bytes are communicated as a result of their processing.

Overall, we can thus conservatively model this adversary as 'passive', but able to collect precise network traffic at the granularity of individual input batches.

Example: Attacking Unmodified Hadoop. An adversary against unmodified Hadoop (running in secure regions, and encrypting all data, but without the VC3 countermeasures) may have finer control of its scheduling, for example by delaying packets in data streams, or exploiting failure-recovery to observe multiple runs of the same jobs with a different assignment of inputs to mappers, thereby collecting traffic information at a finer granularity, possibly for each record.

4.2 Observing Intermediate Traffic

We model the data collected by an adversary observing MapReduce traffic, then we explain how he can use it to trace information from the output of a MapReduce job back to individual input records and how, as he observes runs of multiple jobs on the same input records, he can correlate this information between jobs.

Data dependent intermediate traffic. We model observations of intermediate traffic using a matrix $\mathbb{A}$ with dimensions $M \times R$ where M is the number of mappers and R is the number of reducers. $\mathbb{A}[m, r]$ is the number of intermediate key-value pairs sent from mapper m to reducer r. Since an adversary observes input and output of every mapper and reducer, he can easily construct this matrix.

Before analyzing how he can use $\mathbb{A}$ to learn more about the input dataset, we give an intuition with an aggregate MapReduce job in Figure 1. The matrix $\mathbb{A}$ for this job is shown below, with aggregate volumes of data sent by each mapper (right) and received by each reducer (bottom).

m/r	1	2	3	
1	3	0	0	3
2	1	1	1	3
3	0	1	2	3
	4	2	3	

Every Mapper reads three encrypted records, extracts a zip code and each Reducer counts the number of records per zip code. In the matrix, each entry $\mathbb{A}[m, r]$ indicates how many intermediate key-value pairs produced by the mth mapper have zip code that was returned by the rth reducer. In particular, the adversary sees that the first three records have the same zip code (43301) and the last three records do not have this zip code. Given background knowledge of the distribution of zip codes, the adversary can thus, in this case, label each column of $\mathbb{A}$ with a zip code. Abusing our notation, we refer to the cell $\mathbb{A}[1, 1]$ as $\mathbb{A}[1, \text{`}43301\text{'}]$.

The example illustrates that matrix $\mathbb{A}$ lets the adversary correlate input and output of a MapReduce job as long as (1) records read by a mapper can be correlated with intermediate key-value pairs in $\mathbb{A}$, and (2) there is variation between values in each row and column. The first condition depends on how mappers read and write their input and output, while the second condition depends on the data.

Network traffic from two or more jobs can easily be combined and lead to a 'job composition' attack. The adversary observes a matrix $\mathbb{A}$ from each job and, as long as the same input data is used, he can label each input batch with the results of such inferences. For example, he can observe jobs on zip code, gender and data of birth. Sweeney [27] showed that combinations of such simple demographics often already identify people uniquely. In the rest of this section we show how the adversary can still correlate mapper's inputs and outputs for less trivial input datasets.

Granularity: observing traffic on input batches. In general a mapper can process a sequence (or batch) of records (to amortize the cost of encryption, for example). If the mapper reads a batch, there are several ways in which it could control its I/O. For example, it could sequentially read a record and immediately return the corresponding key-value pair; it could buffer key-value pairs for several records and return all of them when the buffer is full (as in the VC3 implementation); or start reading the next sequence of records while still processing the first sequence.

Different I/O processing creates noise in the signal of the adversary when he tries to correlate the input and the output of a mapper. For example, this noise does not allow the adversary to precisely determine which record resulted in which key-value pair. However, the adversary can still correlate a batch of input records with key-value pairs, i.e., by using a time window for when records are read and intermediate key-value pairs are returned. Similar I/O buffering can be done on the reducer side. However, due to the functionality of the reducer, in some cases it has to read all its input records before returning the output.

In our examples of information leakage, we assume that the mapper would try to protect the correlation between records it reads and intermediate key-value pairs it returns. In particular, we assume that the mapper puts a threshold on how many records it has to process in a batch before returning the output. He further permutes the intermediate key-value pairs to break I/O correlation. However, as we illustrate below, this is only a partial remedy.

4.3 Exploiting Intermediate Traffic

We give concrete evidence of information leakage, both when records are processed one at a time and when they are processed in batches. In the latter case, although it is more difficult to extract information about individual records, we show that it remains possible when the input records are somewhat sorted, and that MapReduce traffic still leaks information about many statistics in the input data.

Our goal is not to uncover new facts about these datasets, readily available from their plaintext, but to show that, more surprisingly, those facts are also available to an adversary that merely observes encrypted traffic. Our experiments also suggest that naive techniques based on padding inputs and outputs would be of limited value for these datasets.

Our experiments are based on two datasets:

- **U.S. 1990 Census Sample [18] (900 MB).** The dataset contains 2.5 million personal records. Every record has 120 attributes, including the `Age`, `Gender`, `POW` (place of work), `POB` (place of birth), `MS` (marital status), etc. Some attributes have been discretized: for instance, `Age` ranges over 8 age groups, such as 20–29.

- **New York 2013 Taxi Rides [28] (24 GB).** This dataset contains records for all the taxi rides (yellow cabs) in New York city in 2013. It is split in 12 monthly segments, and each segment contains approximately 14 million records. The records have 14 attributes and describe trip details including the hashed license number, pickup date and time, drop off date and time, and number of passengers.

The first dataset is representative of personal data commonly stored in the databases of medical institutions, insurance companies, and banks. The second dataset contains sensitive information and, despite some basic anonymization, is susceptible to inference attacks [23, 30]. Some of these attacks use MapReduce [23] to extract correlation between the rides (in plaintext). We show that the same kind of information can also be extracted by traffic analysis.

In this section, the adversary is assumed to have the following subset of the capabilities described in §4.1. He observes only basic aggregate jobs, which all go as follows: M splits the records, with the attribute used for aggregation (e.g., the `Age`) as key; hence R receives all records with the same attribute value; it may return their count, or any other function of their contents. He is also assumed to have statistical information on the attribute values used for splitting (e.g., distribution of age and marital status in the U.S. and popular destinations in New York). This allows him to label columns of $\mathbb{A}$ with the corresponding attribute values.

4.3.1 Individual records

Our first attacks are based on observing mappers, as they consume one record at a time and immediately produce intermediate data. Hence, the intermediate-traffic matrixes have one row for each individual record and, at least for basic aggregate jobs, each row has exactly one non-zero entry. To illustrate the correlation of observations across jobs, we show that, after observing aggregate jobs on distinct attributes, the adversary is able to answer specific queries on *any combination of these attributes*, such as

1. Given the index of a record in a dataset, return the values of these attributes;
2. Test if the dataset contains a record that matches particular values for some of these attributes; and
3. Given the values of some of these attributes, infer the possible values of the others in the dataset.

Census Data. For these attacks, we observe three aggregate jobs, one for the age group, one for the place-of-birth, and one for marital status. This yields three intermediate-traffic matrixes: $\mathbb{A}_{\texttt{Age}}$, $\mathbb{A}_{\texttt{POB}}$ and $\mathbb{A}_{\texttt{MS}}$ with 2.5M rows each.

Figure 2 displays aggregate counts for the three jobs, i.e., the number of key-value pairs assigned to each attribute value. Up to a permutation of the columns, this is the same information as the sums of the columns in $\mathbb{A}_{\texttt{Age}}$, $\mathbb{A}_{\texttt{POB}}$ and $\mathbb{A}_{\texttt{MS}}$. The adversary can determine the key processed by every reducer in these matrices (i.e., label the columns of $\mathbb{A}$) using auxiliary public information on the distribution of, for example, the age of the U.S. population.

Let us analyze the information in each matrix individually. $\mathbb{A}_{\texttt{Age}}$ gives a precise age group for every record, $\mathbb{A}_{\texttt{POB}}$ gives a geographical region for a place of birth for every record, and $\mathbb{A}_{\texttt{MS}}$ gives the marital status for every record. As long as all jobs processed the same dataset, the adversary can combine the information he learns across all jobs. That is if, $\mathbb{A}_{\texttt{Age}}[i, \text{'1–12'}] = 1$ (overloading the reducer key with the label it processed) and $\mathbb{A}_{\texttt{POB}}[i, \text{'Africa'}] = 1$, then the ith record is in the age group "1–12" and was born in Africa.

Thus, the adversary can directly answer queries such as

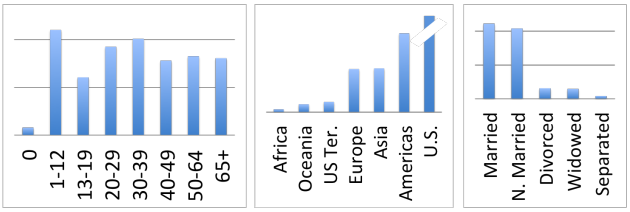

Figure 2: Distribution of Census records across age groups (left), place of birth (center) and marital status (right), where U.S. count is trimmed.

1. *What is the marital status of person #1,326,457 in the census?* Never married, since $\mathbb{A}_{\texttt{MS}}[1326457, \text{'Never Married'}] = 1$.
2. *Is there a person with {`Age: 13-19, POB: Oceania, Marital Status: Divorced`} in the dataset?* Yes, since there is (exactly, in this case) one index $i = 1,005,243$ such that $\mathbb{A}_{\texttt{Age}}[i, \text{'13-19'}]$, $\mathbb{A}_{\texttt{POB}}[i, \text{'Oceania'}]$ and $\mathbb{A}_{\texttt{MS}}[i, \text{'Divorced'}]$ are all equal to 1.

Taxi Data. Our sample attack is based on observations of aggregate job on the pickup day, pickup location, and drop-off location. Suppose an adversary saw a person getting in a taxi on a corner of Linden Blvd and 221st Street in Queens on January 13, 2013. The adversary then looks at row indices in $\mathbb{A}_{\texttt{PickupD}}$ and $\mathbb{A}_{\texttt{PickupL}}$ that have non-zero entries for 'Linden Blvd and 221st Street, Queens' and 'January 13', There is exactly one such index in our dataset, $13,484,400$. The drop-off location is the non-zero entry in $\mathbb{A}_{\texttt{DropoffL}}[i]$ row, that is, '1053 Atlantic Ave, Brooklyn'.

4.3.2 Batch records

Our second series of attacks apply against mappers that securely process large batches of records at a time. We assume that each mapper reads all records in a batch assigned to him, applies M on each record, and returns permuted key-value pairs. Hence, observing an aggregate job yields an intermediate-traffic matrix $\mathbb{A}$ with fewer rows (only one for each batch). Since each job we consider has only a few keys (at most 50) we still assume that there is a reducer for every key. Hence, the adversary knows precisely which intermediate keys in the columns of matrix $\mathbb{A}$ produced a given output value. Thus, each row provides a histogram of the values of the attribute for all the records in the batch.

Though intuitively, batching makes it harder to extract precise information from the dataset, we show that it does not always suffice. In particular, if the data is ordered by some attribute, the information about a batch will provide information conditional on values of that attribute.

In the following experiments, our sample datasets are split into batches, sequentially, as follows: each batch of Census data contains ∼240K records (90Mb), with 10 batches in total; each batch of New York taxi rides contains ∼147K records (24Kb), with 100 batches per monthly segment.

In both cases , an adversary that observes an aggregate job on some arbitrary attribute (`attr`) recorded in the dataset can reliably perform the following tasks:

1. Given an aggregate count of the values of `attr` over the whole dataset, he can infer precise counts of `attr`

values over smaller batches of data (for each U.S. state in the Census, andfor each day of the year for the taxi rides).

2. Given prior information about a specific record, such as its location in a dataset or the value of its attribute, he can infer other attributes for that record (i.e., the place of birth for a personal record, and the pick up day for a taxi ride.)

Census Data. Assume the adversary observes two aggregate jobs for `POW` (place of work) and `POB` (place of birth) and is given access to their aggregate counts. Hence, he is also given the two intermediate-traffic matrixes $\mathbb{A}_{\texttt{POW}}$ and $\mathbb{A}_{\texttt{POB}}$ of these jobs. Instead of linking only one record location in the dataset to a row in $\mathbb{A}$ (as in our simpler attacks), the adversary has to link a batch of records to a row. Hence, he infers a (weighted) set of attribute values for their records, instead of one exact value.

Let us first look closer at $\mathbb{A}_{\texttt{POW}}$. $\mathbb{A}_{\texttt{POW}}$ has 10 rows (since there are 10 batches) and 54 columns, one for each value of `POW`: fifty states, one district, and three special values 'abroad', 'non-specified', and 'not-a-worker'.

The distribution of data across the columns varies as expected. The top four values are 1,339,590, 134,913, 85,000 and 76,766 (recall that this is the number of key-value pairs processed by the corresponding reducers). We can assume that the output of this job is available in the clear and the adversary learns the key processed by each reducer. However, this may not be necessary if auxiliary information is used. Using data available online, one can easily infer that the first group is for 'not-a-worker's, and the next three groups correspond to California, New York and Texas. (In 1990 the population of New York was larger than in Texas or Florida.)

In Figure 3 we plot the number of records assigned from each batch to 53 reducers (we omit 'not-a-worker' option since every batch contributed ∼134K records to it).[1] In other words, Figure 3 is a graphical representation of $\mathbb{A}_{\texttt{POW}}$ where x-axis displays reducer keys (i.e., attribute values of `POW`) and bars show the distribution of batches (rows) in each column (i.e., number of key-value pairs from every batch assigned to a specific attribute value). For example, the fifth bar indicates that there is an attribute value (reducer) that received most of its records from Batch 1 (red) and Batch 2 (green).

From Figure 3, it is evident that records from 10 batches are not assigned uniformly across values of `POW` field (except for 'not-a-worker' option). Moreover, if we assume that most people live and work in the same state, then the plot suggests that the original data was sorted by the state of residence of the survey participants (note that this field is not part of the available dataset and we did not sort the data). That is, the red batch, first five columns, stores data from participants living in states Alabama through California.

$\mathbb{A}_{\texttt{POB}}$ has the same information in its columns as those in Figure 2 (center). However, rows, instead of representing a specific record, show the distribution of a batch of records across values for place of birth. In Figure 4 we illustrate graphical representation of $\mathbb{A}_{\texttt{POB}}$: x-axis are the attribute values for `POB` and the bars show how many intermediate key-value pairs assigned to each of them from all batches.

[1]These figures are best viewed in color.

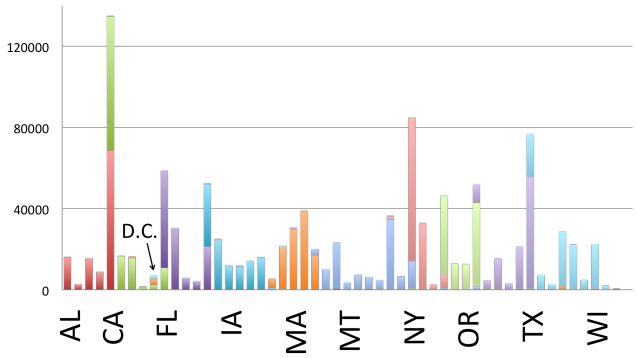

Figure 3: Distribution of records from 10 batches across reducers for "Aggregate by place of work (state)" MapReduce job. The plot suggests that records in the dataset are sorted according to residence state of census participants.

Since the adversary inferred that batches are split by state, Figure 4 uses state names for the batch names.

Given $\mathbb{A}_{\texttt{POW}}$ and $\mathbb{A}_{\texttt{POB}}$ of the network traffic, we show how the adversary is able to answer each question below:

- *What is the distribution of place-of-residence for employees of Washington D.C.?* Figure 3 contains precisely this information. Given a place-of-work state, the adversary looks at the x-axis (ninth bar) to learn precise `POW` counts: 2,321 from green batch, 2,839 from orange batch, 1,682 from the last batch. Since each batch contains only few states, combining this information with the map of the U.S. indicates that the corresponding states of residence are Delaware, D.C., Maryland and Virginia.

- *What is the distribution of place-of-birth for residents of California?* Figure 4 contains this information and lets the adversary compare how these numbers differ from the average and between states. As can be seen, these counts differ significantly from the aggregate values that the adversary learns by looking at the output of the aggregate job for `POB` in Figure 2 (left).

- *What are the likely residence state and place of birth for person 1,721,803 in the Census?* The adversary finds the batch that includes record #1,721,803 (light green batch in Figure 3) and looks up where this batch appears in Figure 4. (In other words, he finds the row of the corresponding job in the matrices and checks the distribution of values in this row across attribute values.)

Finally, we note that anonymity in the Census data is fragile. For example, there are 238 responses with 'non specified' option for `POW`. Who refused to reply? Combining $\mathbb{A}_{\texttt{POW}}$ and the guess that the dataset is sorted by residence state, the adversary infers that these participants live either in Maine, Maryland, Massachusetts, Michigan, or Minnesota (since all records that were aggregated for the option 'non-specified' were from the orange batch).

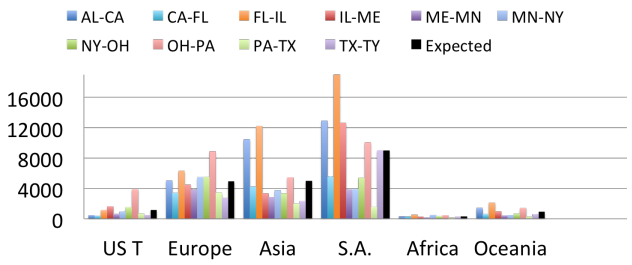

Figure 4: Distribution of records from all batches across reducers for "Aggregate by place of birth" MapReduce job. Batch names are derived from Figure 3. Distribution of batches for "U.S." is omitted for readability. Expected values (black bars) are computed using batch size and distributions in Figure 2 (center).

Taxi Data. Consider a curious adversary who observes intermediate traffic for two aggregate jobs, on `PickUpD` (pick-up day of a taxi ride) and `PassenN` (number of passengers in a taxi ride).

Let us first describe the information available in $\mathbb{A}_{\texttt{PickUpD}}$ and $\mathbb{A}_{\texttt{PassenN}}$. Since it is known that the data in every 100 batches corresponds to a month of the year, we look at the rows that correspond to January. In particular, we consider the same 31 rows of $\mathbb{A}_{\texttt{PickUpD}}$ and $\mathbb{A}_{\texttt{PassenN}}$ (since there are 31 days in January). For each row, we consider its distribution across attribute values (i.e., the days of the year). As expected, there are only 31 non-zero counts in each row.

We plot the number of key-value pairs assigned from each batch to 31 reducers in $\mathbb{A}_{\texttt{PickUpD}}$ in Figure 5 (top) (i.e., similar to Figure 3 for Census dataset). For example, 31% of the records for day 'Jan 1, 2013' came from the dark-blue batch. It is evident that the data is not uniformly distributed. For comparison, the bottom half of Figure 5 plots the attacker's guess on batches distribution if he knew only the aggregate counts for day, i.e., the column sums of $\mathbb{A}_{\texttt{PickUpD}}$. It appears that the data is sorted according to individual days and the 100 batch split divides some of the days in several batches. Furthermore, we note that the original dataset was probably slightly shuffled, however, the day order is still preserved. In particular we see that 98% of Batch 7 (purple batch) was assigned to Jan 3.

In Figure 6 we capture the distribution of some of the rows across all values in $\mathbb{A}_{\texttt{PassenN}}$. Since we know that each batch contains rides of only a few days, we label the batches with the day that is represented the most in that batch. There are 6 passenger counts (there are 0, 9, 208, 255 counts that we ignore since they are also under represented and suggest an error in the dataset). If the adversary sees the output of the aggregate job for `PassenN`, he knows precisely the passenger count. If not, some of the labels are easily inferable (e.g., 4 passenger rides are the most rare while 1 person ride is most popular).

Given $\mathbb{A}_{\texttt{PickupD}}$ and $\mathbb{A}_{\texttt{PassenN}}$, the adversary can thus answer the following queries:

- *What is the number of passengers on Friday nights?*

  This information is available in Figure 6 (or the full version for all days). The adversary simply looks up the bars that corresponds to Fridays. Note that this information is more precise than what the adversary can learn from aggregate information for passenger number (i.e., the last bars of Figure 6). If the adversary is a competitive taxi company these counts could be used for better scheduling of own cars across the week.

- *Did someone take a taxi ride from Canarsie Park in Brooklyn in January? When exactly?*

  The adversary uses combined traffic from the two aggregate jobs. Luckily, there is only one record assigned to pick up location in Canarsie Park in the 100 batches of January. Moreover, the batch that includes the record has most of its taxi rides from January 19 (79K), January 18 (39K), January 20 (24K), and January 6 (6K), with only 11 rides left over 6 other days. Hence, it is most likely that this ride happened on January 19 (and in fact, it did).

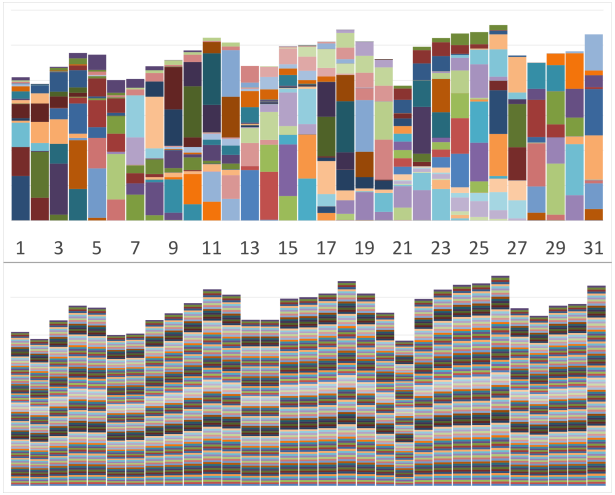

Figure 5: Distribution of taxi records from 100 batches across reducers for "Aggregate by pickup day" job (top) vs. uniform, i.e., attacker's guess without observing network traffic (bottom).

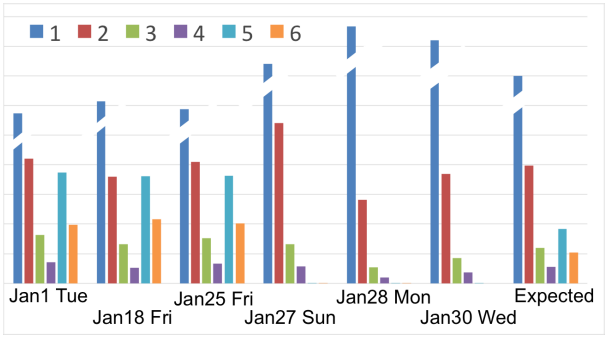

Figure 6: Passenger counts across six days and the expected count if the attacker observed only aggregates (bars for single passenger rides are trimmed).

Finally, we point out that the difference in the distribution of keys in batches is high (e.g., a batch for January 1 vs. a

batch for January 30 differ by 20K rides for 5 and 6 passengers). Similarly, the number of key-value pairs processed by reducers is also different (e.g., reducers that count rides with one passenger vs. five passengers). Hence, padding the inputs to some number of keys to protect the identity of each reducer may become very expensive for long tail distributions. Even for the taxi dataset it would require mappers to pad their traffic to each reducer to 125K key-value pairs. Moreover, it is not clear how mappers can fix this level of padding without communicating with each other.

We note that our examples did not consider filter jobs, i.e., those that select records based on some filter. Such jobs reveal even more information since they give precisely the number of records in each batch that satisfy the filter.

5. SECURITY DEFINITIONS

How can we design systems that are resilient to such traffic analysis attacks? To be able to evaluate protection mechanisms, this section brings forward different security notions for MapReduce on secure hardware. We use the notation of §3.2 and in the following, relate the adversaries that we are considering to the discussion of attacks in §4.1. To protect the input dataset D, we wish MapReduce computations to appear data independent according to the observed traffic and depend only on (1) the M and R functions being computed, (2) the input size $|D|$, and (3) output size $|O|$. This corresponds to an adversary that has some knowledge about the code of M and R and is able to observe some basic performance characteristics.

Ideally, traffic observed from the same MapReduce computation for any two datasets D^0 and D^1 should appear to be the same and not reveal anything about the data. Certainly if $|D^0| > |D^1|$, or vice versa, such an adversary can trivially distinguish the two cases. The same holds for the output of the computation. We thus require that $|D^0| = |D^1|$ and $|O^0| = |O^1|$.

This requirement is, however, still too strict. The reason being that any solution with such a property has to hide the worst case: consider two databases D_0 and D_1 and a map function M, such that M maps each record of D^0 to many records in X^0 all with the same key k, i.e., $\|_i \mathsf{M}(D_i^0)_{|k} = X^0$. At the same time M may map each record of D^1 to records in X^1 with mutually distinct keys. To distribute the computation among R reducers, any secure solution has to send $|X^0|$ records to all R reducers. This largely destroys any advantage initially gained through distribution.

We propose two definitions that avoid this dilemma. The first guarantees indistinguishability only for databases D^0 and D^1 that result in distributed MapReduce computations with the same input and output characteristic for mappers and reducers. Each Mapper must produce same sized output for both D_i^0 and D_i^1, and each reducer (up to a permutation on reducers) must take same sized input and produce same sized output for both D^0 and D^1. We do however hide the amount of data $|\mathsf{M}(D_i)_{|k}|$ that flows between each map and reduce job. In terms of observed intermediate traffic, this corresponds to the adversary learning only the row and column sums for the matrix $\mathbb{A}$ of §4.2. This appears to be the best we can do without a large amount of padding.

For our second definition, we look at the maximum number of records $\max_{k \in K}(|\mathsf{M}(D)_{|k}|)$ that a function R has to process and require that our solution does not reveal anything more than that. This corresponds to revealing only the maximum column sum of matrix $\mathbb{A}$.

5.1 Formal definitions

Our definitions are defined as games between a challenger and an adversary: the adversary chooses two datasets that share some arbitrary background information available to the adversary, the challenger chooses one of them and lets the adversary adaptively request jobs on it. (For instance, arbitrary background information may include "knowledge" of the output of a job computation; this is captured by the adversary choosing two inputs that yield this output.)

Definition 1 (MapReduce Game). *At the start of the game, the adversary picks two datasets D^0 and D^1 and fixes the number M and R of mappers and reducers. He is then given access to $[D]$, where $D = D^0$ or $D = D^1$, and can observe the run of multiple MapReduce jobs (M, R) of his choice. He observes their full network traffic to guess whether $D = D^0$ or $D = D^1$. The adversary's advantage of winning the game is his probability of guessing correctly minus $\frac{1}{2}$.*

The MapReduce game models that the traffic observed for any two datasets D^0 and D^1 should appear to be the same and not reveal anything about the input data. As argued in §4.1, the integrity checks of secure MapReduce systems such as VC3 prevent active interference of the adversary on the network; this enables us to model their adversaries as passive in that sense.

As stated, the MapReduce game is trivial to win, e.g., if $|[D^0]| \neq |[D^1]|$, and, as discussed in §4.1, it is reasonable to assume that the adversary may learn the size of D and a few other aggregate values. We give two variants of our definition, of increasing strength, that specify what each job is allowed to leak about the data as *requirements* on D^0 and D^1, expressed on their matrices $\mathbb{A}^0$ and $\mathbb{A}^1$. Recall that we associate the matrix $\mathbb{A}$ to a dataset D and a MapReduce job (M, R) such that each cell represents the number of intermediate key-value pairs from the mth batch to key k, that is, $\mathbb{A}[m, k] = |\mathsf{M}(D_m)_{|k}|$ (using notations from §3).

Definition 2. Correlation hiding *requires that no efficient adversary has more than a negligible advantage in winning the MapReduce game as long as $|D^0| = |D^1|$ and, for every MapReduce job (M, R) the adversary picked during the game, the following holds:*

1. *Mappers produce the same amount of output, i.e., for all $m \in [1, M]$, we have $\sum_k \mathbb{A}^0[m, k] = \sum_k \mathbb{A}^1[m, k]$.*
2. *Reduce functions take the same amount of input, i.e., there exists a permutation σ on the keys such that, for all keys $k \in K$, we have $\sum_m \mathbb{A}^0[m, k] = \sum_m \mathbb{A}^1[m, \sigma(k)]$.*
3. *The output size of the reduce function R is constant.*

The permutation in Requirement 2 accounts for the fact that the adversary only observes encrypted keys. The definition does not leak the details of $\mathbb{A}[m, k]$, hence it hides which records have common keys, preventing the composition attack in §4.3. It is applicable to map functions M that project a key and a value from a record; typically to allow a reducer function R to then compute some aggregate statistic about the values grouped by the key: Requirement 1 and 3 are clearly met by such functions, and Requirement 2 is a

statistic, such as those in Figure 2, about the distribution of keys in D that may often already be publicly known.

For example, a solution meeting this definition protects against the composition attack based on {`Age`: 13-19, `POB`: Oceania, `Marital Status`: Divorced} in §4.3. The only information leaked is how many people are aged 13-19, how many people live in Oceania and how many people are divorced.

MapReduce jobs for which this definition does not do well are functions (M, R) that perform filtering. A map function M that discards all people living in Oceania from processing leaks this attribute value trivially if no record was returned. Our second definition protects even against such attacks.

Definition 3. Strong hiding *requires that no efficient adversary has more than a negligible advantage in winning the MapReduce game as long as* $|D^0| = |D^1|$ *and, for every MapReduce job* (M, R) *the adversary picked during the game, the following holds:*

1. *The volume of intermediate data is the same:* $|X^0| = |X^1|$*; and the number of keys is the same:* $|K^0| = |K^1|$*;*
2. *The number of records for the most popular key is the same:* $\max_k(\sum_m \mathbb{A}^0[m,k]) = \max_k(\sum_m \mathbb{A}^1[m,k])$.
3. *The output size of the reduce function* R *is constant.*

We give intuition behind each condition. Requirement 1 states that the size of network traffic at each stage of MapReduce has to be the same. That is, the output size and the number of intermediate key-value pairs is the same between two challenge datasets, since otherwise they are trivially distinguishable. Requirement 2 states that the number of intermediate key-value pairs that correspond to the most popular key is the same for both datasets.

The two security definitions differ in their requirements on D^0 and D^1, which restricts the type of data and sequence of (M, R) computations the definition can protect. The latter definition is strictly stronger, since agreement on the cardinality of keys implies agreement for the most popular key, as well as on the sizes of $|O|$ and $|X|$. Requirement 2 allows us to leak the number of records for the most popular key, which is a lower bound on the traffic received by the reducer that must process all these records.

6. SHUFFLE-IN-THE-MIDDLE SOLUTION

Our first solution prevents intermediate traffic analysis on a job by securely shuffling all the key-value pairs produced by the Mappers and consumed by the Reducers. Hence, the adversary may still observe volume of intermediate traffic for each mapper and for each reducer, but it cannot trace traffic from reducers back to individual mappers.

We present our solution using a data shuffle algorithm as a black box that, given $[X]$ and a pseudo-random permutation π on $1 \ldots |X|$, returns $[\pi(X)]$. We then describe our implementation of the Melbourne Shuffle algorithm [22] using MapReduce jobs (§6.1). We finally show that our solution meets Definition 2 (§6.2).

Let X_{M} be the output of the mappers, and X_{R} the output of the shuffle passed to the reducers. X_{M} and π are given as input to a data shuffle job to permute the records. The output X_{R} of the shuffle is then grouped and sent to the Reducers by the MapReduce framework, as before.

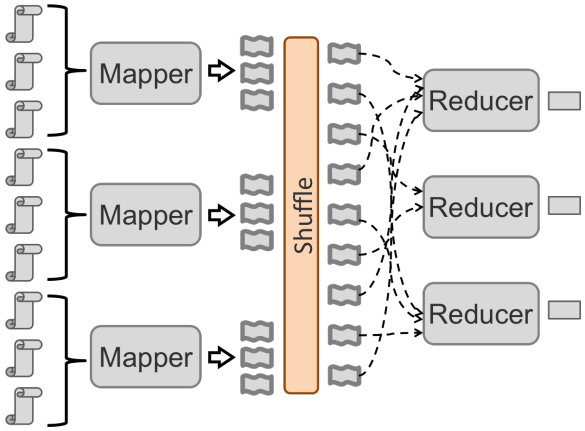

Figure 7: Overview of the Shuffle-in-the-Middle solution (see §6) where all data elements are encrypted and Mapper and Reducer code is executed inside of the secure region. In our solution the shuffle is implemented as two MapReduce jobs.

In more details, each Mapper proceeds similarly to §3.2, except for the content of its output. Recall that a mapper in §3.2 returned a tuple $(r, [x_j])$ where r is the index of the Reducer that processes records with keys k such that $f(k) = r$. Instead, we modify Mapper to return $([r], [x_j])$, so that $[X_{\mathsf{M}}]$ now consists of pairs of ciphertexts.

Then, the data shuffle is invoked on $[X_{\mathsf{M}}]$ with a small adaptation: instead of simply outputting $[\pi(X_{\mathsf{M}})]$, the last step is modified to return the decrypted value of r, while re-encrypting $[x_j]$. Hence, the output of the data shuffle is a list of tuples $(r, [x_j])$ that is a random permutation of the intermediate key-value pairs of the original MapReduce job.

The rest of the protocol is the same as the one for MapReduce on Secure Hardware in §3.2. The MapReduce framework (e.g., Hadoop) groups key-value pairs according to their reducer index r and invokes Reducer on each group.

An illustration of the Shuffle-in-the-Middle solution is given in Figure 7.

6.1 Data Shuffle

Given an encrypted dataset $[D]$ as input, the shuffle yields some permutation $[\pi(D)]$ as output. Since D can be large, we want an efficient implementation of the shuffle within a secure MapReduce framework. Moreover, we want to ensure that the observations about the network traffic that the adversary can make (as described in §4.2) do not leak any information about the data (except its size) and the shuffle permutation π. Hence, an adversary that observes a job implementing either π^0 and π^1, should not be able to say whether the output is an encryption of $\pi^0(D)$ or $\pi^1(D)$.

Sorting networks [5, 1] provide the security guarantees above: their network traffic is independent of the data. However, since these algorithms perform sorting, they incur a logarithmic depth computational overhead (plus additional constants). Instead, for our solutions, we choose the parallel version of the Melbourne Shuffle [22], which offers the same security guarantees, and we implement it as *two* successive runs of the MapReduce job described below. We refer to [22] for a detailed analysis of the algorithm.

Algorithm 1 Melbourne Shuffle Mapper: $\mathsf{Mapper}([d_i, \ldots, d_{i+b}])$ with π, R, max included, for example, in the binary.

Let $d_i, \ldots, d_{i+b}$ be input records in a batch of size b.
Let π be the target secret permutation.
Let R be the number of reducers.
Let max be the max number of records to be sent from a mapper to a reducer.
for $r \in \{1 \ldots R\}$: $\mathsf{bin}[r] \leftarrow []$
for $j \in \{i \ldots i+b\}$:
    $\mathsf{id} \leftarrow (\pi(j) \mod R) + 1$
    $\mathsf{bin}[\mathsf{id}].\mathsf{append}((\pi(j), d_j))$
for $r \in \{1 \ldots R\}$ **do**
    if $\mathsf{len}(\mathsf{bin}[r]) > \mathsf{max}$ **abort**.
    while $(\mathsf{len}(\mathsf{bin}[r]) < \mathsf{max})$: $\mathsf{bin}[r].\mathsf{append}(\mathsf{dummy})$
end for
for $r \in \{1 \ldots R\}$ **do**
    output $r, \mathsf{bin}[r]$
end for

Each mapper takes as input a permutation π (e.g., it takes a key to a pseudo-random permutation) and a batch of b records, and outputs a bin of max records (for some fixed number $\mathsf{max} > b/R$) for each reducer, that is, R bins in total. The mapper assigns each record d_j (where j is the index of the record in D) in the batch to one of the R bins according to its permutation tag: record d_j goes to bin $r = \lceil \pi(j)/R \rceil$. If a bin is assigned more than max records, the algorithm aborts. Otherwise, the mapper pads each bin to max records, by adding dummies with the same size as genuine records, then it encrypts and outputs each bin as a single intermediate value with key r. Pseudocode for the mapper is given in Algorithm 1.

Each reducer takes a list of bins (one from each mapper), removes the dummies, sorts the remaining records by their permutation tags, removes the tags, and outputs the result. Pseudocode for the reducer is given in Algorithm 2.

The security of the shuffle relies on the deterministic traffic that the job above produces: each mapper produces the same number of bins and the size of each bin is the same. Hence, every reducer also receives the same traffic. Its output size is always $|D|/R$ (ensured by how mappers distributed their input).

A single run of the MapReduce job above will fail on some permutations, namely those where a mapper assigns more than max records to the same reducer. For example, if π is the identity function, the job will fail unless $\mathsf{max} \geqslant b$. To remedy this while keeping max small, two successive MapReduce jobs are invoked: the first job is for a uniformly random permutation ρ and the second one is for the target permutation π. Although these invocations may still fail, this happens rarely. Moreover, the analysis of [22] shows that success and failure of the shuffles depends only on ρ, hence, it leaks no information about the actual input and output of the algorithm. Besides, max can be carefully chosen to control the probability of failure: on average, each bin should get b/R records, and balls-and-bins analysis tells us how much to over-provision as we choose max.

6.2 Analysis

THEOREM 1. *If $[\cdot]$ is a semantically secure encryption scheme, f is a pseudo-random function and π is a pseudo-random permutation, then the Shuffle-in-the-Middle solution is correlation hiding (Definition 2).*

Algorithm 2 Melbourne Shuffle Reducer: $\mathsf{Reducer}(r, [X|_r])$

Let r be this reducer's index.
Let $X|_r$, be input values with key r (i.e., all bins with r).
$\mathsf{vals} \leftarrow []$
for $\mathsf{val} \in X|_r$ **do**
    if $\mathsf{val} \neq \mathsf{dummy}$: $\mathsf{vals}.\mathsf{append}(\mathsf{val})$ $\{\mathsf{val} \text{ is } \pi(j), d_j\}$
end for
Sort vals by π tag, strip off tags and output the result.

For the proof, we instantiate our solution with the Melbourne Shuffle algorithm. However, it is easy to see that our solution is secure as long as it relies on any shuffling technique that produces data-independent traffic.

Relying on the semantic security of encryption, the Melbourne Shuffle guarantees that its network traffic (i.e., $\mathbb{A}$ matrices) for the two MapReduce jobs implementing π depend only on the size of the dataset and the number of mappers and reducers performing each job. Hence, an observer of network traffic does not learn anything about π. For our solution that means that the observer cannot trace which records of X_{R} correspond to records in X_{M}.

We now consider the network traffic of the Shuffle-in-the-Middle solution in the Correlation Hiding game as described in Definition 2 for both datasets D^0 and D^1 picked by the adversary. Once one fixes the size of the dataset, M and R and the output size of every map function M (Requirement 1), the traffic produced by the Mappers is deterministic and protected by a semantically secure encryption scheme. Similarly, recall that the traffic produced by the Melbourne Shuffle is deterministic once these values are fixed.

The only possible difference in the observations between D^0 and D^1 are the $r = f(k)$ indexes in $(r, [x_j])$ records in X_{R}. Requirement 2 of the game guarantees there exists a σ permutation on the keys such that, for all keys $k \in K$, we have $\sum_m \mathbb{A}^0[m,k] = \sum_i \mathbb{A}^1[m, \sigma(k)]$. Relying on the pseudo-randomness of f, the probability that a randomly picked function equals either f or $\sigma \circ f$ is the same. Consequently, the count of reducer indexes in X_{R} is equally distributed for both datasets.

We are left to argue that the difference in locations of reducer indexes reveals nothing about the underlying records and hence does not help the adversary win the game. Relying on the pseudo-randomness of π, this follows trivially from the fact that the Melbourne Shuffle can be seen as picking a random permutation π to shuffle the data and this permutation is thus independent of the dataset.

7. SHUFFLE & BALANCE SOLUTION

The Shuffle-In-The-Middle described in §6 prevents the adversary from observing the volume of data exchanged between individual Mappers and $\mathsf{Reducer}$s (the matrix $\mathbb{A}$). However, the adversary still observes the number of records each Mapper produces and the distribution of encrypted keys.

Our second solution meets our stronger Definition 3 by evenly distributing the intermediate traffic sent from each mapper to each reducer. It preserves the data parallelism of MapReduce, and may even improve its performance by facilitating resource allocation and scheduling. But it requires a

more precise load-balancing than what is typically achieved by MapReduce implementations.

7.1 Overview

We are seeking solutions that fit into existing MapReduce implementations, which precludes a complete redesign of mappers with better balancing properties. Instead, we use preliminary MapReduce jobs to plan how to balance (and pad) the intermediate key-value pairs for the 'main' job. We split this pre-processing into offline and online jobs. (In Figure 8 we give an illustration of the Shuffle & Balance solution.)

The offline stage runs on the input data (once for all jobs) and randomizes the ordering of the input records. This erases any correlations between the ordering of inputs and the values of their attributes (as those exploited in §4.3.2), and ensures that all mappers produce the same distribution of key-value pairs. This stage may be implemented by a shuffle (§6.1) or, pragmatically, as the user uploads her input data to the cloud.

The online stage is job specific; it samples the input data to collect statistics about the keys produced by mappers, in order to balance them evenly between reducers and to estimate (with high probability) an upper bound on the number of key-value pairs sent by each mapper to each reducer. Let M and R be the map and reduce functions of the job and R its number of reducers. For the following discussion, recall that we refer to real keys produced by M as a "key" or a "real key", while we refer to one of R reducers using "reducer index" notation.

1. We first run M and R on a fixed sample of the randomized input. We collect statistics on its intermediate keys: a list of pairs $(k_1, f_1), (k_2, f_2), \ldots, (k_l, f_l)$ where k_i ranges over the sampled keys and f_i is the fraction of key-value pairs in the sample with key k_i. This list enables us to estimate the distribution of keys for the whole input dataset, notably its most popular key.

   We also determine the total number of key-value pairs returned for the sample size and the constant output size ℓ of R.

   This task is easily expressed as a small job whose traffic pattern depends only on the size of the sample. The statistics we collect are reminiscent of those maintained in databases to optimize, for instance, joins on record fields; they may similarly be cached and shared between jobs that map data on the same attributes.

2. We generate a key assignment for the job: a function from all (potential) intermediate keys to $1..R$, intended to balance $\mathbb{A}[m, r]$ by grouping keys so that every reducer gets roughly the same number of records, as detailed in §7.3.

   We also estimate a safe upper bound on the fraction of traffic sent from any mapper to reducer r and an upper bound on the number of different keys assigned per reducer. Our algorithms are detailed in §7.3.

The 'main' job then runs, essentially unchanged, except that (1) every mapper uses the resulting assignment to map keys to reducers, instead of the random, uniform intermediate partition in the base solution; (2) every mapper finally sends

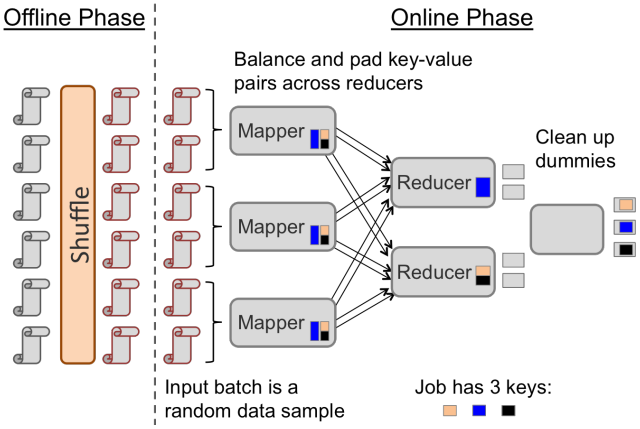

Figure 8: Overview of the Shuffle & Balance solution (see §7) where all data elements are encrypted and Mapper and Reducer code is executed inside of the secure region. Colored boxes represent three different keys; the size of each box represents the number of key-value pairs with this key.

dummy key-value pairs to every reducer, up to its upper bound estimate; and (3) every reducer silently discards intermediate dummies, and pads its output with dummy values up to its upper bound estimate.

As an optional, post-processing stage, we may use a shuffle on the reducer outputs, to discard their dummy values and erase any correlation between key frequencies and output ordering, or pragmatically leave this simple task to the user.

Definition 3 leaks the exact values of $\max_k(\sum_m \mathbb{A}[m, k])$, $|X|$, and the number of keys in the dataset, whereas our solution leaks these values as observed in a random sample. How do our estimates relate to the exact values for the dataset? An estimate, with certain confidence, yields a range in which the exact value lies. Since our sample is chosen at random, the estimates depend on these three values and our target confidence level, but also on the actual records in the sample. To meet the definition, we formally require that statistics to be collected on the whole input dataset. We note, however, that for large shuffled datasets, even relatively small samples already provide excellent estimates.

7.2 Parameter Estimation

Sampling phase enables us to estimate parameters of the job that we are willing to reveal as per Definition 3 and which are not known in advance. These parameters are the number of key-value pairs $|X|$, the frequency of the most popular key in X (the maximal value of the fractions f_i above, written α in the following) and the number of keys $|K|$.

Estimating $|X|$ and α. The relation between $|D|$ and $|X|$ can be described as $|X| = \sigma \mathsf{m} |D|$ where m is the maximum number of pairs that M can return and σ is the fraction that is actually returned on this dataset. Then, $|X_{|k}| = \alpha |X|$, where k is the most popular key. Using a sample of D we can estimate σ as $\hat{\sigma}$ and α as $\hat{\alpha}$. We note that for jobs that always return one record (i.e., $|X| = |D|$ and $\sigma = 1$) we only need to estimate α.

In order to estimate σ, we model number of intermediate key-value pairs produced by a mapper on a sample of size s

as a Binomial random variable with parameters $\mathsf{m}s$ and σ. Let X^s be intermediate key-value pairs from the sample of size s and $|X^s|$ be the number of pairs. From the sample we get the empirical fraction $\bar{\sigma} = |X^s|/(\mathsf{m}s)$ which is our guess at σ. Intuitively, the larger the sample, the better is our guess of σ. Parameter estimation using the Chernoff Bound [20, Chapter 4.2] gives us exactly this relation between the sample size, an upper bound on a parameter and the probability that the parameter exceeds this bound. In particular, if

$$s \geqslant \frac{2}{\mathsf{m}\epsilon^2} \ln \frac{1}{\delta}$$

then $\sigma \geqslant \bar{\sigma} + \epsilon$ with probability at most δ. Once we choose parameters δ and ϵ, we choose a sample of size s, compute its $\bar{\sigma}$ and set $\hat{\sigma}$ to $\bar{\sigma} + \epsilon$.

The process for estimating α can be modeled as a Binomial random variable with parameters $|X^s|$ and α. Again using the Chernoff bound, if

$$|X^s| \geqslant \frac{2}{\epsilon'^2} \ln \frac{1}{\delta'}$$

then $\alpha \geqslant \bar{\alpha} + \epsilon'$ with probability at most δ' where $\bar{\alpha}$ is the fraction of key-value pairs with the most popular key in X^s. Once we fix ϵ' and δ' we can set $\hat{\alpha} = \bar{\alpha} + \epsilon'$. If $|X^s|$ is small, as a result, our error estimate ϵ' may be too high. For example, even if s is large enough for a good estimate of σ, the resulting $|X^s|$ may not be enough for estimating α well (e.g., consider a map function with a very restrictive filter). In this case, we may need to increase s to get a higher number of sampled intermediate key-value pairs.

We note that in both cases we chose to overestimate our parameters. Overestimation leads to higher padding cost but lower probability of failure of the mapper protocol.

Estimating $|K|$. Given the output of a map function on a sample we wish to estimate the number of distinct keys in the output the MapReduce job will produce on the whole dataset D. To this end, we use the estimation technique from [9] to set our estimate to an upper bound on $|K|$:

$$\sum_{i>1} e_i + \frac{|D|}{|X^s|} e_1$$

where e_i is the number of distinct keys in X^s that occur exactly i times in X^s. We note that better estimates can be achieved if prior information on distribution of K is known in advance.

7.3 Mapping vs. Bin Packing

In this section, we explain how we use the statistics collected in the online stage to produce a secure balanced assignment. In particular, we explain how to allocate sufficient bandwidth between mappers and reducers to fit *any* distribution with the same α. (Recall that Definition 3 enables us to leak only the maximal frequency, not the detailed key distribution.) The algorithms are explained using α, while in the instantiation of these algorithms we use our estimate $\hat{\alpha}$ where $\hat{\alpha} \geqslant \alpha$ with carefully chosen parameters (Section 7.2).

Our problem, at heart, is an instance of bin packing, so we first review bin packing basics before giving our algorithm.

Bin packing. The (offline) bin packing instance is expressed in terms of a fixed bin capacity c and a list of items, each with a weight at most c. (In our case, a key is viewed as an item and its frequency as its weight.) The goal of bin packing algorithms is to minimize the number of bins N needed to allocate all items without an overflow. Since the offline bin-packing problem is NP-hard, approximation algorithms, such as First Fit Decreasing (FFD) [10], return both a number of bins and a guarantee on how far it can be from the optimum in the worst case. The FFD algorithm places items in decreasing weight order, allocating new bins on demand: it places the heaviest item in the first bin, then proceeds with the next item and tries to place it into one of the open bins (i.e., the bins that already have items) without exceeding its capacity. If there is no space left in any open bin, it places the item in a new bin.

Bin Packing, Obliviously. Our problem is more general: given only some maximal item weight α, we must find a bin capacity c and an upper bound on the number of bins N so that FFD packing will succeed on *any* weighted list of items with maximal weight α. That is, N and c depend only on α and not the distribution of the rest of the elements.

In the general case, we choose $N = \frac{2}{\alpha} - 1$ and $c = \alpha$. In the next section we prove that such N and c are enough to pack any sequence of items with maximal weight of α using FFD. Since the weights of all items sum up to 1, $Nc - 1$ is the overhead of the solution in terms of dummy key-value pairs that have to be added to fill in the bins. Hence, the values of N and c above yield an overhead of $1 - \alpha$. We can reduce the overhead above in several special cases; for example:

- If $\alpha \ll \frac{1}{R}$, then we may pick $N = R$ and $c = \frac{1}{R} + \alpha$, which yields a low overhead of αR.
- If $\alpha \geqslant \frac{1}{2}$, we have at least one very large reducer of capacity α and everything else will fit into a second, smaller reducer of capacity $1 - \alpha$. In this special case, there is no actual need for FFD.

The general and special cases of fixing N and c ensure that, from a traffic analysis viewpoint, the number and capacities of bins (which, as described next, entirely determine the matrix $\mathbb{A}$ for the job we protect) depend *only* on α.

Bin Packing Keys to Reducers. Once c is fixed, we are ready to bin-pack the distribution of keys we have sampled, and to generate our (secret) assignment for the main job. To this end, we add two dummy keys with weight 0, for the smallest and largest keys (if not already present in the sample). We partition the domain of all intermediate keys into intervals, such that the total weight of all the keys in any given interval is bounded by α. Hence, there is at least one interval that contains one single key, with maximal weight α. The inclusion of dummy keys ensures that assignment is a total function, even on keys that we have not sampled. We then sort these intervals by decreasing weight and run the FFD algorithm on them (assured that at most N bins will be used) to get a mapping between key intervals and bins.

We independently distribute N bins between our R reducers, such that each reducer gets at most $\lceil N/R \rceil$ bins. Hence, some reducers may get less than $\lceil N/R \rceil$ bins, or no bins at all if $N < R$. We denote the capacity level of rth reducer as c_r. Finally, we use this (public) mapping and the (secret) FFD output to produce an assignment that maps each key interval to the reducer of the bin it has been placed into.

7.4 Padding Traffic to Fixed Sizes

Intermediate Key-Value Pairs. We select a level of padding for the traffic sent from each mapper to each reducer based on two considerations: we must hide the actual number of key-value pairs received by each reducer—that is, the actual usage of each bin we have allocated—by filling those bins with dummies; and we must over-provision to accommodate (with high probability) for any difference in key distribution between the dataset and the output of each mapper. To this end, the assignment is supplemented with padding target, to be used by every mapper to compute the apparent number of intermediate key-value pairs it must send to every reducer (as a function of the size of its input). In particular, given a capacity level c_r for rth reducer, a mapper with batch size b sends $\mathsf{Padded}(c_r, b)$ key-value pairs to rth reducer. We describe how Padded is computed below.

Given a capacity level c_r and mapper's batch size b, $\mu_{r,b} = c_r \times \sigma \times \mathsf{m} \times b$ is the number of key-value pairs with reducer index r expected to be returned by a mapper on the input size b. (Recall that c_r is expressed as a proportion of the intermediate key-value pairs and $b \times \sigma \times \mathsf{m}$ is the expected number of pairs to be produced from a batch of size b.) Since mapper's batch is only a sample of the dataset, the actual number of key-value pairs with index r, $\mathcal{X}_{r,b}$, may differ from $\mu_{r,b}$. However, since the sample is a random sample of the dataset (due to randomization during the offline phase) it will not "lie too far" from the mean. To this end, $\mathsf{Padded}(c_r, b)$ returns a bound on how many key-value pairs with index r a random sample may contain with probability θ, i.e.,

$$\Pr(\mathcal{X}_{r,b} \geqslant \mathsf{Padded}(c_r, b)) \leqslant \theta.$$

This bound is determined as follows. We treat the number of intermediate records (mapper's output) with index r as a Binomial random variable $\mathcal{X}_{r,b}$ with expected value $\mu_{r,b}$. Then, using the Chernoff bound [20, Chapter 4.2]:

$$\Pr(\mathcal{X}_{r,b} \geqslant (1+\epsilon)\mu_{r,b}) \leqslant \left(\frac{\mathrm{e}^{\epsilon}}{(1+\epsilon)^{(1+\epsilon)}}\right)^{\mu_{r,b}} = \theta.$$

Once the desired failure probability is fixed, we can rearrange the above formula to express ϵ in terms of θ and $\mu_{r,b}$, and then set $\mathsf{Padded}(c_r, b)$ to $(1+\epsilon)\mu_{r,b}$.

So far we considered only the event that a particular mapper does not exceed the padded level with probability more than θ. In the algorithm we wish to bound that *no* mapper sends more than a bounded number of key-value pairs to *any* of the R reducers. We can do so by applying the union bound to $M \times R$ possible events and bound first the probability $\theta \times M \times R$ and only then fix θ.

Interestingly, the resulting matrix of observable intermediate traffic $\mathbb{A}$ for the main job is not necessarily uniform, as mappers may process batches of different sizes and reducers may process different numbers of bins, but this matrix depends only on α (recall that c_r is determined using α), R, and the sizes of the mapper inputs.

Reducer Output. Preventing traffic analysis on the job output is simpler. We set rkey to bound the maximum number of keys that may be allocated to a single reducer given R, $|K|$ and α. We count the maximum number of rare keys that may be assigned to any single bin, assuming that large keys are distributed elsewhere. In particular, we set it to $\mathsf{rkey} = |K| - \lceil 1/\alpha \rceil + 1$. Then, for every bin assigned to a reducer, the reducer output is padded up to $\mathsf{rkey} \times \ell$ where ℓ is the output size of R on a list of values with the same key.

7.5 Analysis

We first prove that N and c we chose in §7.3 for computing $\mathsf{assignment}$ does not lead to an overflow for any distribution with a fixed α. Then we argue that the Shuffle & Balance solution is secure.

LEMMA 2. *FFD uses at most $N = 2/\alpha - 1$ bins, each with capacity $c = \alpha$, when packing any sequence of items with item weight at most α.*

PROOF. Let us compute values of N that lead to an overflow if FFD is used with bin capacity $c = \alpha$. Since we know there is at least one item of weight α we assume that FFD assigns it to the first bin.

Let β be the weight of the item that cannot fit in the first N bins and, hence, $N+1$th bin has to be opened. Then the following constraints should hold:

1. $\beta \leqslant 1 - \alpha - (N-1) \times c'$ where $c' \leqslant c$ is the minimum weight among N open bins. This constraint ensures that the total weight of leftover items (i.e., items not assigned to the first N bins) is at least β.
2. $\alpha - c' < \beta$, since a new bin has to be opened the leftover space in any of the open bins must be less than β.
3. $\beta \leqslant c'$, since FFD orders items in descending order, β has to be less than an item weight placed before.

Let us combine constraints 1 and 2:

$$\begin{aligned} \alpha - c' &< 1 - \alpha - (N-1) \times c' \\ (N-1) \times c' - c' &< 1 - 2\alpha \\ c' &< (1-2\alpha)/(N-2) \end{aligned} \tag{1}$$

Then combining constraints 2 and 3 gives us $\alpha/2 < c'$. Finally we can combine result of Equation (1) and $2\alpha < c'$ to get a bound on N in terms of α:

$$\begin{aligned} \alpha/2 &< (1-2\alpha)/(N-2) \\ N < 2(1-2\alpha)/\alpha + 2 &= (2-4\alpha+2\alpha)/\alpha = \\ &(2-2\alpha)/\alpha = 2/\alpha - 2 \end{aligned}$$

Hence, to avoid overflow we need to set N to a value at least $2/\alpha - 1$. □

LEMMA 3. *Let R be the number of reducers. If $\alpha \ll \frac{1}{R}$, then FFD uses at most $N = R$ bins each of capacity $c = \frac{1}{R} + \alpha$, when packing any sequence of items with item weight at most α.*

PROOF. Assume there is an item with weight β that does not fit in the first N bins. Let c' be the minimum bin weight across all N open bins. Then it must be the case that (1) $\beta \leqslant 1 - c'R$, since there is at least one item with a non-zero weight β left to pack, and (2) $c' + \beta > 1/R + \alpha$, since there is no room for the item with weight β among open bins. From (2) we get $c' > 1/R$ since $\alpha - \beta \geqslant 0$. Then $1 - c'R < 0$ and using (1) $\beta < 0$. Hence, there cannot be a leftover item with positive weight. □

THEOREM 4. *If $[\cdot]$ is a semantically secure encryption scheme and the permutations π and π' used in pre- and (optional) post-processing are pseudo-random, then the Shuffle & Balance solution is strongly hiding (Definition 3).*

We show that if mappers do not fail (i.e., do not send more traffic that what they padded for), then the traffic generated by Shuffle & Balance depends only on constraints on the two databases that the adversary is allowed to pick. We then show that the failure probability — the probability that a mapper has to output more traffic than predicted – is low. Our proof is by construction.

In Lemmas 2 and 3 we showed that our choices for the number of bins N and their capacity c are sufficient for bin-packing any sequence with maximal key weight α. This ensures that bin packing always produces a valid key-reducer assignment. Furthermore, N and c depend only on α.

Given key-reducer assignment, a mapper reads a batch of size b and sends $\mathsf{Padded}(c_r, b)$ key-value pairs to rth reducer. The value $\mathsf{Padded}(c_r, b)$ depends on c_r, b, m and σ and the failure probability θ. The values b, m and θ are public, while c_r depends on c, N (both depend on α) and public parameter R and σ depends on $|X|$. Since α and $|X|$ are constraints of Definition 3, traffic sent from mappers to reducers is not revealing anything about the data in D.

A mapper can fail with probability θ if there is a reducer r s.t. the mapper needs to send it more traffic than allocated by Padded. The choice of θ determines the probability of a random sample diverging from the padded schedule it has to fit to (we treat mapper's batch as a random sample due to the offline shuffle of D). Hence, it can be set arbitrarily low by increasing the padding level as returned by Padded.

Now consider reducers. Let b_m be the batch size of mth mapper. A reducer with index r reads $\sum_{1 \leqslant m \leqslant M} \mathsf{Padded}(c_r, b_m)$ key-values pairs and returns output of size $\mathsf{rkey} \times \ell$ for every bin assigned to it. Hence, the output size of the reducer depends only on ℓ, N, $|K|$ and α.

Observing the optional cleanup phase (relying on π') or the size of the output (if the user performs it herself) shows the reduction from size $\mathsf{rkey} \times \ell \times N$ to $|O|$, which is again a restriction in the adversarial game.

Our proof assumes that the parameter α, the number of key-value pairs $|X|$ and the number of keys $|K|$ are precise. In our instantiation we use estimates of these values from a random sample which, with high probability, give us upper bounds on each of these parameters (Section 7.2). Using upper bounds ensures that our algorithms do no fail with high probability, while revealing values of these parameters in a chosen random sample.

Our solution hides any distribution of keys with maximum frequency α, but it does reveal α. This is justified, because at least one reducer must process all the key-value pairs for the most frequent key. However, this can be mitigated (notably when $\alpha \ll \frac{1}{R}$) by increasing α before computing c.

8. EVALUATION

We have evaluated our framework using a local Hadoop cluster of 8 workstations connected with a Netgear GS108 1Gbps switch. Our nodes ran under Microsoft Windows Server 2012 R2 64-Bit on workstations with a 2.9 GHz Intel Core i5-4570 (Haswell) processor, 8 GB of RAM, and a 250 GB Samsung 840 Evo SSD. We implemented our solutions in Java for experiments on plain Hadoop and in C++ for VC3 experiments, which use AES-GCM encryption implemented with AES-NI instructions and a software emulator for SGX.

We perform experiments on the two datasets presented in §4.3: a census data sample (900 MB) and the New York taxi rides (2.5 GB per month). We perform two types of jobs: aggregate and aggregate-filter, where the latter is an aggregate over records filtered by some parameter. The baseline run times correspond to the initial job on Hadoop without protection. The reported numbers are averaged over 5 runs.

Table 1: Run times for Shuffle-in-the-Middle (S).

DataSet/Job	Base	Run time (Shuffle)
Census/`Age` grouped	20	91 (25)
Taxi Jan/`PassenN`	39	122 (38)
Taxi Jan/`PickupD`	40	131 (43)

Table 2: Run times for Shuffle & Balance (S) where `PassenN-1` aggregates passenger counts without the most popular key.

`Attrib`	Taxi Jan (2.5 GB)			Taxi Jan-Apr (10 GB)		
	α	$\lvert K\rvert$	Run time (×Base)	α	$\lvert K\rvert$	Run time (×Base)
`PassenN`	.71	6	45 (1.01)	.71	6	61 (0.93)
`PickUpD`	.038	31	48 (1.12)	.01	120	78 (1.21)
`PassenN-1`	.47	5	43 (1.09)	.45	5	55 (1.06)

The run times for the Shuffle-in-the-Middle solution in Java are summarized in Table 1. This experiment involves 4 MapReduce jobs: for mapping, shuffling twice, and reducing. Hence, no batching and parallelization is enabled and all jobs are treated sequentially. In contrast, in the base execution of this job, grouping of intermediate pairs by keys starts as soon as mappers produce some output. Starting the shuffle as soon as the map job outputs its first key-value pairs may reduce the I/O overhead (similarly starting sorting keys before the last shuffle finishes). Shuffling costs highly depend on the size of the data; for passenger count a value is simply a number vs. a date for pickup date job.

Next we measure the run times of our Java implementation for the Shuffle & Balance solution on several attributesof the Taxi dataset of size 2.5 GB (Jan) and 10 GB (Jan-Apr) with $R = 15$. The results for the online phase for a randomized taxi dataset are presented in Table 2. For each job we show the frequency of the most popular key and the number of keys. Shuffle & Balance is more efficient than our first solution, assuming one can run the jobs on shuffled data: performance overhead increases on average by 7%. In one example, our solution even outperforms the baseline. This is due to the smaller number of key-value pairs returned to the system by the mappers and, hence, lower overhead for the framework to group them together. Recall that our solution pre-groups values with the same key during bin packing, thereby using a smaller number of keys but larger values.

Finally, we implemented the Melbourne Shuffle as two runs of the MapReduce job presented in §6.1, both in Java and in C++ for VC3. Table 3 gives the run times for the offline phase of our Shuffle & Balance solution on two datasets. Recall that this phase is run once per dataset, and not once per every job.

Discussion. From the experiments we can see that the Shuffle & Balance solution (§7) has a much better performance than the Shuffle-in-the-Middle solution (§6). The additional padding produced by the former does not affect the performance as much as two additional MapReduce jobs for

Table 3: Run times for Melbourne Shuffle in offline phase (in seconds).

DataSet	Java	VC3
Census	76	122
Taxi Rides Jan	160	153

performing the shuffle do. The Shuffle-in-the-Middle may still be of interest if one cannot perform the offline phase (shuffling) of the Shuffle & Balance solution before running MapReduce jobs.

We also note that if hiding key distribution is not required, the Shuffle & Balance solution can perform a "lighter" version of the online phase and reduce the amount of padding. In this case, one pads only to hide the difference between mappers' inputs without executing bin packing.

9. RELATED WORK

Several systems protect confidentiality of data in the cloud. CryptDB [24] and MrCrypt [29] use partial homomorphic encryption to run some computations on encrypted data; they neither protect confidentiality of code, nor guarantee the integrity of results. On the upside, they do not use trusted hardware. TrustedDB [4], Cipherbase [3], and Monomi [31] use trusted hardware to process database queries over encrypted data, but do not protect the confidentiality and integrity of all code and data. Haven [6] can run databases on a single machine.

All systems above are vulnerable to side-channel attacks. For example, Xu *et al.* [33] show how side-channel attacks can be exploited in systems such as Haven where an untrusted operating system controls page faults. We also refer the reader to [33] for an overview on side-channel attacks.

Several security-enhanced MapReduce systems have been proposed. Airavat [25] defends against possibly malicious map function implementations using differential privacy. SecureMR [32] is an integrity enhancement for MapReduce that relies on redundant computations. Ko *et al.* propose a hybrid security model for MapReduce where sensitive data is handled in a private cloud while non-sensitive processing is outsourced to a public cloud provider [17]. PRISM [7] is a privacy-preserving word search scheme for MapReduce that utilizes private information retrieval methods.

Nayak *et al.* [21] propose a programming model for secure parallel processing of data represented as a graph using oblivious sorting and garbled circuits. Goodrich and Mitzenmacher [15] describe a simulation of MapReduce that resembles a sequential version of our Shuffle-in-the-Middle solution using a sorting network instead of a shuffle to protect against traffic analysis. This method can be parallelized using a step from §6 where oblivious sorting uses a reducer number (computed as a pseudo-random function of each key) to sort key-value pairs and returns reducer keys in the clear. In independent parallel work, Dinh *et al.* [11] also consider securing MapReduce using a mix network to shuffle traffic between mappers and reducers. The three solutions above rely either on oblivious sort or mix network, and thus incur a logarithmic depth overhead. In comparison, our use of the Melbourne Shuffle in our first solution, Shuffle in the Middle, requires only two additional map-reduce jobs, and incurs a constant depth overhead. Besides, our second solution, Shuffle & Balance, dominates the first, even with the Melbourne Shuffle: the security guarantees are stronger (it hides key distributions, mapper output sizes, and reducer input sizes) and incurs a much smaller overhead (§8).

Oblivious RAM (ORAM) [13] is a general, well-studied technique for protecting computations against memory traffic analysis. Though ORAMs are becoming more efficient, they incur a logarithmic overhead on every access and do not hide I/O volume. Moreover most ORAMs, except for the recent theoretical work by Boyle *et al.* [8] with polylogarithmic access overhead, are intrinsically sequential.

10. REFERENCES

[1] M. Ajtai, J. Komlós, and E. Szemerédi. An $O(n \log n)$ sorting network. In *Proceedings of the Fifteenth Annual ACM Symposium on Theory of Computing*, STOC '83, pages 1–9, New York, NY, USA, 1983. ACM.

[2] Apache Software Foundation. Hadoop. `http://wiki.apache.org/hadoop/`, 15/05/15.

[3] A. Arasu, S. Blanas, K. Eguro, R. Kaushik, D. Kossmann, R. Ramamurthy, and R. Venkatesan. Orthogonal security with Cipherbase. In *Conference on Innovative Data Systems Research (CIDR)*, 2013.

[4] S. Bajaj and R. Sion. TrustedDB: A trusted hardware-based database with privacy and data confidentiality. *Knowledge and Data Engineering, IEEE Transactions on*, 26(3):752–765, March 2014.

[5] K. E. Batcher. Sorting networks and their applications. In *Proc. 1968 Spring Joint Computer Conf.*, pages 307–314. AFIPS Press, 1968.

[6] A. Baumann, M. Peinado, and G. Hunt. Shielding applications from an untrusted cloud with haven. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.

[7] E.-O. Blass, R. Di Pietro, R. Molva, and M. Önen. Prism—privacy-preserving search in MapReduce. In S. Fischer-Hübner and M. Wright, editors, *Privacy Enhancing Technologies*, volume 7384 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2012.

[8] E. Boyle, K.-M. Chung, and R. Pass. Oblivious parallel RAM. Cryptology ePrint Archive, Report 2014/594, 2014. `http://eprint.iacr.org/`.

[9] M. Charikar, S. Chaudhuri, R. Motwani, and V. Narasayya. Towards estimation error guarantees for distinct values. In *Proceedings of the Nineteenth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '00, pages 268–279, New York, NY, USA, 2000. ACM.

[10] E. Coffman Jr., J. Csirik, G. Galambosa, S. Martello, and D. Vigo. Bin packing approximation algorithms: Survey and classification. In P. M. Pardalos, D.-Z. Du, and R. L. Graham, editors, *Handbook of Combinatorial Optimization*, pages 455–531. Springer New York, 2013.

[11] A. Dinh, P. Saxena, C. Ee-chien, Z. Chunwang, and O. B. Chin. M2r: Enabling stronger privacy in mapreduce computation. In *24th USENIX Security Symposium (USENIX Security 15)*, Washington, D.C., Aug. 2015. USENIX Association.

[12] C. Gentry. Fully homomorphic encryption using ideal lattices. In *Proceedings of the Forty-first Annual ACM*

Symposium on Theory of Computing, STOC '09, pages 169–178, New York, NY, USA, 2009. ACM.

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

[14] S. Goldwasser and S. Micali. Probabilistic encryption. *J. Comput. Syst. Sci.*, 28(2):270–299, 1984.

[15] M. Goodrich and M. Mitzenmacher. Privacy-preserving access of outsourced data via oblivious RAM simulation. In L. Aceto, M. Henzinger, and J. Sgall, editors, *International Colloquium on Automata, Languages and Programming (ICALP)*, volume 6756 of *Lecture Notes in Computer Science*, pages 576–587. Springer Berlin Heidelberg, 2011.

[16] J. Katz and Y. Lindell. *Introduction to Modern Cryptography*. Chapman and Hall/CRC Press, 2007.

[17] S. Y. Ko, K. Jeon, and R. Morales. The Hybrex model for confidentiality and privacy in cloud computing. In *USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)*, 2011.

[18] M. Lichman. UCI machine learning repository, 2013.

[19] F. Mckeen, I. Alexandrovich, A. Berenzon, C. Rozas, H. Shafi, V. Shanbhogue, and U. Savagaonkar. Innovative instructions and software model for isolated execution. In *Workshop on Hardware and Architectural Support for Security and Privacy (HASP)*, 2013.

[20] M. Mitzenmacher and E. Upfal. *Probability and Computing: Randomized Algorithms and Probabilistic Analysis*. Cambridge University Press, New York, NY, USA, 2005.

[21] K. Nayak, X. S. Wang, S. Ioannidis, U. Weinsberg, N. Taft, and E. Shi. GraphSC: Parallel secure computation made easy. In *IEEE Symposium on Security and Privacy*, 2015.

[22] O. Ohrimenko, M. Goodrich, R. Tamassia, and E. Upfal. The melbourne shuffle: Improving oblivious storage in the cloud. In J. Esparza, P. Fraigniaud, T. Husfeldt, and E. Koutsoupias, editors, *International Colloquium on Automata, Languages and Programming (ICALP)*, volume 8573 of *Lecture Notes in Computer Science*, pages 556–567. Springer Berlin Heidelberg, 2014.

[23] V. Pandurangan. On taxis and rainbows: Lessons from NYC's improperly anonymized taxi logs, 2014.

[24] R. A. Popa, C. M. S. Redfield, N. Zeldovich, and H. Balakrishnan. CryptDB: Protecting confidentiality with encrypted query processing. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 85–100, New York, NY, USA, 2011. ACM.

[25] I. Roy, S. T. Setty, A. Kilzer, V. Shmatikov, and E. Witchel. Airavat: Security and privacy for MapReduce. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2010.

[26] F. Schuster, M. Costa, C. Fournet, C. Gkantsidis, M. Peinado, G. Mainar-Ruiz, and M. Russinovich. VC3: Trustworthy data analytics in the cloud using SGX. In *IEEE Symposium on Security and Privacy*, 2015.

[27] L. Sweeney. Simple demographics often identify people uniquely. Carnegie Mellon University, Data Privacy Working Paper 3, 2000.

[28] NYC taxi trips. `www.andresmh.com/nyctaxitrips/`, 16/05/15.

[29] S. D. Tetali, M. Lesani, R. Majumdar, and T. Millstein. MrCrypt: Static analysis for secure cloud computations. *SIGPLAN Not.*, 48(10):271–286, Oct. 2013.

[30] A. Tockar. Riding with the stars: Passenger privacy in the NYC taxicab dataset, 2014.

[31] S. Tu, M. F. Kaashoek, S. Madden, and N. Zeldovich. Processing analytical queries over encrypted data. *Proc. VLDB Endow.*, 6(5):289–300, Mar. 2013.

[32] W. Wei, J. Du, T. Yu, and X. Gu. SecureMR: A service integrity assurance framework for mapreduce. In *Proceedings of the 2009 Annual Computer Security Applications Conference*, ACSAC '09, pages 73–82, Washington, DC, USA, 2009. IEEE Computer Society.

[33] Y. Xu, W. Cui, and M. Peinado. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *IEEE Symposium on Security and Privacy*, 2015.