M^{2}R: Enabling Stronger Privacy in MapReduce Computation

Tien Tuan Anh Dinh, Prateek Saxena, Ee-Chien Chang, Beng Chin Ooi, Chunwang Zhang
School of Computing, National University of Singapore
ug93tad@gmail.com, prateeks@comp.nus.edu.sg, changec@comp.nus.edu.sg
ooibc@comp.nus.edu.sg, zhangchunwang@gmail.com

Abstract

New big-data analysis platforms can enable distributed computation on encrypted data by utilizing trusted computing primitives available in commodity server hardware. We study techniques for ensuring privacy-preserving computation in the popular MapReduce framework. In this paper, we first show that protecting only individual units of distributed computation (e.g. map and reduce units), as proposed in recent works, leaves several important channels of information leakage exposed to the adversary. Next, we analyze a variety of design choices in achieving a stronger notion of private execution that is the analogue of using a distributed oblivious-RAM (ORAM) across the platform. We develop a simple solution which avoids using the expensive ORAM construction, and incurs only an additive logarithmic factor of overhead to the latency. We implement our solution in a system called M^{2}R, which enhances an existing Hadoop implementation, and evaluate it on seven standard MapReduce benchmarks. We show that it is easy to port most existing applications to M^{2}R by changing fewer than 43 lines of code. M^{2}R adds fewer than 500 lines of code to the TCB, which is less than 0.16% of the Hadoop codebase. M^{2}R offers a factor of 1.3× to 44.6× lower overhead than extensions of previous solutions with equivalent privacy. M^{2}R adds a total of 17% to 130% overhead over the insecure baseline solution that ignores the leakage channels M^{2}R addresses.

1 Introduction

The threat of data theft in public and private clouds from insiders (e.g. curious administrators) is a serious concern. Encrypting data on the cloud storage is one standard technique which allows users to protect their sensitive data from such insider threats. However, once the data is encrypted, enabling computation on it poses a significant challenge. To enable privacy-preserving computation, a range of security primitives have surfaced recently, including trusted computing support for hardware-isolated computation [2, 5, 38, 40] as well as purely cryptographic techniques [20,21,47]. These primitives show promising ways for running computation securely on a single device running an untrusted software stack. For instance, *trusted computing* primitives can isolate units of computation on an untrusted cloud server. In this approach, the hardware provides a confidential and integrity-protected execution environment to which encryption keys can be made available for decrypting the data before computing on it. Previous works have successfully demonstrated how to securely execute a unit of user-defined computation on an untrusted cloud node, using support from hardware primitives available in commodity CPUs [8, 14, 38, 39, 49] .

In this paper, we study the problem of enabling privacy-preserving *distributed computation* on an untrusted cloud. A sensitive distributed computation task comprises many units of computation which are scheduled to run on a multi-node cluster (or cloud). The input and output data between units of computation are sent over channels controlled by the cloud provisioning system, which may be compromised. We assume that each computation node in the cluster is equipped with a CPU that supports trusted computing primitives (for example, TPMs or Intel SGX). Our goal is to enable a privacy-preserving execution of a distributed computation task. Consequently, we focus on designing privacy in the popular MapReduce framework [17]. However, our techniques can be applied to other distributed dataflow frameworks such as Spark [62], Dryad [26], and epiC [27].

Problem. A MapReduce computation consists of two types of units of computation, namely *map* and *reduce*, each of which takes key-value tuples as input. The MapReduce provisioning platform, for example Hadoop [1], is responsible for scheduling the map/reduce operations for the execution in a cluster and for providing a data channel between them [31]. We aim to achieve a strong level of security in the distributed execution of a MapReduce task (or job) — that is, the adversary learns nothing beyond the execution time and the number of input and output of each computation unit. If we view each unit of computation as one atomic operation of a larger distributed program, the execution can be thought of as running a set of operations on data values passed

via a data channel (or a global "RAM") under the adversary's control. That is, our definition of privacy is analogous to the strong level of privacy offered by the well-known oblivious RAM protocol in the monolithic processor case [22].

We assume that the MapReduce provisioning platform is compromised, say running malware on all nodes in the cluster. Our starting point in developing a defense is a baseline system which runs each unit of computation (map/reduce instance) in a hardware-isolated process, as proposed in recent systems [49, 59]. Inputs and outputs of each computation unit are encrypted, thus the adversary observes only encrypted data. While this baseline offers a good starting point, merely encrypting data in-transit between units of computation is not sufficient (see Section 3). For instance, the adversary can observe the pattern of data reads/writes between units. As another example, the adversary can learn the synchronization between map and reduce units due to the scheduling structure of the provisioning platform. Further, the adversary has the ability to duplicate computation, or tamper with the routing of encrypted data to observe variations in the execution of the program.

Challenges. There are several challenges in building a practical system that achieves our model of privacy. First, to execute map or reduce operations on a single computation node, one could run all computation units — including the entire MapReduce platform — in an execution environment that is protected by use of existing trusted computing primitives. However, such a solution would entail little trust given the large TCB, besides being unwieldy to implement. For instance, a standard implementation of the Hadoop stack is over 190K lines of code. The scope of exploit from vulnerabilities in such a TCB is large. Therefore, the first challenge is to enable practical privacy by minimizing the increase in platform TCB and without requiring any algorithmic changes to the original application.

The second challenge is in balancing the needs of privacy and performance. Addressing the leakage channels discussed above using generic methods easily yields a solution with poor practical efficiency. For instance, hiding data read/write patterns between specific map and reduce operations could be achieved by a generic oblivious RAM (ORAM) solution [22, 55]. However, such a solution would introduce a slowdown proportional to polylog in the size of the intermediate data exchange, which could degrade performance by over 100× when gigabytes of data are processed.

Our Approach. We make two observations that enable us to achieve our model of privacy in a MapReduce implementation. First, on a single node, most of the MapReduce codebase can stay outside of the TCB (i.e. code performing I/O and scheduling related tasks). Thus, we design four new components that integrate readily to the existing MapReduce infrastructure. These components which amount to fewer than 500 lines of code are the only pieces of trusted logic that need to be in the TCB, and are run in a protected environment on each computation node. Second, MapReduce computation (and computation in distributed dataflow frameworks in general) has a specific structure of data exchange and execution between map and reduce operations; that is, the map writes the data completely before it is consumed by the reduce. Exploiting this structure, we design a component called *secure shuffler* which achieves the desired security but is much less expensive than a generic ORAM solution, adding only a $O(\log N)$ additive term to the latency, where N is the size of the data.

Results. We have implemented a system called $\mathtt{M}^2\mathtt{R}$ based on Hadoop [1]. We ported 7 applications from a popular big-data benchmarks [25] and evaluated them on a cluster. The results confirm three findings. First, porting MapReduce jobs to $\mathtt{M}^2\mathtt{R}$ requires small development effort: changing less than 45 lines of code. Second, our solution offers a factor of 1.3× to 44.6× (median 11.2×) reduction in overhead compared to the existing solutions with equivalent privacy, and a total of $17\% - 130\%$ of overhead over the baseline solution which protects against none of the attacks we focus on in this paper. Our overhead is moderately high, but $\mathtt{M}^2\mathtt{R}$ has high compatibility and is usable with high-sensitivity big data analysis tasks (e.g. in medical, social or financial data analytics). Third, the design is scalable and adds a TCB of less than 0.16% of the original Hadoop codebase.

Contributions. In summary, our work makes three key contributions:

- *Privacy-preserving distributed computation*. We define a new pragmatic level of privacy which can be achieved in the MapReduce framework requiring no algorithmic restructuring of applications.
- *Attacks*. We show that merely encrypting data in enclaved execution (with hardware primitives) is insecure, leading to significant privacy loss.
- *Practical Design*. We design a simple, non-intrusive architectural change to MapReduce. We implement it in a real Hadoop implementation and benchmark its performance cost for privacy-sensitive applications.

2 The Problem

Our goal is to enable privacy-preserving computation for distributed dataflow frameworks. Our current design and

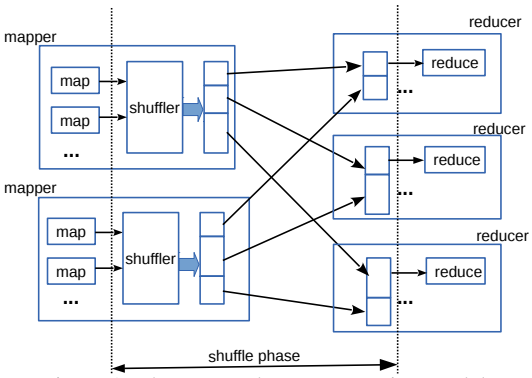

Figure 1: The MapReduce computation model.

implementation are specific to MapReduce framework, the computation structure of which is nevertheless similar to other distributed dataflow engines [26, 27, 62], differing mainly in supported operations.

Background on MapReduce. The MapReduce language enforces a strict structure: the computation task is split into map and reduce operations. Each instance of a map or reduce, called a *computation unit* (or unit), takes a list of key-value tuples[1]. A MapReduce task consists of sequential phases of map and reduce operations. Once the map step is finished, the intermediate tuples are grouped by their key-components. This process of grouping is known as *shuffling*. All tuples belonging to one group are processed by a reduce instance which expects to receive tuples sorted by their key-component. Outputs of the reduce step can be used as inputs for the map step in the next phase, creating a chained MapReduce task. Figure 1 shows the dataflow from the map to the reduce operations via the shuffling step. In the actual implementation, the provisioning of all map units on one cluster node is locally handled by a *mapper* process, and similarly, by a *reducer* process for reduce units.

2.1 Threat Model

The adversary is a malicious insider in the cloud, aiming to subvert the confidentiality of the client's computation running on the MapReduce platform. We assume that the adversary has complete access to the network and storage back-end of the infrastructure and can tamper with the persistent storage or network traffic. For each computation node in the cluster, we assume that the adversary can corrupt the entire software stack, say by installing malware.

We consider an adversary that perpetrates both passive and active attacks. A passive or *honest-but-curious* attacker passively observes the computation session, behaving honestly in relaying data between computation units, but aims to infer sensitive information from the observed data. This is a pragmatic model which includes adversaries that observe data backed up periodically on disk for archival, or have access to performance monitoring interfaces. An active or *malicious* attacker (e.g. an installed malware) can deviate arbitrarily from the expected behavior and tamper with any data under its control. Our work considers both such attacks.

There are at least two direct attacks that an adversary can mount on a MapReduce computation session. First, the adversary can observe data passing between computation units. If the data is left unencrypted, this leads to a direct breach in confidentiality. Second, the adversary can subvert the computation of each map/reduce instance by tampering with its execution. To address these basic threats, we start with a baseline system described below.

Baseline System. We consider the *baseline system* in which each computation unit is hardware-isolated and executed privately. We assume that the baseline system guarantees that the program can only be invoked on its entire input dataset, or else it aborts in its first map phase. Data blocks entering and exiting a computation unit are encrypted with authenticated encryption, and all side-channels from each computation unit are assumed to be masked [51]. Intermediate data is decrypted only in a hardware-attested computation unit, which has limited memory to securely process up to T inputs tuples. Systems achieving this baseline have been previously proposed, based on differing underlying hardware mechanisms. VC^3 is a recent system built on Intel SGX [49].

Note that in this baseline system, the MapReduce provisioning platform is responsible for invoking various trusted units of computation in hardware-isolated processes, passing encrypted data between them. In Section 3, we explain why this baseline system leaks significant information, and subsequently define a stronger privacy objective.

2.2 Problem Definition

Ideally, the distributed execution of the MapReduce program should leak nothing to the adversary, except the total size of the input, total size of the output and the running time. The aforementioned baseline system fails to achieve the ideal privacy. It leaks two types of information: (a) the input and output size, and processing time of individual computation unit, and (b) dataflow among the computation units.

We stress that the leakage of (b) is significant in many applications since it reveals relationships among the input. For instance, in the well-known example of computing Pagerank scores for an encrypted graph [44], flows from a computation unit to another correspond to edges

[1]To avoid confusion of the tuple key with cryptographic key, we refer to the first component in the tuple as *key-component*.

in the input graph. Hence, leaking the dataflow essentially reveals the whole graph edge-structure!

Techniques for hiding or reducing the leakage in (a) by padding the input/output size and introducing timing delays are known [35, 41]. Such measures can often require algorithmic redesign of the application [9] or use of specialized programming languages or hardware [33,34], and can lead to large overheads for applications where the worst case running time is significantly larger than the average case. We leave incorporating these orthogonal defenses out-of-scope.

Instead, in this work, we advocate focusing on eliminating leakage on (b), while providing a formulation that clearly captures the information that might be revealed. We formulate the admissible leakage as Ψ which captures the information (a) mentioned above, namely the input/output size and running time of each trusted computation unit invoked in the system. We formalize this intuition by defining the execution as a protocol between trusted components and the adversary, and define our privacy goal as achieving *privacy modulo-*Ψ.

Execution Protocol. Consider an honest execution of a program on input $I = \langle x_1, x_2, \ldots, x_n \rangle$. For a given map-reduce phase, let there be n map computation units. Let us label the map computation units such that the unit with label i takes x_i as input. Recall that the tuples generated by the map computation units are to be shuffled, and divided into groups according to the key-components. Let K to be the set of unique key-components and let $\pi : [n+1, n+m] \rightarrow K$ be a randomly chosen permutation, where $m = |K|$. Next, m reduce computation units are to be invoked. We label them starting from $n+1$, such that the computation unit i takes tuples with key-component $\pi(i)$ as input.

Let I_i, O_i, T_i be the respective input size (measured by number of tuples), output size, and processing time of the computation unit i, and call $\Psi_i = \langle I_i, O_i, T_i \rangle$ the *IO-profile* of computation unit i. The profile Ψ of the entire execution on input I is the sequence of Ψ_i for all computation units $i \in [1, \ldots, n+m]$ in the execution protocol. If an adversary $\widetilde{\mathscr{A}}$ can initiate the above protocol and observe Ψ, we say that the adversary has access to Ψ.

Now, let us consider the execution of the program on the same input $I = \langle x_1, x_2, \ldots, x_n \rangle$ under a MapReduce provisioning protocol by an adversary $\mathscr{A}$. A semi-honest adversary $\mathscr{A}$ can obtain information on the value of the input, output and processing time of every trusted instance, including information on trusted instances other than the map and reduce computation units. If the adversary is malicious, it can further tamper with the inputs and invocations of the instances. In the protocol, the adversary controls 6 parameters:

- **(C1)** the start time of each computation instance,
- **(C2)** the end time of each instance,
- **(C3)** the encrypted tuples passed to its inputs,
- **(C4)** the number of computation instances,
- **(C5)** order of computation units executed,
- **(C6)** the total number of map-reduce phases executed.

Since the adversary $\mathscr{A}$ can obtain "more" information and tamper the execution, a question to ask is, can the adversary $\mathscr{A}$ gain more knowledge compared to an adversary $\widetilde{\mathscr{A}}$ who only has access to Ψ? Using the standard notions of indistinguishability[2] and adversaries [28], we define a secure protocol as follows:

Definition 1 (Privacy modulo-Ψ). *A provisioning protocol for a program is modulo-*Ψ *private if, for any adversary* $\mathscr{A}$ *executing the MapReduce protocol, there is a adversary* $\widetilde{\mathscr{A}}$ *with access only to* Ψ*, such that the output of* $\mathscr{A}$ *and* $\widetilde{\mathscr{A}}$ *are indistinguishable.*

The definition states that the output of the adversaries can be directly seen as deduction made on the information available. The fact that all adversaries have output indistinguishable from the one which knows Ψ suggests that no additional information can be gained by any $\mathscr{A}$ beyond that implied by knowledge of Ψ.

Remarks. First, our definition follows the scenario proposed by Canneti [11], which facilitates universal composition. Hence, if a protocol is private module-Ψ for one map-reduce phase, then an entire sequence of phases executed is private module-Ψ. Note that our proposed $\mathtt{M}^2\mathtt{R}$ consists of a sequence of map, shuffle, and reduce phases where each phase starts only after the previous phase has completed, and the chain of MapReduce jobs are carried out sequentially. Thus, universal composition can be applied. Second, we point out that if the developer restructures the original computation to make the IO-profile the same for all inputs, then Ψ leaks nothing about the input. Therefore, the developer can consider using orthogonal techniques to mask timing latencies [41], hiding trace paths and IO patterns [34] to achieve ideal privacy, if the performance considerations permit so.

2.3 Assumptions

In this work, we make specific assumptions about the baseline system we build upon. First, we assume that the underlying hardware sufficiently protects each computation unit from malware and snooping attacks. The range of threats that are protected against varies based on the underlying trusted computing hardware. For instance,

[2] non-negligible advantage in a distinguishing game

traditional TPMs protect against software-only attacks but not against physical access to RAM via attacks such as cold-boot [24]. More recent trusted computing primitives, such as Intel SGX [40], encrypt physical memory and therefore offer stronger protection against adversaries with direct physical access. Therefore, we do not focus on the specifics of how to protect each computation unit, as it is likely to change with varying hardware platform used in deployment. In fact, our design can be implemented in any virtualization-assisted isolation that protects user-level processes on a malicious guest OS [12, 52, 57], before Intel SGX becomes available on the market.

Second, an important assumption we make is that of information leakage via side-channels (e.g. cache latencies, power) from a computation unit is minimal. Indeed, it is a valid concern and an area of active research. Both software and hardware-based solutions are emerging, but they are orthogonal to our techniques [18, 29].

Finally, to enable arbitrary computation on encrypted data, decryption keys need to be made made available to each hardware-isolated computation unit. This provisioning of client's keys to the cloud requires a set of trusted administrator interfaces and privileged software. We assume that such trusted key provisioning exists, as is shown in recent work [49, 65].

3 Attacks

In this section, we explain why a baseline system that merely encrypts the output of each computation unit leaks significantly more than a system that achieves privacy modulo-Ψ. We explain various subtle attack channels that our solution eliminates, with an example.

Running Example. Let us consider the canonical example of the Wordcount job in MapReduce, wherein the goal is to count the number of occurrences of each word in a set of input files. The map operation takes one file as input, and for each word w in the file, outputs the tuple $\langle w, 1\rangle$. All outputs are encrypted with standard authenticated encryption. Each reduce operation takes as input all the tuples with the same tuple-key, i.e. the same word, and aggregates the values. Hence the output of reduce operations is an encrypted list of tuples $\langle w, w_c\rangle$, where w_c is the frequency of word w for all input files. For simplicity, we assume that the input is a set of files $F = \{F_1, \ldots, F_n\}$, each file has the same number of words and is small enough to be processed by a map operation[3].

What does Privacy modulo-Ψ Achieve? Here all the map computation units output same size tuples, and after grouping, each reduce unit receives tuples grouped by words. The size of map outputs and group sizes constitute Ψ, and a private modulo-Ψ execution therefore leaks some statistical information about the collection of files in aggregate, namely the frequency distribution of words in F. However, it leaks nothing about the contents of words in the individual files — for instance, the frequency of words in any given file, and the common words between any pair of files are not leaked. As we show next, the baseline system permits a lot of inference attacks as it fails to achieve privacy modulo-Ψ. In fact, eliminating the remaining leakage in this example may not be easy, as it may assume apriori knowledge about the probability distribution of words in F (e.g. using differential privacy [48]).

Passive Attacks. Consider a semi-honest adversary that executes the provisioning protocol, but aims to infer additional information. The adversary controls 6 parameters **C1-C6** (Section 2.2) in the execution protocol. The number of units (**C4**) and map-reduce phases executed (**C6**) are dependent (and implied) by Ψ in an honest execution, and do not leak any additional information about the input. However, parameters **C1,C2,C3** and **C5** may directly leak additional information, as explained below.

- *Dataflow Patterns (Channel C3).* Assume that the encrypted tuples are of the same size, and hence do not individually leak anything about the underlying plain text. However, since the adversary constitutes the data communication channel, it can correlate the tuples written out by a map unit and read by a specific reduce unit. In the Wordcount example, the i^{th} map unit processes words in the file F_i, and then the intermediate tuples are sorted before being fed to reduce units. By observing which map outputs are grouped together to the same reduce unit, the adversary can learn that the word w_i in file F_i is the same as a word w_j in file F_j. This is true if they are received by the same reduce unit as one group. Thus, data access patterns leak significant information about overlapping words in files.
- *Order of Execution (Channel C5).* A deterministic order of execution of nodes in any step can reveal information about the underlying tuples beyond what is implied by Ψ. For instance, if the provisioning protocol always sorts tuple-keys and assigns them to reduce units in sorted order, then the adversary learns significant information. In the WordCount example, if the first reduce unit always corresponds to words appearing first in the sorted order, this would leak information about specific words processed by the reduce unit. This is not directly implied by Ψ.
- *Time-of-Access (Channel C1,C2)* Even if data access patterns are eliminated, time-of-access is an-

[3]Files can be processed in fixed size blocks, so this assumption is without any loss of generality

other channel of leakage. For instance, an optimizing scheduler may start to move tuples to the reduce units even before the map step is completed (pipelining) to gain efficiency. In such cases, the adversary can correlate which blocks written by map units are read by which reduce units. If outputs of all but the i^{th} map unit are delayed, and the j^{th} reduce unit completes, then the adversary can deduce that there is no dataflow from the i^{th} map unit to j^{th} reduce unit. In general, if computation units in a subsequent step do not synchronize to obtain outputs from all units in the previous step, the time of start and completion leaks information.

Active Attacks. While we allow the adversary to abort the computation session at any time, we aim to prevent the adversary from using active attacks to gain advantage in breaking confidentiality. We remind readers that in our baseline system, the adversary can only invoke the program with its complete input set, without tampering with any original inputs. The output tuple-set of each computation unit is encrypted with an authenticated encryption scheme, so the adversary cannot tamper with individual tuples. Despite these preliminary defenses, several channels for active attacks exist:

- *Tuple Tampering*. The adversary may attempt to duplicate or eliminate an entire output tuple-set produced by a computation unit, even though it cannot forge individual tuples. As an attack illustration, suppose the adversary wants to learn how many words are unique to an input file F_i. To do this, the adversary can simply drop the output of the i^{th} map unit. If the number of tuples in the final output reduces by k, the tuples eliminated correspond to k unique words in F_i.
- *Misrouting Tuples*. The adversary can reorder intermediate tuples or route data blocks intended for one reduce unit to another. These attacks subvert our confidentiality goals. For instance, the adversary can bypass the shuffler altogether and route the output of i^{th} map unit to a reduce unit. The output of this reduce unit leaks the number of unique words in F_i. Similar inference attacks can be achieved by duplicating outputs of tuples in the reduce unit and observing the result.

4 Design

Our goal is to design a MapReduce provisioning protocol which is private modulo-Ψ and adds a small amount of the TCB to the existing MapReduce platform. We explain the design choices available and our observations that lead to an efficient and clean security design.

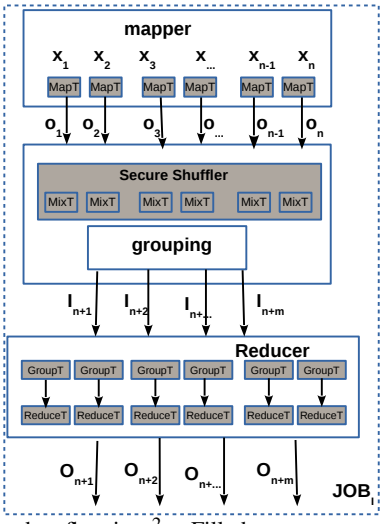

Figure 2: The data flow in M^{2}R. Filled components are trusted. Input, intermediate and output tuples are encrypted. The original map and reduce operations are replaced with `mapT` and `reduceT`. New components are the mixer nodes which use `mixT`, and another trusted component called `groupT`.

4.1 Architecture Overview

The computation proceeds in phases, each consisting of a map step, a shuffle step, and a reduce step. Figure 2 depicts the 4 new trusted components our design introduces into the dataflow pipeline of MapReduce. These four new TCB components are `mapT`, `reduceT`, `mixT` and `groupT`. Two of these correspond to the execution of map and reduce unit. They ensure that output tuples from the map and reduce units are encrypted and each tuple is of the same size. The other 2 components implement the critical role of *secure shuffling*. We explain our non-intrusive mechanism for secure shuffling in Section 4.2. Further, all integrity checks to defeat active attacks are designed to be distributed requiring minimal global synchronization. The shuffler in the MapReduce platform is responsible for grouping tuples, and invoking reduce units on disjoint ranges of tuple-keys. On each cluster node, the reducer checks the grouped order and the expected range of tuples received using the trusted `groupT` component. The outputs of the reduce units are then fed back into the next round of map-reduce phase.

Minimizing TCB. In our design, a major part of the MapReduce's software stack deals with job scheduling and I/O operations, hence it can be left outside of the TCB. Our design makes no change to the grouping and scheduling algorithms, and they are outside our TCB as shown in the Figure 2. Therefore, the design is conceptually simple and requires no intrusive changes to be implemented over existing MapReduce implementations. Developers need to modify their original applications to prepare them for execution in a hardware-protected pro-

cess in our baseline system, as proposed in previous systems [38, 39, 49]. Beyond this modification made by the baseline system to the original MapReduce, M^{2}R requires a few additional lines of code to invoke the new privacy-enhancing TCB components. That is, MapReduce applications need modifications only to invoke components in our TCB. Next, we explain how our architecture achieves privacy and integrity in a MapReduce execution, along with the design of these four TCB components.

4.2 Privacy-Preserving Execution

For any given execution, we wish to ensure that each computation step in a phase is private modulo-Ψ. If the map step, the shuffle step, and the reduce step are individually private modulo-Ψ, by the property of serial composibility, the entire phase and a sequence of phases can be shown to be private. We discuss the design of these steps in this section, assuming a honest-but-curious adversary limited to passive attacks. The case of malicious adversaries is discussed in Section 4.3.

4.2.1 Secure Shuffling

As discussed in the previous section, the key challenge is performing secure shuffling. Consider the naive approach in which we simply move the entire shuffler into the platform TCB of each cluster node. To see why this is insecure, consider the grouping step of the shuffler, often implemented as a distributed sort or hash-based grouping algorithm. The grouping algorithm can only process a limited number of tuples locally at each mapper, so access to intermediate tuples must go to the network during the grouping process. Here, network data access patterns from the shuffler leak information. For example, if the shuffler were implemented using a standard merge sort implementation, the merge step leaks the relative position of the pointers in sorted sub-arrays as it fetches parts of each sub-array from network incrementally[4].

One generic solution to hide data access patterns is to employ an ORAM protocol when communicating with the untrusted storage backend. The grouping step will then access data obliviously, thereby hiding all correlations between grouped tuples. This solution achieves strong privacy, but with an overhead of $O(\log^k N)$ for each access when the total number of tuples is N [55]. Advanced techniques can be employed to reduce the overhead to $O(\log N)$, i.e. $k = 1$ [43]. Nevertheless, using a sorting algorithm for grouping, the total overhead becomes $O(N \log^{k+1} N)$, which translates to a factor of $30-100\times$ slowdown when processing gigabytes of shuffled data.

[4]This can reveal, for instance, whether the first sub-array is strictly lesser than the first element in the second sorted sub-array.

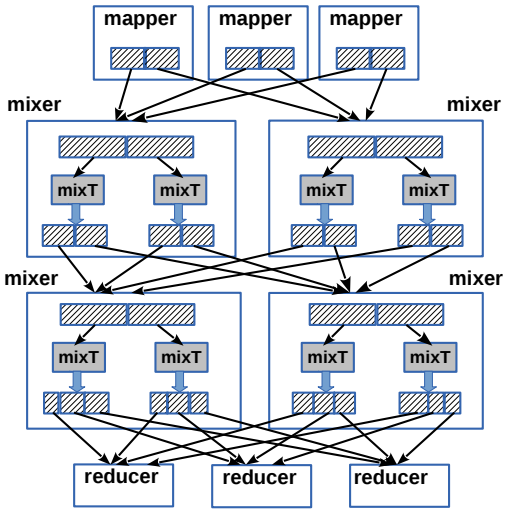

Figure 3: High level overview of the map-mix-reduce execution using a 2-round mix network.

A more advanced solution is to perform oblivious sorting using sorting networks, for example, odd-even or bitonic sorting network [23]. Such an approach hides data access patterns, but admits a $O(\log^2 N)$ latency (additive only). However, sorting networks are often designed for a fixed number of small inputs and hard to adapt to tens of gigabytes of distributed data.

We make a simple observation which yields a non-intrusive solution. Our main observation is that in MapReduce and other dataflow frameworks, the sequence of data access patterns is fixed: it consists of cycles of tuple writes followed by reads. The reduce units start reading and processing their inputs only after the map units have finished. In our solution, we rewrite intermediate encrypted tuples with re-randomized tuple keys such that there is no linkability between the re-randomized tuples and the original encrypted map output tuples. We observe that this step can be realized by secure mix networks [30]. The privacy of the computation reduces directly to the problem of secure mixing. The total latency added by our solution is an additive term of $O(\log N)$ in the worst case. Since MapReduce shuffle step is based on sorting which already admits $O(N \log N)$ overhead, our design retains the asymptotic runtime complexity of the original framework.

Our design achieves privacy using a cascaded mix network (or cascaded-mix) to securely shuffle tuples [30]. The procedure consists of a cascading of κ intermediate steps, as shown in Figure 3. It has κ identical steps (called mixing steps) each employing a number of trusted computation units called `mixT` units, the execution of which can be distributed over multiple nodes called mixers. Each `mixT` takes a fixed amount of T tuples that it can process in memory, and passes exactly the same number of encrypted tuples to all `mixT` units in the sub-

sequent step. Therefore, in each step of the cascade, the mixer utilizes N/T `mixT` units for mixing N tuples. At $\kappa = \log \frac{N}{T}$, the network ensures the strongest possible unlinkability, that is, the output distribution is statistically indistinguishable from a random distribution [30].

Each `mixT` unit decrypts the tuples it receives from the previous step, randomly permutes them using a linear-time algorithm and re-encrypts the permuted tuples with fresh randomly chosen symmetric key. These keys are known only to `mixT` units, and can be derived using a secure key-derivation function from a common secret. The processing time of `mixT` are padded to a constant. Note that the re-encryption time has low variance over different inputs, therefore such padding incurs low overhead.

Let Ω represents the number of input and output tuples of cascaded-mix with κ steps. Intuitively, when κ is sufficiently large, an semi-honest adversary who has observed the execution does not gain more knowledge than Ω. The following lemma states that indeed this is the case. We present the proof in Appendix A.

Lemma 1. *Cascaded-mix is private module-Ω under semi-honest adversary, given that the underlying encryption scheme is semantically secure.*

4.2.2 Secure Grouping

After the mixing step, the shuffler can group the randomized tuple keys using its original (unmodified) grouping algorithm, which is not in the TCB. The output of the cascaded-mix is thus fed into the existing grouping algorithm of MapReduce, which combines all tuples with the same tuple-key and forward them to reducers. Readers will notice that if the outputs of the last step of the cascaded-mix are probabilistically encrypted, this grouping step would need to be done in a trusted component. In our design, we add a last $(\kappa+1)$-th step in the cascade to accommodate the requirement for subsequent grouping. The last step in the cascade uses a deterministic symmetric encryption F_s, with a secret key s, to encrypt the key-component of the final output tuples. Specifically, the $\langle a,b\rangle$ is encrypted to a ciphertext of the form $\langle F_s(a), E(a,b)\rangle$, where $E(\cdot)$ is a probabilistic encryption scheme. This ensures that the two shuffled tuples with the same tuple-keys have the same ciphertext for the key-component of the tuple, and hence the subsequent grouping algorithm can group them without decrypting the tuples. The secret key s is randomized in each invocation of the cascaded-mix, thereby randomizing the ciphertexts across two map-reduce phases or jobs.

What the adversary gains by observing the last step of mixing is the tuples groups which are permuted using $F_s(\cdot)$. Thus, if $F_s(\cdot)$ is a pseudorandom function family, the adversary can only learn about the size of each group, which is already implied by Ψ. Putting it all together with the Lemma 1, we have:

Theorem 1. *The protocol M^2R is modulo-Ψ private (under semi-honest adversary), assuming that the underlying private-key encryption is semantically secure, and $F_s(\cdot)$ is a pseudorandom function family.*

4.3 Execution Integrity

So far, we have considered the privacy of the protocol against honest-but-curious adversaries. However, a malicious adversary can deviate arbitrarily from the protocol by mounting active attacks using the 6 parameters under its control. In this section, we explain the techniques necessary to prevent active attacks.

The program execution in $\mathtt{M^2R}$ can be viewed as a directed acyclic graph (DAG), where vertices denote trusted computation units and edges denote the flow of encrypted data blocks. $\mathtt{M^2R}$ has 4 kinds of trusted computation units or vertices in the DAG: `mapT`, `mixT`, `groupT`, and `reduceT`. At a high-level, our integrity-checking mechanism works by ensuring that nodes at the j^{th} level (by topologically sorted order) check the consistency of the execution at level $j-1$. If they detect that the adversary deviates or tampers with the execution or outputs from level $j-1$, then they abort the execution.

The MapReduce provisioning system is responsible for invoking trusted computation units, and is free to decide the total number of units spawned at each level j. We do not restrict the MapReduce scheduling algorithm to decide which tuples are processed by which reduce unit, and their allocation to nodes in the cluster. However, we ensure that all tuples output at level $i-1$ are processed at level i, and there is no duplicate. Note that this requirement ensures that a computation in step i starts only after outputs of previous step are passed to it, implicitly synchronizes the start of the computation units at step i. Under this constraint, it can be shown that channels **C1-C2** (start-end time of each computation node) can only allow the adversary to delay an entire step, or distinguish the outputs of units within one step, which is already implied by Ψ. We omit a detailed proof in this paper. Using these facts, we can show that the malicious adversary has no additional advantage compared to an honest-but-curious adversary, stated formally below.

Theorem 2. *The protocol M^2R is private modulo-Ψ under malicious adversary, assuming that the underlying authenticated-encryption is semantically secure (confidentiality) and secure under chosen message attack (integrity), and $F_s(\cdot)$ is a pseudorandom function family.*

Proof Sketch: Given a malicious adversary $\mathscr{A}$ that executes the $\mathtt{M^2R}$ protocol, we can construct an adversary

$\widetilde{\mathscr{A}}$ that simulates $\mathscr{A}$, but only has access to Ψ in the following way. To simulate $\mathscr{A}$, the adversary $\widetilde{\mathscr{A}}$ needs to fill in information not present in Ψ. For the output of a trusted unit, the simulation simply fills in random tuples, where the number of tuples is derived from Ψ. The timing information can likewise be filled-in. Whenever $\mathscr{A}$ deviates from the protocol and feeds a different input to a trusted instance, the simulation expects the instance will halt and fills in the information accordingly. Note that the input to $\mathscr{A}$ and the input constructed for the simulator $\widetilde{\mathscr{A}}$ could have the same DAG of program execution, although the encrypted tuples are different. Suppose there is a distinguisher that distinguishes $\mathscr{A}$ and $\widetilde{\mathscr{A}}$, let us consider the two cases: either the two DAG's are the same or different. If there is a non-negligible probability that they are the same, then we can construct a distinguisher to contradict the security of the encryption, or $F_s(\cdot)$. If there is a non-negligible probability that they are different, we can forge a valid authentication tag. Hence, the outputs of $\mathscr{A}$ and $\widetilde{\mathscr{A}}$ are indistinguishable. ■

Integrity Checks. Nearly all our integrity checks can be distributed across the cluster, with checking of invariants done locally at each trusted computation. Therefore, our integrity checking mechanism can largely bundle the integrity metadata with the original data. No global synchronization is necessary, except for the case of the `groupT` units as they consume data output by an untrusted grouping step. The `groupT` checks ensure that the ordering of the grouped tuples received by the designated `reduceT` is preserved. In addition, `groupT` units synchronize to ensure that each reducer processes a distinct range of tuple-keys, and that all the tuple-keys are processed by at least one of the reduce units.

4.3.1 Mechanisms

In the DAG corresponding to a program execution, the MapReduce provisioning system assigns unique *instance ids*. Let the vertex i at the level j has the designated id (i, j), and the total number of units at level j be $|V_j|$. When a computation instance is spawned, its designed instance id (i, j) and the total number of units $|V_j|$ are passed as auxiliary input parameters by the provisioning system. Each vertex with id (i, j) is an operation of type `mapT`, `groupT`, `mixT` or `reduceT`, denoted by the function $OpType(i, j)$. The basic mechanism for integrity-checking consists of each vertex emitting a *tagged-block* as output which can be checked by trusted components in the next stage. Specifically, the tagged block is 6-tuple $B = \langle O, LvlCnt, SrcID, DstID, DstLvl, DstType \rangle$, where:

O	is the encrypted output tuple-set,
$LvlCnt$	is the number of units at source level,
$SrcID$	is the instance id of the source vertex,
$DstID$	is instance id of destination vertex or `NULL`
$DstLvl$	is the level of the destination vertex,
$DstType$	is the destination operation type.

In our design, each vertex with id (i, j) fetches the tagged-blocks from all vertices at the previous level, denoted by the multiset $\mathscr{B}$, and performs the following consistency checks on $\mathscr{B}$:

1. The LvlCnt for all $b \in \mathscr{B}$ are the same (say $\ell(\mathscr{B})$).
2. The SrcID for all $b \in \mathscr{B}$ are distinct.
3. For set $S = \{SrcID(b) \mid b \in \mathscr{B}\}$, $|S| = \ell(\mathscr{B})$.
4. For all $b \in \mathscr{B}$, $DstLvl(b) = j$.
5. For all $b \in \mathscr{B}$, $DstID(b) = (i, j)$ or `NULL`.
6. For all $b \in \mathscr{B}$, $DstType(b) = OpType(i, j)$.

Conditions 1,2 and 3 ensure that tagged-blocks from *all* units in the previous level are read and that they are distinct. Thus, the adversary has not dropped or duplicated any output tuple. Condition 4 ensures that the computation nodes are ordered sequentially, that is, the adversary cannot misroute data bypassing certain levels. Condition 6 further checks that execution progresses in the expected order — for instance, the map step is followed by a mix, subsequently followed by a group step, and so on. We explain how each vertex decides the right or expected order independently later in this section. Condition 5 states that if the source vertex wishes to fix the recipient id of a tagged-block, it can verifiably enforce it by setting it to non-NULL value.

Each tagged-block is encrypted with standard authenticated encryption, protecting the integrity of all metadata in it. We explain next how each trusted computation vertex encodes the tagged-block.

Map-to-Mix DataFlow. Each `mixT` reads the output metadata of all `mapT`. Thus, each `mixT` knows the total number of tuples N generated in the entire map step, by summing up the counts of encrypted tuples received. From this, each `mixT` independently determines the total number of mixers in the system as N/T. Note that T is the pre-configured number of tuples that each `mixT` can process securely without invoking disk accesses, typically a 100M of tuples. Therefore, this step is completely decentralized and requires no co-ordination between `mixT` units. A `mapT` unit invoked with id (i, j) simply emit tagged-blocks, with the following structure: $\langle \cdot, |V_j|, (i, j), \texttt{NULL}, j+1, \texttt{mixT} \rangle$.

Mix-to-Mix DataFlow. Each `mixT` re-encrypts and permutes a fixed number (T) of tuples. In a κ-step cascaded mix network, at any step s ($s < \kappa - 1$) the `mixT` outputs T/m tuples to each one of the m `mixT` units in the

step $s+1$. To ensure this, each mixT adds metadata to its tagged-block output so that it reaches only the specified mixT unit for the next stage. To do so, we use the *DstType* field, which is set to type $\texttt{mixT}_{s+1}$ by the mixer at step s. Thus, each mixT node knows the total number of tuples being shuffled N (encoded in OpType), its step number in the cascaded mix, and the public value T. From this each mixT can determine the correct number of cascade steps to perform, and can abort the execution if the adversary tries to avoid any step of the mixing.

Mix-to-Group DataFlow. In our design, the last mix step (i, j) writes the original tuple as $\langle F_s(k), (k, v, tctr)\rangle$, where the second part of this tuple is protected with authenticated-encryption. The value $tctr$ is called a tuple-counter, which makes each tuple globally distinct in the job. Specifically, it encodes the value (i, j, ctr) where ctr is a counter unique to the instance (i, j). The assumption here is that all such output tuples will be grouped by the first component, and each group will be forwarded to reducers with no duplicates. To ensure that the outputs received are correctly ordered and untampered, the last mixT nodes send a special tagged-block to groupT nodes. This tagged-block contains the count of tuples corresponding to $F_s(k)$ generated by mixT unit with id (i, j). With this information each groupT node can locally check that:

- For each received group corresponding to $g = F_s(k)$, the count of distinct tuples $(k, \cdot, i, j, ctr)$ it receives tallies with that specified in the tagged-block received from mixT node (i, j), for all blocks in $\mathscr{B}$.

Finally, groupT units need to synchronize to check if there is any overlap between tuple-key ranges. This requires an additional exchange of tokens between groupT units containing the range of group keys and tuple-counters that each unit processes.

Group-to-Reduce Dataflow. There is a one-to-one mapping between groupT units and reduceT units, where the former checks the correctness of the tuple group before forwarding to the designated reduceT. This communication is analogous to that between mixT units, so we omit a detailed description for brevity.

5 Implementation

Baseline Setup. The design of M^{2}R can be implemented differently depending on the underlying architectural primitives available. For instance, we could implement our solution using Intel SGX, using the mechanisms of VC^3 to achieve our baseline. However, Intel SGX is not yet available in shipping CPUs, therefore we use a trusted-hypervisor approach to implement the baseline system, which minimizes the performance overheads from the baseline system. We use Intel TXT to securely boot a trusted Xen-4.4.3 hypervisor kernel, ensuring its static boot integrity[5]. The inputs ands output of map and reduce units are encrypted with AES-GCM using 256-bit keys. The original Hadoop jobs are executed as user-level processes in ring-3, attested at launch by the hypervisor, making an assumption that they are protected during subsequent execution. The MapReduce jobs are modified to call into our TCB components implemented as x86 code, which can be compiled with SFI constraints for additional safety. The hypervisor loads, verifies and executes the TCB components within its address space in ring-0. The rest of Hadoop stack runs in ring 3 and invokes the units by making hypercalls. Note that the TCB components can be isolated as user-level processes in the future, but this is only meaningful if the processes are protected by stronger solutions such as Intel SGX or other systems [12, 14, 52].

M^{2}R TCB. Our main contributions are beyond the baseline system. We add four new components to the TCB of the baseline system. We have modified a standard Hadoop implementation to invoke the mixT and groupT units before and after the grouping step. These two components add a total 90 LoC to the platform TCB. No changes are necessary to the original grouping algorithm. Each mapT and reduceT implement the trusted map and reduce operation — same as in the baseline system. They are compiled together with a static utility code which is responsible for (a) padding each tuple to a fixed size, (b) encrypting tuples with authenticated encryption, (c) adding and verifying the metadata for tagged-blocks, and (d) recording the instance id for each unit. Most of these changes are fairly straightforward to implement. To execute an application, the client encrypts and uploads all the data to M^{2}R nodes. The user then submits M^{2}R applications and finally decrypts the results.

6 Evaluation

This section describes M^{2}R performance in a small cluster under real workloads. We ported 7 data intensive jobs from the standard benchmark to M^{2}R, making less than 25% changes in number of lines of code (LoC) to the original Hadoop jobs. The applications add fewer than 500 LoC into the TCB, or less than 0.16% of the entire Hadoop software stack. M^{2}R adds $17-130\%$ overhead in running time to the baseline system. We also compare M^{2}R with another system offering the same level of privacy, in which encrypted tuples are sent back to a trusted client. We show that M^{2}R is up to $44.6\times$ faster compared

[5]Other hypervisor solutions such as TrustVisor [39], Overshadow [14], Nova [56], SecVisor [50] could equivalently be used

Job	*LoC changed (vs. Hadoop job)*	*TCB increase (vs. Hadoop codebase)*	*Input size (vs. plaintext size)*	*Shuffled bytes*	*# App hypercalls*	*# Platform hypercall*
Wordcount	10 (15%)	370 (0.14%)	2.1G (1.06×)	4.2G	3277173	35
Index	28 (24%)	370 (0.14%)	2.5G (1.15×)	8G	3277173	59
Grep	13 (13%)	355 (0.13%)	2.1G (1.06×)	75M	3277174	10
Aggregate	16 (18%)	395 (0.15%)	2G (1.19×)	289M	18121377	12
Join	30 (22%)	478 (0.16%)	2G (1.19×)	450M	11010647	14
Pagerank	42 (20%)	429 (0.15%)	2.5G (4×)	2.6G	1750000	21
KMeans	113 (7%)	400 (0.12%)	1G (1.09×)	11K	12000064	8

Table 1: Summary of the porting effort and TCB increase for various M^{2}R applications, and the application runtime cost factors. Number of app hypercalls consists of both `mapT` and `reduceT` invocations. Number of platform hypercalls include `groupT` and `mixT` invocations.

Job	*Baseline (vs. no encryption)*	*M^{2}R (% increase vs. baseline)*	*Download-and-compute (× M^{2}R)*
Wordcount	570 (221)	1156 (100%)	1859 (1.6×)
Index	666 (423)	1549 (130%)	2061 (1.3×)
Grep	70 (48)	106 (50%)	1686 (15.9×)
Aggregate	125 (80)	205 (64%)	9140 (44.6×)
Join	422 (211)	510 (20%)	5716 (11.2×)
Pagerank	521 (334)	755 (44%)	1209 (1.6×)
KMeans	123 (71)	145 (17%)	6071 (41.9×)

Table 2: Overall running time (s) of M^{2}R applications in comparison with other systems: (1) the baseline system protecting computation only in single nodes, (2) the download-and-compute system which does not use trusted primitives but instead sends the encrypted tuples back to trusted servers when homomorphic encrypted computation is not possible [59].

to this solution.

6.1 Setup & Benchmarks

We select a standard benchmark for evaluating Hadoop under large workloads called HiBench suite [25]. The 7 benchmark applications, listed in Table 1, cover a wide range of data-intensive tasks: compute intensive (KMeans, Grep, Pagerank), shuffle intensive (Wordcount, Index), database queries (Join, Aggregate), and iterative (KMeans, Pagerank). The size of the encrypted input data is between 1 GB and 2.5 GB in these case studies. Different applications have different amount of shuffled data, ranging from small sizes (75MB in Grep, 11K in KMeans) to large sizes (4.2GB in Wordcount, 8GB in Index).

Our implementation uses the Xen-4.3.3 64-bit hypervisor compiled with trusted boot option. The rest of M^{2}R stack runs on Ubuntu 13.04 64-bit version. We conduct our experiments in a cluster of commodity servers equipped with 1 quad-core Intel CPU 1.8GHz, 1TB hard drive, 8GB RAM and 1GB Ethernet cards. We vary our setup to have between 1 to 4 compute nodes (running mappers and reducers) and between 1 to 4 mixer nodes for implementing a 2-step cascaded mix network. The results presented below are from running with 4 compute nodes and 4 mixers each reserving a 100MB buffer for mixing, averaged over 10 executions.

6.2 Results: Performance

Overheads & Cost Breakdown. We observe a linear scale-up with the number of nodes in the cluster, which confirms the scalability of M^{2}R. In our benchmarks (Table 2), we observe a total overhead of between $17\% - 130\%$ over the baseline system that simply encrypts inputs and outputs of map/reduce units, and utilizes none of our privacy-enhancing techniques. It can also be seen that in all applications except for Grep and KMeans, running time is proportional to the size of data transferred during shuffling (shuffled bytes column in Table 1). To understand the cost factors contributing to the overhead, we measure the time taken by the secure shuffler, by the `mapT` and `reduceT` units, and by the rest of the Hadoop system which comprises the time spent on I/O, scheduling and other book-keeping tasks. This relative cost breakdown is detailed in Figure 4. From the result, we observe that the cost of the secure shuffler is significant. Therefore, reducing the overheads of shuffling, by avoiding the generic ORAM solution, is well-incentivized and is critical to reducing the overall overheads. The two main benchmarks which have high overheads of over 100%, namely Wordcount and Index, incur this cost primarily due to the cost of privacy-preserving shuffling a large amount of data. In benchmarks where the shuffled data is small (Grep, KMeans), the use of `mapT`/`reduceT` adds relatively larger overheads than that from the secure shuffler. The second observation is that the total cost of the both shuffler and other trusted components is comparable to that of Hadoop, which provides evidence that M^{2}R preserves the asymptotic complexity of Hadoop.

Comparison to Previous Solutions. Apart from the baseline system, a second point of comparison are previously proposed systems that send encrypted tuples to the user for private computation. Systems such as Monomi [59] and AutoCrypt [58] employ homomorphic encryption for computing on encrypted data on the single servers. For operations that cannot be done on the server using partially homomorphic encryption, such Monomi-like systems forward the data to a trusted set of servers (or to the client's private cloud) for decryption. We refer to this approach as download-and-compute approach. We estimate the performance of a Monomi-like system extended to distributed computation tasks, for achieving privacy equivalent to ours. To compare, we assume that the system uses Paillier, ElGamal and randomized

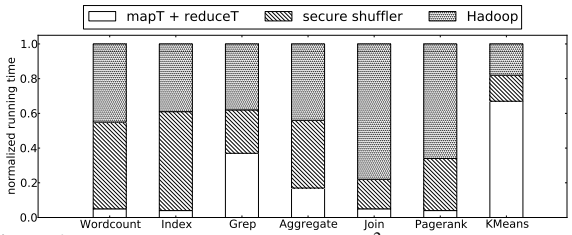

Figure 4: Normalized break-down time for M^{2}R applications. The running time consists of the time taken by mapT and reduceT, plus the time by the secure shuffler. The rest comes from the Hadoop runtime.

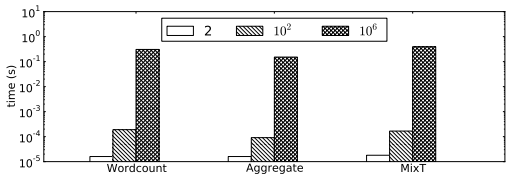

Figure 5: Cost of executing mapT instance of the Wordcount and Aggregate job, and the cost for executing mixT. Input sizes (number of ciphertexts per input) varies from 2 to 10^6.

search schemes for homomorphic computation, but not OPE or deterministic schemes (since that leaks more than M^{2}R and our baseline system do). We run operations that would fall outside such the expressiveness of the allowed homomorphic operations, including shuffling, as a separate network request to the trusted client. We batch network requests into one per MapReduce step. We assume that the network round trip latency to the client is only 1ms — an optimistic approximation since the average round trip delay in the same data center is $10-100$ms [4, 61]. We find that this download-and-compute approach is slower compared to ours by a factor of $1.3\times$ to $44.6\times$ (Table 2), with the median benchmark running slower by $11.2\times$. The overheads are low for case-studies where most of the computation can be handled by homomorphic operations, but most of the benchmarks require conversions between homomorphic schemes (thereby requiring decryption) [58, 59] or computation on plaintext values.

Platform-Specific Costs. Readers may wonder if the evaluation results are significantly affected by the choice of our implementation platform. We find that the dominant costs we report here are largely complementary to the costs incurred by the specifics of the underlying platform. We conduct a micro-benchmark to evaluate the cost of context-switches and the total time spent in the trusted components to explain this aspect. In our platform, the cost of each hypercall (switch to trusted logic) is small ($13\mu s$), and the execution of each trusted component is largely proportional to the size of its input data as shown in Figure 5. The time taken by the trusted computation grows near linearly with the input data-size, showing that the constant overheads of context-switches and other platform's specifics do not contribute to the reported results significantly. This implies that simple optimizations such as batching multiple trusted code invocations would not yield any significant improvements, since the overheads are indeed proportional to the total size of data and not the number of invocations. The total number of invocations (via hypercalls) for app-specific trusted logic (mapT, reduceT) is proportional to the total number input tuples, which amounts for less than half a second overhead even for millions of input tuples. The number of invocations to the other components (mixT and groupT) is much smaller ($8-59$) and the each invocation operates on large inputs of a few gigabytes; therefore the dominant cost is not that of context-switches, but that of the cost of multi-step shuffling operation itself and the I/O overheads.

6.3 Results: Security & Porting Effort

Porting effort. We find that the effort to adapt all benchmarks to M^{2}R is modest at best. For each benchmark, we report the number of Java LoC we changed in order to invoke the trusted components in M^{2}R, measured using the sloccount tool [6]. Table 1 shows that all applications except for KMeans need to change fewer than 43 LoC. Most changes are from data marshaling before and after invoking the mapT and reduceT units. KMeans is more complex as it is a part of the Mahout distribution and depends on many other utility classes. Despite this, the change is only 113 LoC, or merely 7% of the original KMeans implementation.

TCB increase. We define our *TCB increase* as the total size of the four trusted components. This represents the additional code running on top of a base TCB, which in our case is Xen. Note that our design can eliminate the base TCB altogether in the future by using SGX enclaves, and only retain the main trusted components we propose in M^{2}R. The TCB increase comprises the per-application trusted code and platform trusted code. The former consists of the code for loading and executing mapT, reduceT units (213 LoC) as well as the code for implementing their logic. Each map/reduce codebase itself is small, fewer than 200 LoC, and runs as trusted components in the baseline system itself. The platform trusted code includes that of mixT and groupT, which amounts to 90 LoC altogether. The entire Hadoop software stack is over 190*K* LoC and M^{2}R avoids moving all of it into the TCB. Table 1 shows that all jobs have TCB increases of fewer than 500 LoC, merely 0.16% of the Hadoop codebase.

Security. M^{2}R achieves stronger privacy than previous

[6] http://www.dwheeler.com/sloccount

Job	M^{2}R	*Baseline (additional leakage)*
Wordcount	# unique words + count	word-file relationship
Index	# unique words + count	word-file relationship
Grep	nothing	nothing
Aggregate	# groups + group size	record-group relationship
Join	# groups + group size	record-group relationship
Pagerank	node in-degree	whole input graph
KMeans	nothing	nothing

Table 3: Remaining leakage of M^{2}R applications, compared with that in the baseline system.

platforms that propose to use encrypted computation for big-data analysis. Our definition allows the adversary to observe an admissible amount of information, captured by Ψ, in the computation but hides everything else. It is possible to quantitatively analyze the increased privacy in information-theoretic terms, by assuming the probability distribution of input data [37, 53]. However, here we present a qualitative description in Table 3 highlighting how much privacy is gained by the techniques introduced in M^{2}R over the baseline system. For instance, consider the two case studies that incur most performance overhead (Wordcount, Index). In these examples, merely encrypting the map/reduce tuples leaks information about which file contains which words. This may allow adversaries to learn the specific keywords in each file in the dataset. In M^{2}R, this leakage is reduced to learning only the total number of unique words in the complete database and the counts of each, hiding information about individual files. Similarly, M^{2}R hides which records are in which group for database operations (Aggregate and Join). For Pagerank, the baseline system leaks the complete input graph edge structure, giving away which pair of nodes has an edge, whereas M^{2}R reduces this leakage to only the in-degree of graph vertices. In the two remaining case studies, M^{2}R provides no additional benefit over the baseline.

7 Related Work

Privacy-preserving data processing. One of M^{2}R's goal is to offer large-scale data processing in a privacy preserving manner on untrusted clouds. Most systems with this capability are in the database domain, i.e. supporting SQL queries processing. CryptDB [47] takes a purely cryptographic approach, showing the practicality of using partially homomorphic encryption schemes [3, 15, 45, 46, 54]. CryptDB can only work on a small set of SQL queries and therefore is unable to support arbitrary computation. Monomi [59] supports more complex queries, by adopting the download-and-compute approach for complex queries. As shown in our evaluation, such an approach incurs an order of magnitude larger overheads.

There exist alternatives supporting outsourcing of query processing to a third party via server-side trusted hardware, e.g. IBM 4764/5 cryptographic co-processors. TrustedDB [7] demonstrated that a secure outsourced database solution can be built and run at a fraction of the monetary cost of any cryptography-enabled private data processing. However, the system requires expensive hardware and a large TCB which includes the entire SQL server stack. Cipherbase improves upon TrustedDB by considering encrypting data with partially homomorphic schemes, and by introducing a trusted entity for query optimization [6]. M^{2}R differs to these systems in two fundamental aspects. First, it supports general computation on any type of data, as opposed to being restricted to SQL and structured database semantics. Second, and more importantly, M^{2}R provides confidentiality in a distributed execution environment which introduces more threats than in a single-machine environment.

VC^3 is a recent system offering privacy-preserving general-purpose data processing [49]. It considers MapReduce and utilizes Intel SGX to maintain a small TCB. This system is complementary to M^{2}R, as it focuses on techniques for isolated computation, key management, etc. which we do not consider. The privacy model in our system is stronger than that of *VC*^{3} which does not consider traffic analysis attacks.

GraphSC offers a similar security guarantee to that of M^{2}R for specialized graph-processing tasks [42]. It provides a graph-based programming model similar to GraphLab's [36], as opposed to the dataflow model exposed by M^{2}R. GraphSC does not employ trusted primitives, but it assumes two non-colluding parties. There are two main techniques for ensuring data-oblivious and secure computation in GraphSC: sorting and garbled circuits. However, these techniques result in large performance overheads: a small Pagerank job in GraphSC is $200,000\times - 500,000\times$ slower than in GraphLab without security. M^{2}R achieves an overhead of $2\times - 5\times$ increase in running time because it leverages trusted primitives for computation on encrypted data. A direct comparison of oblivious sorting used therein instead of our secure shuffler is a promising future work.

Techniques for isolated computation. The current implementation of M^{2}R uses a trusted hypervisor based on Xen for isolated computation. Overshadow [14] and CloudVisor [63] are techniques with large TCB, whereas Flicker [38] and TrustVisor [39] reduce the TCB at the cost of performance. Recently, Minibox [32] enhances a TrustVisor-like hypervisor with two-way protection providing security for both the OS and the applications (or PALs). Advanced hardware-based techniques include Intel SGX [40] and Bastion [12] provide a hardware protected secure mode in which applications can be executed at hardware speed. All these techniques are complementary to ours.

Mix networks. The concept of mix network is first described in the design of untraceable electronic mail [13]. Since then, a body of research has concentrated on building, analyzing and attacking anonymous communication systems [16, 19]. Canetti presents the first definition of security that is preserved under composition [11], from which others have shown that the mix network is secure under Canetti's framework [10, 60]. Security properties of cascaded mix networks were studied in [30]. We use these theoretical results in our design.

8 Conclusion & Future Work

In this paper, we defined a model of privacy-preserving distributed execution of MapReduce jobs. We analyzed various attacks channels that break data confidentiality on a baseline system which employs both encryption and trusted computing primitives. Our new design realizes the defined level of security, with a significant step towards lower performance overhead while requiring a small TCB. Our experiments with M^2R showed that the system requires little effort to port legacy MapReduce applications, and is scalable.

Systems such as M^2R show evidence that specialized designs to hide data access patterns are practical alternatives to generic constructions such as ORAM. The question of how much special-purpose constructions benefit important practical systems, as compared to generic constructions, is an area of future work. A somewhat more immediate future work is to integrate our design to other distributed dataflow systems. Although having the similar structure of computation, those systems are based on different sets of computation primitives and different execution models, which presents both opportunities and challenges for reducing the performance overheads of our design. Another avenue for future work is to realize our model of privacy-preserving distributed computation in the emerging in-memory big-data platforms [64], where only very small overheads from security mechanisms can be tolerated.

9 Acknowledgements

The first author was funded by the National Research Foundation, Prime Minister's Office, Singapore, under its Competitive Research Programme (CRP Award No. NRF-CRP8-2011-08). A special thanks to Shruti Tople and Loi Luu for their help in preparing the manuscript. We thank the anonymous reviewers for their insightful comments that helped us improve the discussions in this work.

References

[1] Apache hadoop. `http://hadoop.apache.org`.

[2] Trusted computing group. `www.trustedcomputinggroup.org`.

[3] R. Agrawal, J. Kiernan, R. Srikant, and Y. Xu. Order preserving encryption for numeric data. In *SIGMOD*, pages 563–574, 2004.

[4] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan. Data center tcp (dctcp). In *SIGCOMM*, 2010.

[5] T. Alves and D. Felton. Trustzone: integrated hardware and software security. AMD white paper, 2004.

[6] A. Arasu, S. Blanas, K. Eguro, M. Joglekar, R. Kaushik, D. Kossmann, R. Ramamurthy, P. Upadhyaya, and R. Venkatesan. Secure database-as-a-service with cipherbase. In *SIGMOD*, pages 1033–1036, 2013.

[7] S. Bajaj and R. Sion. TrustedDB: a trusted hardware based database with privacy and data confidentiality. In *SIGMOD*, pages 205–216, 2011.

[8] A. Baumann, M. Peinado, and G. Hunt. Shielding applications from an untrusted cloud with haven. In *OSDI*, 2014.

[9] M. Blanton, A. Steele, and M. Alisagari. Data-oblivious graph algorithms for secure computation and outsourcing. In *ASIACCS*, pages 207–218. ACM, 2013.

[10] J. Camenisch and A. Mityagin. Mix-network with stronger security. In *Privacy Enhancing Technologies*, 2006.

[11] R. Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *IEEE Symposium on Foundations of Computer Science*, 2001.

[12] D. Champagne and R. B. Lee. Scalable architectural support for trusted software. In *HPCA*, 2010.

[13] D. L. Chaum. Untraceable electronic mail, return addresses, and digital pseudonyms. *Communications of the ACM*, 1981.

[14] X. Chen, T. Garfinkel, E. Lewis, P. Subrahmanyam, C. Waldspurger, D. Boneh, J. Dwoskin, and D. Ports. Overshadow: a virtualization-based approach to retrofitting protection in commodity operating systems. In *ASPLOS*, pages 2–13, 2008.

[15] R. Curtmola, J. A. Garay, S. Kamara, and R. Ostrovsky. Searchable symmetric encryption: improved definitions and efficient constructions. In *ACM CCS'06*, 2006.

[16] G. Danezis and C. Diaz. A survey of anonymous communication channels. Technical report, Technical Report MSR-TR-2008-35, Microsoft Research, 2008.

[17] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. In *OSDI*, 2014.

[18] J. Demme, R. Martin, A. Waksman, and S. Sethumadhavan. Side-channel vulnerability factor: a metric for measuring information leakage. In *ISCA*, 2012.

[19] R. Dingledine. Anonymity bibliography. http://freehaven.net/anonbib/.

[20] C. Gentry. Fully homomorphic encryption using ideal lattices. In *ACM Symposium on Theory of Computing*, May-June 2009.

[21] C. Gentry and S. Halevi. A working implementation of fully homomorphic encryption. In *EUROCRYPT*, 2010.

[22] O. Goldreich and R. Ostrovsky. Software protection and simulation on oblivious rams. *Journal of the ACM*, 1996.

[23] M. T. Goodrich and M. Mitzenmacher. Privacy-preserving access of outsourced data via oblivious ram simulation. In *ICALP*, 2011.

[24] J. A. Halderman, S. D. Schoen, N. Heninger, W. Clarkson, W. Paul, J. A. Calandrino, A. J. Feldman, J. Appelbaum, and E. W. Felten. Lest we remember: cold-boot attacks on encryption keys. *Communications of the ACM*, 52(5):91–98, 2009.

[25] S. Huang, J. Huang, Y. Liu, L. Yi, and J. Dai. Hibench: a representative and comprehensive hadoop benchmark suite. In *ICDE workshops*, 2010.

[26] M. Isard, M. Budiu, Y. Y. andAndrew Birrell, and D. Fetterly. Dryad: Distributed data-parallel programs from sequential building blocks. In *Eurosys*, 2007.

[27] D. Jiang, G. Chen, B. C. Ooi, K.-L. Tan, and S. Wu. epic: an extensible and scalable system for processing big data. In *VLDB*, 2014.

[28] J. Katz and Y. Lindell. *Introduction to modern cryptography*. CRC Press, 2014.

[29] T. Kim, M. Peinado, and G. Mainar-Ruiz. Stealthmem: system-level protection against cache-based side channel attacks in the cloud. In *USENIX Security*, 2012.

[30] M. Klonowski and M. Kutyłowski. Provable anonymity for networks of mixes. In *Information Hiding*, pages 26–38. Springer, 2005.

[31] F. Li, B. C. Ooi, M. T. Ozsu, and S. Wu. Distributed data management using mapreduce. *ACM Computing Survey*, 46(6), 2014.

[32] Y. Li, J. McCune, J. Newsome, A. Perrig, B. Baker, and W. Drewry. Minbox: a two-way sandbox for x86 native code. In *USENIX ATC*, 2014.

[33] C. Liu, M. Hicks, A. Harris, M. Tiwari, M. Maas, and E. Shi. Ghostrider: A hardware-software system for memory trace oblivious computation, 2015.

[34] C. Liu, M. Hicks, and E. Shi. Memory trace oblivious program execution. In *IEEE CSF*, pages 51–65. IEEE, 2013.

[35] C. Liu, Y. Huang, E. Shi, J. Katz, and M. Hicks. Automating efficient ram-model secure computation. In *Security and Privacy (SP), 2014 IEEE Symposium on*, pages 623–638. IEEE, 2014.

[36] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. Distributed graphlab: A framework for machine learning and data mining in the cloud. In *VLDB*, 2012.

[37] L. Luu, S. Shinde, P. Saxena, and B. Demsky. A model counter for constraints over unbounded strings. In *PLDI*, page 57, 2014.

[38] J. M. McCun, B. Parno, A. Perrig, M. K. Reiter, and H. Isozaki. Flicker: An execution infrastructure for tcb minimization. In *EuroSys*, 2008.

[39] J. McCune, Y. Li, N. Qu, Z. Zhou, A. Datta, V. Gligor, and A. Perrig. Trustvisor: Efficient tcb reduction and attestation. In *IEEE Symposium on Security and Privacy*, pages 143–158, 2010.

[40] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagaonkar. Innovative instructions and software model for isolated execution. In *HASP*, 2013.

[41] D. Molnar, M. Piotrowski, D. Schultz, and D. Wagner. The program counter security model: Automatic detection and removal of control-flow side channel attacks. In *Information Security and Cryptology-ICISC 2005*, pages 156–168. Springer, 2006.

[42] 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.

[43] O. Ohrimenko. *Data-oblivious algorithms for privacy-preserving access to cloud storage*. PhD thesis, Brown University, 2014.

[44] L. Page, S. Brin, R. Motwani, and T. Winograd. The pagerank citation ranking: bringing order to the web. Technical report, Stanford InfoLab, 1999.

[45] P. Paillier. Public-key cryptosystems based on composite degree residuosity classes. In *EUROCRYPT*, May 1999.

[46] R. A. Popa, F. H. Li, and N. Zeldovich. An ideal-security protocol for order-preserving encoding. In *IEEE Symposium on Security and Privacy*, pages 463–477, 2013.

[47] R. A. Popa, C. M. S. Redfield, N. Zeldovich, and H. Balakrishnan. Cryptdb: Protecting confidentiality with encrypted query processing. In *SOSP*, 2011.

[48] I. Roy, S. T. V. Setty, A. Kilzer, V. Shmatikov, and E. Witchel. Airavat: Security and privacy for mapreduce. In *NSDI*, 2010.

[49] F. Schuster, M. Costa, C. Fournet, C. Gkantsidis, M. Peinado, G. Mainar-Ruiz, and M. Russinovich. Vc3: Trustworthy data analytics in the cloud. Technical report, Microsoft Research, 2014.

[50] A. Seshadri, M. Luk, N. Qu, and A. Perrig. Secvisor: a tiny hypervisor to provide lifetime kernel code integrity for commodity oses. In *SOSP*, pages 335–50, 2007.

[51] S. Shinde, Z. L. Chua, V. Narayanan, and P. Saxena. Preventing your faults from telling your secrets: Defenses against pigeonhole attacks. *CoRR*, abs/1506.04832, 2015.

[52] S. Shinde, S. Tople, D. Kathayat, and P. Saxena. Podarch: Protecting legacy applications with a purely hardware tcb. Technical report, National University of Singapore, 2015.

[53] V. Shmatikov and M.-H. Wang. Measuring relationship anonymity in mix networks. In *ACM workshop on Privacy in electronic society*, pages 59–62. ACM, 2006.

[54] D. X. Song, D. Wagner, and A. Perrig. Practical techniques for searches on encrypted data. In *IEEE Symposium on Security and Privacy*, May 2000.

[55] E. Stefanov, M. van Dijk, E. Shi, C. Fletcher, L. Ren, X. Yu, and S. Devadas. Path oram: an extremely simple oblivious ram protocol. In *CCS*, 2013.

[56] U. Steinberg and B. Kauer. Nova: a microhypervisor-based secure virtualization architecture. In *Eurosys*, 2010.

[57] J. Szefer and R. B. Lee. Architectural support for hypervisor-secure virtualization. In *ASPLOS*, 2012.

[58] S. Tople, S. Shinde, Z. Chen, and P. Saxena. AUTOCRYPT: enabling homomorphic computation on servers to protect sensitive web content. In *ACM CCS*, pages 1297–1310, 2013.

[59] S. Tu, M. F. Kaashoek, S. Madden, and N. Zeldovich. Processing analytical queries over encrypted data. In *VLDB*, 2013.

[60] D. Wikström. A universally composable mix-net. In *Theory of Cryptography*. 2004.

[61] C. Wilson, H. Ballani, T. Karagiannis, and A. Rowtron. Better never than late: meeting deadlines in datacenter networks. In *SIGCOMM*, 2011.

[62] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed dataset: A fault-tolerant abstraction for in-memory cluster computing. In *NSDI*, 2012.

[63] F. Zhang, J. Chen, H. Chen, and B. Zang. Cloudvisor: retrofitting protection of virtual machines in multi-tenant cloud with nested virtualization. In *SOSP*, pages 203–216, 2011.

[64] H. Zhang, G. Chen, B. C. Ooi, K.-L. Tan, and M. Zhang. In-memory big data management and processing: a survey. *TKDE*, 27(7):1920–1948, 2015.

[65] Z. Zhou, J. Han, Y.-H. Lin, A. Perrig, and V. Gligor. Kiss: "key it simple and secure" corporate key management. In *TRUST*, 2013.

Appendix A Security Analysis

Proof (Lemma 1):

Consider the "ideal mixer" that takes as input a sequence $\langle x_1,\ldots,x_N\rangle$ where each $x_i \in [1,N]$, picks a permutation $p:[1,N]\to[1,N]$ randomly and then output the sequence $\langle x_{p(1)},x_{p(2)},\ldots,x_{p(N)}\rangle$. Klonowski et al. [30] investigated the effectiveness of the cascaded network of mixing, and showed that $O(\log\frac{N}{T})$ steps are suffice to bring the distribution of the mixed sequence statistically close to the output of the ideal mixer, where T is the number of items an instance can process in memory. Our proof relies on the above-mentioned result.

Let us assume that κ, the number of steps carried out by cascaded-mix, is sufficiently large such that the distribution of the mixed sequence is statistically close to the ideal mixer.

Consider an adversary $\mathscr{S}$ that executes the cascaded-mix. Let us construct an adversary $\mathscr{A}$ who simulates $\mathscr{S}$ but only has access to Ω. To fill in the tuple values not present in Ω, the simulation simply fills in random tuples. Note that the number of tuples can be derived from Ω.

Now, suppose that on input $x_1,\ldots,x_N$, the output of $\mathscr{A}$ and $\mathscr{S}$ can be distinguished by $\mathscr{D}$. We want to show that this contradicts the semantic security of the underlying encryption scheme, by constructing a distinguisher $\widetilde{\mathscr{D}}$ who can distinguish multiple ciphertexts from random with polynomial-time sampling (i.e. the distinguisher sends the challenger multiple messages, and receive more than one sample).

Let $z=\langle z_1,z_2,\ldots,z_N\rangle$ be the output of the mixer on input $x_1,\ldots,x_N$. The distinguisher $\widetilde{\mathscr{D}}$ asks the challenger for a sequence of ciphertexts of z. Let $c_{i,j}$'s be the ciphertexts returned by the challenger, where $c_{i,j}$ is the i-th ciphertexts of z_j. To emulate $\mathscr{S}$, likewise, $\widetilde{\mathscr{D}}$ needs to feed the simulation with the intermediate data generated by `mixT`. Let $y_{i,j}$ be the i-th intermediate ciphertext in round j the distinguisher $\widetilde{\mathscr{D}}$ generated for the emulation. The $y_{i,j}$'s are generated as follow:

1. $\widetilde{\mathscr{D}}$ simulates the cascaded-mix by randomly picking a permutation for every `mixT`. Let $p_j:[1,N]\to[1,N]$ be the overall permutation for round j. Let $\hat{p}_j$ be the permutation that moves the i-th ciphertext in the input, to its location after j rounds. That is, $\hat{p}_j(i)=p_j(\hat{p}_{j-1}(i))$, and $\hat{p}_0(i)=i$.
2. Set $y_{i,j}=c_{\hat{p}_j(i),j}$ for each i,j.

Let v be the output of $\mathscr{D}$'s simulation. Note that if $x_{i,j}$'s are random ciphertexts, then the distribution of v is the same as the output distribution of $\mathscr{A}$. On the other hand, if $x_{i,j}$'s are ciphertexts of z, then the input to the emulation is statistically close to the input of $\mathscr{S}$, and thus distribution of v is statistically close to the output distribution of S.

Since $\mathscr{D}$ can distinguish output of $\mathscr{S}$ from $\mathscr{A}$'s, $\widetilde{\mathscr{D}}$ can distinguish the ciphertexts of z from random. □