Privacy-Preserving Data Management for Outsourced Databases

Hung Dang, Anh Dinh, Ee-Chien Chang, Beng Chin Ooi, Shruti Tople, Prateek Saxena
School of Computing, National University of Singapore

ABSTRACT

Cloud providers are realizing the outsourced database model in the form of database-as-a-service offerings. Security in terms of data privacy remains an obstacle because data storage and processing are done on an untrusted cloud. As such, providing a strong notion of security under additional constraints of functionality and performance is challenging, for which advanced encryption and recent trusted computing primitives alone prove insufficient.

This paper proposes a practical system for privacy-preserving data management, called PRAMOD, in which data is stored in encrypted form and data-dependent computations are carried out inside a trusted environment. The system supports popular algorithms underlying many data management applications, including sort, compaction, join and group aggregation. Data privacy is ensured even when data movement between different components (caused by limited private memory) is observed by the adversary. For many algorithms, this is achieved by appending a component called *scrambler* which breaks the linkage between the input and output. Our experimental study indicates reasonable overheads over a baseline system with a weaker level of security. In addition, PRAMOD shows better performance than state-of-the-art solutions with similar levels of security. For example, PRAMOD achieves 4.4× speedup over the alternative data-oblivious sorting algorithm.

1. INTRODUCTION

Big data is the driving force behind the database-as-a-service model offered by incumbent cloud providers. Amazon, Google, Microsoft, etc. are providing cost-effective and scalable solutions for storing and managing tremendous volumes of data. However, security in terms of data privacy remains a challenge, as the data is being handled by an untrusted party. Despite being a well-studied problem in the context of outsourced database in the past [40, 41, 33], data privacy in the big data era faces new challenges. First, the cloud providers have more incentives in learning and mining content of the outsourced data because it contains commercial values [44, 45]. Second, even when the providers are trusted, multi-tenancy, complexity of the software stack, and the distributed computing model contribute the large attack surface [16, 18]. Third, there is a tight constraint on the performance overhead, since most data analytics tasks, e.g. data mining, consume huge CPU cycles which are directly billable [17, 10].

The first step toward securing the data is to encrypt it before outsourcing to the cloud, but this only protects data at rest [40, 33]. To enable more operations over encrypted data, one option is to use fully homomorphic encryption, but existing schemes are not yet practical [20, 14]. Another option is to employ partially homomorphic encryption schemes [38, 19] which are more practical but limited in the range of supported operations [41, 46]. Recent works have advocated combining encryption with trusted computing primitives [11, 29, 42, 47, 8, 10]. Confidentiality and integrity protected execution environments are provisioned by hardware (IBM secure coprocessors [2], Intel SGX [3]) or by hardware-software combination [34, 35]. In these environments, the computations are performed on decrypted data, and the results are encrypted before returned. However, there is a limit on the amount of data such a secure environment can process at any time (upper bound being the size of physical memory allocated to a process), giving rise to a communication channel between the trusted environment and the untrusted storage. This channel often can leak information about the data [43, 16, 18]. For instance, by observing I/O access patterns (or access patterns for short) during merge-sort, an attacker can infer the order of the original input. Such leakage can be eliminated by either generic oblivious-RAM (ORAM) or operation-specific data-oblivious algorithms[1] [39, 25, 43]. However, both approaches are complex and incur high performance overheads. Moreover, it is also worth noting that since vetting software for implicit vulnerabilities is non-trivial and expensive, keeping the codebase which resides in the secure environment simple is essential for overall security of the system. Unfortunately, ORAM approach and operation-specific data-oblivious algorithms are often convoluted.

In this work, we aim to enable practical, privacy-preserving data management. In particular, the data management algorithms running on the untrusted cloud leak no information about the inputs via access patterns, while

[1]An algorithm is data-oblivious if once the input and output sizes are fixed, it will perform a similar sequence of operations on all inputs

incurring reasonable performance overhead. We consider two basic algorithms, namely sort and compaction, and two complex algorithms based on sort and compaction, namely group aggregation and join. These algorithms underlie many data management applications. Sort is fundamental to any database system. Compaction is vital in many distributed key-value stores where updates are directly appended to disk and compaction process is frequently scheduled to improve query performance [7, 32, 6]. Join is a common algorithm for data integration which is becoming more important given the variety of data sources [27]. Group aggregation is widely used in decision support systems to summarize data, making it an integral part of data warehouse systems. These last two algorithms account for a majority of queries in the TPC benchmarks: 80/99 join queries in TPC-DS and YY aggregation queries in TPC-DI.

We make two observations in this paper. The first observation is that, in order to prevent leakage from access patterns for sort and compaction, it is sufficient to randomly scramble (or permute) the input before feeding it to the actual algorithm. For example, consider a merge-sort algorithm where the original input is randomly scrambled. An adversary observing access patterns will be able to infer only the order of the scrambled input, which says nothing about the order of the original input. This approach to security — scrambling the input before executing the algorithm — leads to two important results. First, it is superior in performance compared with generic ORAM solutions, because the overhead factor is additive rather than multiplicative. Second, this approach generalizes to all algorithms implementing the same operation, hence we can take advantage of state-of-the-arts algorithms to achieve simpler solution with better performance than existing data-oblivious algorithms. For example, scrambling followed by an optimized merge sort (or any other popular sort algorithms) is simpler than existing data-oblivious external sort algorithms [25], and we show later that the former indeed has better performance. We note that the simplicity of this approach implies smaller TCB which translates to better security. The second observation is that, for a complex algorithm made up of a sequence of sub-steps, there will be no leakage via access patterns if the sub-steps themselves do not leak information via access patterns. We use this observation to achieve security for group aggregation and join algorithm by implementing them based on sort and compaction.

We design a system called PRAMOD (PRivate dAta Management for Outsourced Databases), in which user data is encrypted with an authenticated encryption scheme, and data-dependent computations are to be performed in trusted computation units (or trusted units). Data movements and other house keeping tasks are done by an untrusted worker. We assume that trusted units are securely provisioned either by pure hardware or a hardware-software combination. PRAMOD consists of a component called *scrambler* which randomly and securely permutes data. For input of size n, the scrambler runs in $O(n)$ and requires private memory of size $O(\sqrt{n})$. It is prepended to the traditional, not necessarily privacy-preserving sort and compaction algorithms to make them privacy-preserving. The latter are then composed with other privacy-preserving steps to realize group aggregation and join. We implement the four algorithms in PRAMOD to evaluate their performances and costs of security. Compared with a baseline system offering a weaker level of security, the overhead added by PRAMOD is $2.43\times - 4.99\times$. We also compare PRAMODagainst state-of-the-art data-oblivious algorithms [25, 22, 8, 9] which offer similar level of security. The results demonstrate that our implementations achieve speedup as high as $4.4\times$. In summary, we make the following contributions:

1. We define a security model for privacy-preserving data management algorithms. The model implies data confidentiality even when the adversary can observe I/O access patterns.

2. We propose a system — PRAMOD — for implementing algorithms that meet our security definition. Certain classes of algorithms achieve security by simply adding a scrambling step. Other more complex algorithms achieve security by decomposing into smaller, privacy-preserving substeps.

3. We describe the implementation of four algorithms in PRAMOD, namely: pSort, pCompact, pAgregation and pJoin. We find that they are efficient. Especially, pCompact achieves $O(n)$ running time, outperforming the state-of-the-art data-oblivious compaction algorithm which runs in $O(n \log n)$.

4. We benchmark these implementations against the baseline and state-of-the-art implementations. Experiment results demonstrate reasonable security overheads over the less secure (baseline) implementations, and running time speedup of $4.4\times$, $3.5\times$, $3.8\times$ and $2.6\times$ over ones with similar level of security.

The rest of this paper is organized as follows. Section 2 discusses the system, security model and the problem being addressed. Section 3 describes the design of PRAMOD, followed by details of the four algorithms in Section 4. Section 5 reports our experimental evaluation of PRAMOD. We discuss related works in Section 6 before concluding in Section 7.

2. PROBLEM DEFINITION

Our goal is to enable privacy-preserving data management algorithms on outsourced data with trusted computing. This section discusses the scope in detail. We present a simplified, running example to illustrate our ideas, and describe a baseline system to help motivate our work.

Running Example. We consider a user storing her data consisting of integer-value records on the cloud. The user then wishes to sort her data, as a pre-processing step for other tasks such as database loading, ranking, deduplication, etc. More often than not, the user will encrypt her data so that no untrusted party can learn its content. Although sorting encrypted data is possible to a certain extent, it is highly impractical [5]. Thus, to efficiently sort the data, the user must delegate the decryption capability to a trusted unit, which decrypts the records and sorts them in its secure memory before re-encrypting and sending them back to the cloud storage. Because secure memory is limited in size, the cloud provider performs sorting using external, k-way merge sort algorithm. Figure 1 depicts a simple example of 3-way merge sort in which the secure memory is limited to holding only 3 records at a time. The input consists of 9 records, and sorting involves 1 merging step.

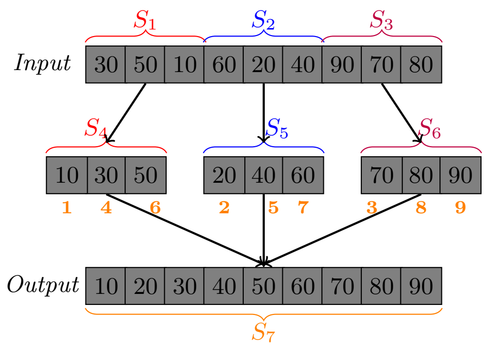


Figure 1: An example of 3-way external merge-sort. The records are encrypted (filled objects), black numbers represent integer-value of the records while orange numbers denote the order in which each encrypted records is read into the trusted unit during the merging.

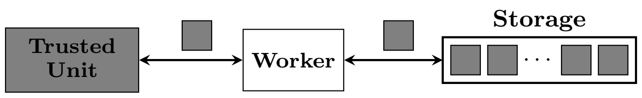


Figure 2: Computation model of a cloud server, consisting of a trusted unit capable of processing a limited of number of records at a time. Storage is untrusted, and its communication with the trusted unit is mediated by an untrusted worker (honest-but-curious). Only the trusted unite can see and compute the content of the encrypted records (filled squares).

2.1 Baseline System and Adversary Model

We consider a system in which the user uploads data to the cloud and relies entirely on the latter to store and carry out computations on her data. The data $X = \langle x_1, x_2, \ldots, x_n \rangle$ consists of a sequence of n equal-sized, key-value records. We use $key(x)$ and $value(x)$ to denote the key and value component of record x, respectively. In this system, data records are protected by a semantically secure encrypted scheme. The cloud server's computation model is shown in Figure 2. It consists of a *trusted unit*, a *worker* and a storage component. The trusted unit can hold up to $m = O(\sqrt{n})$ records and some constant number of control variables at any time, and it is trusted by the user. Read and write access to the untrusted storage are mediated by the worker which is also responsible for other house-keeping tasks. Both the worker and storage component see only encrypted data.

Threat Model. The adversary is a malicious insider at the cloud provider, who has complete access to the cloud infrastructure, either via misuses of privilege or via exploiting vulnerabilities in the software stack. This work considers honest-but-curious (or passive) attacker which tries to learn information from what observable but does not tamper with the data or the computation. This is a realistic model, given that insider threats are a serious concern to organizations as they are one of main causes of security breaches [?]. Active attackers who deviate from the expected computation are out of scope, but we refer readers to recent works demonstrating effective defense against this stronger model [18, 42].

We assume the worker and storage component are under the attacker's control, but the trusted unit are sufficiently

Algorithm	*Baseline*	PRAMOD	*Oblivious Algorithms*
Sort	Order of original input	Input & Output sizes	Input & Output sizes
Compaction	Distribution of removed records		
Group aggregation	Distribution of original input		
Join	Distribution of original input		

Table 1: Leakage of PRAMOD's algorithms, compared with that of the baseline system and relevant oblivious algorithms

protected. Specifically, the trusted unit corresponds to the user's trusted computing base (TCB), the worker and storage component correspond to the cloud software stack and storage controller. We exclude physical attacks that could compromise the trusted unit's confidentiality and integrity, such as cold-boot or attacks subverting the CPU security mechanisms. For TCB based on hardware-software combination, we assume that the software is void of vulnerabilities and malwares. Furthermore, there is no side-channel leakage (via cache or power analysis) from the trusted unit. Finally, we assume that decryption keys have already been delivered securely to the trusted unit; details of the specific key provisioning scheme is beyond our scope.

Leakage of the Baseline System. Let us use the running example to demonstrate how the baseline system fails to ensure data privacy. Figure 1 shows the encrypted input divided into three blocks. In the first round, the trusted unit independently sorts each block in-memory and returns three sorted, encrypted blocks. Next, 3-way merging is performed: three records are kept in the memory at a time and new encrypted records are pulled, with help from the worker, from the sorted blocks. However, the adversary observes that the trusted unit first takes one record from each sorted block, writes 1 record out, then takes another record from the first block. Hence, it can infer that the smallest record comes from S_1. In general, such inference can reveal relative order of records from different blocks. For algorithms taking data from different sources, this leakage can expose the source's identity.

2.2 Problem Overview

This paper concerns two basic algorithms — sort and compaction — and others based on these algorithms. Let $\mathcal{P}$ be the algorithm over input X, our first goal is to restrict leakage from the execution of $\mathcal{P}$ to only input and output sizes, i.e. $|X|$ and $|\mathcal{P}(X)|$. The baseline system fails this goal, as summarized in Table 1. One solution is to employ oblivious RAM [43] directly as the storage backend. However, the overhead per read/write request is at least $O(\log n)$, rendering it impractical for big data processing. Another option is to use application-specific algorithms such as data-oblivious sort [25], but they are convoluted and do not generalize well to other algorithms. Thus, our second goal is to attain a simple design with low performance overhead.

2.3 Security Definition

We now describe the formal security definition that allows the adversary to learn only the input and output sizes. Let $Q_{\mathcal{P}}(X) = \langle q_1, q_2, \ldots, q_z \rangle$ be the read/write sequence observed by the adversary during the execution of the operation $\mathcal{P}$ on input X. In the baseline system, $Q_{\mathcal{P}}(X)$ represents I/O requests made by the trusted units to the worker. Each q_i is a 4-value tuple $\langle$*ops, addr, time, info*$\rangle$ where *ops*

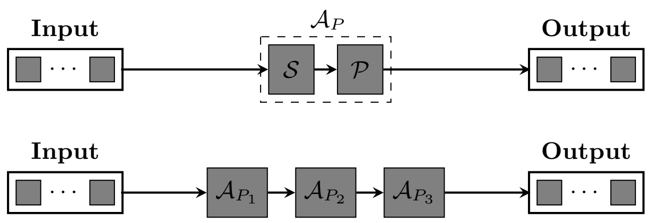


Figure 3: PRAMOD constructs privacy-preserving algorithm $\mathcal{A}_P$ from an algorithm $\mathcal{P}$ by appending it with the scrambler $\mathcal{S}$. PRAMOD can also combine simple privacy-preserving algorithms to build a more complex algorithm which is also privacy-preserving.

is the type of the request ("read" or "write"), *addr* is the address accessed by *ops*, *time* is the time of request and *info* is the record's metadata. The last component is useful when the trusted unit offloads parts of the processing on non-sensitive data fields to the worker. For example, in secondary sorting operation where the secondary sort keys are non-sensitive, after sorting encrypted records on the primary keys, the trusted unit sets *info* in each record to be the secondary sort key, thus allowing the worker to complete the sorting operation.

During the execution of $\mathcal{P}$, $Q_{\mathcal{P}}(X)$ is the only source of leakage from which the attacker can learn information about X. Our security definition dictates that $Q_{\mathcal{P}}(X)$ reveals nothing about X besides its output sizes $|\mathcal{P}(X)|$. Specifically:

Definition 1 (Privacy-Preserving Operation). *An operation $\mathcal{P}$ is privacy-preserving if for any two datasets X_1, X_2 of size n and $|\mathcal{P}(X_1)| = |\mathcal{P}(X_2)|$, the $Q_{\mathcal{P}}(X_1)$ is computationally indistinguishable from $Q_{\mathcal{P}}(X_2)$.*

Informally, the definition says that for any two inputs which are of the same size and induce outputs also of the same size, the operation is privacy-preserving if the observed I/O sequences are *similar.* No computationally bounded adversary can distinguish the two I/O sequences, thus the observed execution does not reveal any information about the input.

Discussion. We note that a similar security definition, *data-obliviousness*, theoretically provides stronger security than ours [22, 12]. $\mathcal{P}$ is data-oblivious (or oblivious for short) if the observed I/O sequences are the same for X_1 and X_2, i.e. $Q_{\mathcal{P}}(X_1) = Q_{\mathcal{P}}(X_2)$ (where $|X_1| = |X_2|$ and $|\mathcal{P}(X_1)| = |\mathcal{P}(X_2)|$). This definition implies perfect zero leakage against all adversaries, as opposed to ours which guarantees negligible leakage against computationally bounded adversaries. However, we remark that both definitions assume that the adversary does not know the content of the record as they are encrypted, hence the overall security relies on security of the encryption scheme. Since practical encryption schemes fail to achieve perfect secrecy, it is reasonable to expect that, in practice, a privacy-preserving operation with respect to Definition 1 and a data-oblivious operation offer the same level of security.

3. DESIGN

This section describes PRAMOD's design, focusing on the new component added to the baseline system — the scrambler. We explain how to design privacy-preserving primitives, namely sort and compaction, using the scrambler. Later, we discuss how to construct the more complex algorithms using these two primitives.

PRAMOD is built from the baseline system described earlier in Section 2.1. Specifically, data records are encrypted with a semantically secure scheme, and data-dependent computations are done inside the trusted unit with limited memory. Algorithms implemented in the baseline system may not be privacy preserving. For example, implementing merge sort in the baseline system reveals the input order. In PRAMOD, privacy-preserving algorithms are realized in two ways. First, for algorithms which essentially re-arranges the input, PRAMODcan achieve our security definition by prepending it with a scrambling step (Figure 3[a]). This step, implemented by the scrambler, randomly permutes the input before feeding it to the traditional non-privacy-preserving algorithm. It essentially breaks the linkage between input and output, therefore preventing leakage during execution of the algorithm. To ensure overall security, however, there must also be no leakage during the scrambling step.

Second, a complex algorithm which involves more data processing than simple rearrangement can be decomposed to a sequence of sub-steps $\mathcal{A}_{P_1}, \mathcal{A}_{P_2}, \ldots,$ all of which are privacy-preserving (Figure 3[b]). We remark that for complex algorithms, input scrambling alone is not sufficient to render them privacy-preserving. For instance, consider group aggregation algorithm which first rearranges data into groups and then aggregates the values of each group. Even if the inputs are scrambled beforehand, the adversary can still learn information about group sizes, which is considered as a leakage since it is beyond what admitted by definition 1 (the number of groups).

3.1 The Scrambler

The goal of the scrambler is to privately realize a permutation $\pi : [1..n] \rightarrow [1..n]$, which can be generated by a pseudo-random permutation [28] and represented by a short secret seed, on the input X, obtaining $\widetilde{X}$ such that $\widetilde{X}[j] = X[\pi[i]]$. In particular, π is uniformly chosen at random, and the linkage between $X[i]$ and $\widetilde{X}[j] = X[\pi[i]]$ cannot be learned from observing I/O patterns during the execution. In other words, the scrambler ensures that the adversary is oblivious to the underlying permutation π. A simple solution which sequentially scans through X and places the i_{th} record at position $\pi[i]$ in $\widetilde{X}$ reveals π, because the adversary is able to observe all read/write accesses on the storage.

The scrambler can be implemented using the *shuffle* algorithm recently introduced in [36] or a *cascaded mix-network* proposed in [15]. There are two key differences between these approaches. First, the shuffle algorithm takes as input X and π and outputs $\widetilde{X}$, whereas the cascaded mix-network takes as input only X and generates $\widetilde{X}$ without a priori knowledge of the permutation π. The output space of the former is of size $(1-\epsilon) \times n!$ where ϵ is a negligible function, while that of the latter is much smaller: $(1-\epsilon') \times n!$ where ϵ' is a non-negligible function [30]. Thus, the shuffle algorithm offers higher privacy protection than mix-network. Second, the former's time complexity and trusted memory requirement are $O(n)$ and $O(\sqrt{n})$ respectively, which is lower than those of the latter: $O(n \log n)$ and $O(n^{0.85})$ respectively.

We implement the Melbourne shuffle algorithm [36] using the trusted unit. The algorithm consists of two phases: distribution and clean-up phase. The overall performance and probability of the algorithm aborting due to overflow of memory segments depends on two variables p_1 and p_2. Larger values of p_1 and p_2 lead to higher running time but lower probability of aborting. We refer readers to Appendix A for more details.

3.2 Privacy-Preserving Algorithms

We first show how to construct basic privacy-preserving algorithms using the scrambler. Next, we discuss how to build more complex ones.

3.2.1 Basic algorithms

We consider algorithms which essentially rearrange the input. Specifically, given the output $\mathcal{P}(X) = Y = \langle y_1, y_2, .., y_n \rangle$, the algorithm can be characterized by a permutation (or tag) $T = \langle t_1, t_2, .., t_n \rangle$ such that $Y[i] = X[T[i]]$. In other words, the tag T represents the linkage between input and output records. For example, in the example in Figure 1, $T = \langle 3, 5, 1, 6, 2, 4, 9, 7, 8 \rangle$. Let $Q_{\mathcal{P}}(X)$ be the observed read/write sequence defined earlier in Section 2.3. Let $\mathcal{A}_P$ be the algorithm derived from $\mathcal{P}$ by prepending it with the scrambler. We show that if $Q_{\mathcal{P}}(X) = Q_{\mathcal{P}}(T)$, then $\mathcal{A}_P$ is privacy preserving.

THEOREM 1. *Given any input X and its corresponding tag T, if an algorithm $\mathcal{P}$, when being computed on X and T, generate the same read/write sequences (i.e. $Q_{\mathcal{P}}(T) = Q_{\mathcal{P}}(X)$), then the derived algorithm $\mathcal{A}_P$, which first has the scrambler permute the input and then applies $\mathcal{P}$ on the scrambled input, is privacy-preserving.*

Proof Sketch: We consider two inputs X_1, X_2 of the same size whose outputs computed by a $\mathcal{P}$ are also of the same size (i.e. $|\mathcal{P}(X_1)| = |\mathcal{P}(X_2)|$). We denote by T_1, T_2 tags of X_1, X_2, and by $\widetilde{T}_1$, $\widetilde{T}_2$ the tags of the scrambled input $\widetilde{X}_1, \widetilde{X}_2$, respectively. The property of the scrambler assures that the distributions of $\widetilde{T}_1$ and $\widetilde{T}_2$ are indistinguishable. This, together with the fact that the scrambler obliviously transforms T_1 to $\widetilde{T}_1$ and T_2 to $\widetilde{T}_2$, imply that $Q_{\mathcal{P}}(T_1)$ is indistinguishable from $Q_{\mathcal{P}}(T_2)$. Moreover, since $Q_{\mathcal{P}}(T_1) = Q_{\mathcal{P}}(X_1)$ and $Q_{\mathcal{P}}(T_2) = Q_{\mathcal{P}}(X_2)$, it follows that $Q_{\mathcal{P}}(X_1)$ and $Q_{\mathcal{P}}(X_2)$ are also indistinguishable. Therefore, $\mathcal{A}_{\mathrm{P}}$ is privacy-preserving.

□

3.2.2 Complex algorithms

We consider a complex algorithm which can be decomposed into a sequence of substeps. Specifically, let $\mathcal{A}_P = (\mathcal{A}_{P_1}; \mathcal{A}_{P_2}; ..; \mathcal{A}_{P_m})$ be the algorithm consisting of m substeps, in which the output of $\mathcal{A}_{P_i}$ is the input of $\mathcal{A}_{P_{i+1}}$. It is trivial that if every sub-step $\mathcal{A}_{P_i}$ is privacy-preserving, then so is $\mathcal{A}_P$. It is derived from the following theorem (we omit the proof due to space constraint).

THEOREM 2. *Given two algorithms $\mathcal{P}_1, \mathcal{P}_2$ which are privacy preserving. The combined algorithm $\mathcal{P}$ which executes $\mathcal{P}_1$ followed by $\mathcal{P}_2$ is also privacy preserving.*

3.3 Discussion

Algorithm	*Baseline*	*Data-oblivious*	PRAMOD
Sort	$O(n \log n)$	$O(n \log^2 n)$	$O(n \log n)$
Compaction	$O(n)$	$O(n \log n)$	$O(n)$
Group aggregation	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
Join	$O(n_1 \log n_1 + n_2 \log n_2)$	$O(n_1 \log n_1 + n_2 \log n_2 + l \log l)$	$O(n_1 \log n_1 + n_2 \log n_2 + l \log l)$

Table 2: Comparison of time complexities of different implementations. For join algorithm, l is the size of the result join set.

One important implication of Theorem 1 is that the output of $\mathcal{A}_P$ is not always the same as that of $\mathcal{P}$ because the input has been permuted. For example, consider merge sort algorithm on $X = \langle 0_0, 0_1, 0_2, 0_3, 0_4, 0_5 \rangle$ where the subscripts indicate the original positions in the input. The output $\mathcal{P}(X) = \langle 0_0, 0_3, 0_1, 0_4, 0_2, 0_5 \rangle$, whereas $\mathcal{A}_P(X) = \langle 0_0, 0_2, 0_1, 0_5, 0_3, 0_4 \rangle$ for a certain permutation generated by the scrambler. We note that to achieve the output consistency, it is sufficient to make $\mathcal{P}$ invariant to input permutation, that is $\mathcal{P}(X) = \mathcal{P}(X')$ where X' is a permutation of X. In practice, this can be achieved by adding a pre-processing step which transform the inputs, and a post-processing step to reverse the effect. In the sort example, the pre-processing step adds metadata to the keys so that the input contains no duplicates (for instance, by using the record address as the secondary key), and the post-processing step removes the metadata.

The algorithms considered so far are deterministic. However, Theorem 1 also generalizes to probabilistic algorithms (quick sort, for example). Essentially, we transform these algorithms to take the random choices as additional input, thus making them deterministic and to which the theorem is applicable.

We remark that there exists many privacy-preserving algorithms, which makes it possible to implement complex algorithms that meet our security definition. As noted earlier, scanning algorithms which scan through the input and write the output to the same address are inherently privacy-preserving. Existing data-oblivious algorithms proposed in the literature, such as oblivious data expansion [9], are also privacy-preservingThese algorithms can be ported directly to PRAMOD. In fact, for complex algorithms for which data-oblivious implementations exist, we can re-use these implementations directly by replacing data-oblivious substeps with more efficient privacy-preserving algorithms in PRAMOD. We demonstrate this approach later with the privacy-preserving join algorithm.

4. IMPLEMENTATION

In this section, we describe implementations of four algorithms, namely sort, compaction, group aggregation and join. The first two algorithms achieve security directly using the scrambler, the last two derive security from that of their substeps. For each algorithm, we first explain how it is implemented in the baseline system, then contrast it with the privacy preserving implementation. Finally, we compare the performance of different implementations, the results of which are shown in Table 2

4.1 Sort

The algorithm rearranges the input according to a certain order of the record keys.

Baseline solution. We implement the well known external merge sort [31]. First, the input is divided into $s = n/m$ blocks (for simplicity, suppose $s < m$). Each block is sorted entirely inside the trusted unit. Next, all s sorted blocks are combined in 1 merge step using s-way merge. Specifically, the trusted memory is divided into $s + 1$ parts, s of which serve as input buffers, one for each sorted block. The last part is used as the output buffer. *s-way* merge results in optimal I/O performance as the input is read only once during merging. This solution, however, leaks information about the input order, as illustrated in the example (Section 2).

Algorithm 1 Privacy-Preserving Sort

```
1: procedure Sort($X$)
2:     $X' \leftarrow$ MakeKeyDistinct($X$);
3:     $\widetilde{X} \leftarrow$ Scramble ($X'$);
4:     $Y' \leftarrow$ExternalMergeSort($\widetilde{X}$);
5:     $Y \leftarrow$ RevertKey($Y'$);
6:     return $Y$;
7: end procedure
```

Privacy-Preserving Solution . Algorithm 1 shows the privacy-preserving sort algorithm — pSort — consisting of four steps. (1) The input is transformed to X' whose keys are distinct (pre-processing step). This can be done by appending the address of the record to its key, i.e. $key(x'_i) = key(x_i)||i$. (2) X' is securely permuted by the scrambler, the result is $\widetilde{X}$. (3) Merge sort is performed on $\widetilde{X}$, in which the comparison function uses the address to break ties. (4) Output of step 3 is scanned through to remove the address information (post-processing).

We stress that even though merge sort is used in step 3, this solution can be applied to any other algorithms. This generality is advantageous to our system – it can adopt the algorithm most appropriate and efficient for the targeted applications.

Performance Analysis. The scrambler and merge sort run in $O(n)$ and $O(n \log n)$ respectively, hence pSort runs in $O(n \log n)$. To the best of our knowledge, the most efficient external, oblivious sort algorithms are from Goodrich *et al.* [25, 22] which run in $O(n \log^2 n)$ for the deterministic version, and $O(n \log n$ for the randomized version. It can be seen that pSort has asymptotically optimal performance, while being simpler than existing oblivious algorithms.

4.2 Compaction

The algorithm removes *marked* records from the input. The output is a compact dataset of size $n' \leq n$ which is order-preserving: if x_i and x_j are to be retained and $i < j$, then in the output x_i appears before x_j. We assume that a record is marked with 1 if it is to be retained, and with 0 if it is to be dropped. Note that that the output size n' is not a secret, i.e. our security definition allows this to be learned by the adversary. This is reasonable, because the purpose of the algorithm is to reduce the number of records stored on the storage which, in turn, resides in the untrusted environment and is accessible to the adversary. Keeping n' secret would incur unnecessary complexity and defeat the purpose of compaction.

Baseline solution. We adopt a simple approach which sequentially pulls data records into the trusted unit, decrypts them inside the secure environment, re-encrypts and writes back to the storage only those whose labels are 1. Those with labels 0 are discarded. This approach is efficient, but it reveals positions of the discarded records.

Algorithm 2 Privacy-Preserving Compaction

```
1: procedure Compact($X$)
2:     $X' \leftarrow$ Mark($X$)
3:     $\widetilde{X} \leftarrow$ Scramble ($X'$)
4:     $\widetilde{Y} \leftarrow$ Filter ($\widetilde{X}$)
5:     $Y \leftarrow$ Arrange ($\widetilde{Y}$)
                                  ▷ Arrange() is offloaded to the worker
6:     return $Y$
7: end procedure
```

Privacy Preserving Solution . Algorithm 2 shows the privacy preserving compaction algorithm — pCompact — consisting of four steps. (1) In the pre-processing step, the trusted unit initiates two counters, $C_1 = 0$ and $C_2 = n$. Scanning through X, if a record is to be retained, it marks the record with C_1 and increments C_1 by 1; if a record is to be removed, it marks the record with C_2 and decrements C_2 by 1. (2) The scrambler randomly permutes the marked dataset to $\widetilde{X}$. (3) The baseline algorithm is applied on $\widetilde{X}$ to get a compact dataset $\widetilde{Y}$. Note that $\widetilde{Y}$ is not order-preserving. (4) The mark information of the retaining records is revealed to the worker which arranges records to their desired positions. The worker can finish this step by one linear scan through $\widetilde{Y}$ as opposed to sorting $\widetilde{Y}$ with respect to the mark information. We remark that revealing the marking of the records does not leak any sensitive information on X, thus it does not compromise pCompact's privacy.

Performance Analysis. Every step of pCompact runs in linear time, thus the algorithm runs in $O(n)$. The data-oblivious algorithm [22] first performs oblivious stable sort on the labels (which are either 0 or 1), then discards the last $n-n'$ records. This approach runs in $O(n \log n)$ with a small constant factor. It can be seen that, pCompact is asymptotically better than the state-of-the-art oblivious compaction algorithm.

4.3 Grouping and Aggregation

This algorithm groups input records based on their keys and then performs an aggregation, such as summing or averaging, over the group members. Specifically, let $K = \langle k_1, k_2, \ldots, k_{n'} \rangle$ be the set of distinct keys, the output of this algorithm is the sequence $Y = \langle y_1, y_2, \ldots, y_{n'} \rangle$ in which $key(y_i) = k_i$ and $value(y_i) = \text{Agg}(value(x) : x \in X; key(x) = k_i)$.

Baseline solution. The algorithm starts by sorting the records on the keys. Next, the sorted records are scanned, the aggregate values accumulated and written out immediately after encountering the last keys of the groups. Even if privacy-preserving sort is used, because of this last step, the overall execution reveals the size of each group.

X	V_2	V_1	W_1	W_2	X_{1exp}	X_{2exp}	Y
$\langle a, fde\rangle_{X_1}$	$\langle a, fde\rangle_{X_1}(1)$	$\langle d, kbs\rangle_{X_2}(1)$	$\langle b, xdj\rangle(1)$	$\langle a, maj\rangle(2)$	$\langle b, xdj\rangle$	$\langle b, med\rangle$	$\langle b, xdjmed\rangle$
$\langle a, tol\rangle_{X_1}$	$\langle a, tol\rangle_{X_1}(2)$	$\langle c, tfn\rangle_{X_2}(1)$					
$\langle a, maj\rangle_{X_2}$	$\langle a, maj\rangle_{X_2}(2)$	$\langle b, med\rangle_{X_2}(1)$	$\langle b, lxv\rangle(1)$	$\langle b, med\rangle(2)$	$\langle b, lxv\rangle$	$\langle b, med\rangle$	$\langle b, lxvmed\rangle$
$\langle b, lxv\rangle_{X_1}$	$\langle b, lxv\rangle_{X_1}(1)$	$\langle b, xdj\rangle_{X_1}(1)$					
$\langle b, xdj\rangle_{X_1}$	$\langle b, xdj\rangle_{X_1}(2)$	$\langle b, lxv\rangle_{X_1}(1)$	$\langle a, tol\rangle(1)$	$\langle c, tfn\rangle(0)$	$\langle a, tol\rangle$	$\langle a, maj\rangle$	$\langle a, tolmaj\rangle$
$\langle b, med\rangle_{X_2}$	$\langle b, med\rangle_{X_2}(2)$	$\langle a, maj\rangle_{X_2}(1)$					
$\langle c, tfn\rangle_{X_2}$	$\langle c, tfn\rangle_{X_2}(0)$	$\langle a, tol\rangle_{X_1}(1)$	$\langle a, fde\rangle(1)$	$\langle d, kbs\rangle(0)$	$\langle a, fde\rangle$	$\langle a, maj\rangle$	$\langle a, fdemaj\rangle$
$\langle d, kbs\rangle_{X_2}$	$\langle d, kbs\rangle_{X_2}(0)$	$\langle a, fde\rangle_{X_1}(1)$					

Table 3: Example of join for input $X_1 = \{\langle a, fde\rangle, \langle a, tol\rangle, \langle b, lxv\rangle, \langle b, xdj\rangle\}$ and $X_2 = \{\langle a, maj\rangle, \langle b, med\rangle, \langle c, tfn\rangle, \langle d, kbs\rangle\}$. Values in parentheses appeared in columns W_1 and W_2 represent records' degree in the join graph while those in columns V_1 and V_2 are running sum computed by FRSum() and RRSum().

Algorithm 3 Privacy-Preserving Aggregation

```
 1: procedure PrivateSum($X$)
 2:     $G \leftarrow$ pSort($X$)
 3:     $k = k_1$
        ▷ $k_1$ is first element in the the set of distinct keys $K$.
 4:     $v = 0$
 5:     for each g in G do
 6:         if $key(g) = k$ then
 7:             $v \leftarrow v + value(g)$
 8:             Add $\langle dummy\rangle$ to $V$ ▷ output dummy records
 9:         else
10:             Add $\langle k, v\rangle$ to $V$
11:             $k \leftarrow key(g)$
12:             $v \leftarrow value(g)$
13:         end if
14:     end for
15:     $Y \leftarrow$ pCompact($V$) ▷ Remove all $\langle dummy\rangle$ from $V$
16:     return $Y$
17: end procedure
```

Algorithm 4 Privacy-Preserving Join

```
 1: procedure Join($X_1, X_2$)
 2:     $X \leftarrow X_1||X_2$
 3:     $S \leftarrow$ pSort($X$)
            ▷ tie is broken such that $X_1$ records always come
    before $X_2$ records
 4:     $V_2 \leftarrow$ FRSum($S$)
 5:     $V_1 \leftarrow$ RRSum($S$)
 6:     $W_1 \leftarrow$ pCompact($V_1$)
 7:     $W_2 \leftarrow$ pCompact($V_2$)
 8:     $X_{1exp} \leftarrow$ OExpand ($W_1$)
 9:     $X_{2exp} \leftarrow$ OExpand ($W_2$)
10:     $Y \leftarrow X_{1exp} \cdot X_{2exp}$
    ▷ stitch expansion of $X_1$ and $X_2$ to get the join output
11:     return $Y$
12: end procedure
```

Privacy-Preserving Solution . Algorithm 3 shows our privacy-preserving algorithm based on sort, compaction and a scanning step. First, we sort X using pSort algorithm, obtaining G in which records of the same keys are inherently grouped together. Next, the trusted unit scans through G, processes each record in G and computes an intermediate result V. To prevent the worker from inferring the sizes of each group, the trusted unit not only outputs a valid aggregation value, but also dummy records. Finally, pCompact is used to removes dummy records in V. Since these 3 steps are privacy-preserving, it follows from Theorem 2 that pAgregation is also privacy-preserving.

Performance Analysis. pAgregation is constructed from pSort and pCompact, thus it runs $O(n \log n)$ time. The data oblivious algorithm [8] adopts the same workflow, and its time complexity is also $O(n \log n)$.

4.4 Join

The algorithm takes as input two datasets X_1, X_2 of size n_1, n_2 and outputs $Y = X_1 \bowtie X_2$. For simplicity, we use the following conventions. A record $x_i \in X_1$ matches with another record $x_j \in X_2$ if and only if $key(x_i) = key(x_j)$. Denote $y_{ij} = x_i \cdot x_j$ as the join output of x_i and x_j, it follows that $key(y_{ij}) = key(x_i) = key(x_j)$ and $value(y_{ij}) = value(x_i)||value(x_j)$. Unlike previous algorithms, the output size of pJoin can be larger than its input sizes.

Baseline solution. We consider the sort-merge join algorithm. Specifically, X_1 and X_2 are first sorted, then interleaved linear scans are performed to find matching records. However, combined leakage from the sorting step and the matching step may reveal the entire join graph.

Privacy-Preserving Solution. Algorithm 4 shows our privacy-preserving join algorithm — pJoin — which is based on the data oblivious algorithm proposed by Arasu *el al.* [9]. It consists of two stages. The first stage computes the degree of each record in the join graph, followed by the second stage that duplicates each record a number of times indicated by its degree. The output is generated by "stitching" corresponding (duplicated) records with each other. pJoin basically replaces the oblivious algorithm in [9] for computing record degrees with pSort and pCompact (line 2-7). However, it use the oblivious expansion algorithm without change (line 8-9). It follows from Theorem 2 that pJoin is privacy preserving.

In the first stage, pJoin first combines X_1 and X_2 into one big dataset X, then privately sorts X, ensuring that for those records that have the same keys, tie is broken by placing X_1's records before X_2's. Since X is sorted, records having the same key are naturally grouped together. The trusted unit then scans the entire X in two passes. The first pass, FRSum(), assumes that each X_1 record has *weight* of 1 while X_2 record has weight of 0. It scans X from left to right and computes the sum of weights in each group. At the end of this pass, each record in X_2 is associated with a weight representing its degree in the join graph. The second pass, RRSum(), similarly scans from right to left, at the end of which each record in X_1 is associated with its degree in the join graph. After the two passes, pCompact is invoked twice to remove X_2 and X_1 records from V_1 and V_2, respectively, giving two weight sequences W_1 and W_2.

In the second stage, each record in X_1 and X_2 is duplicated a number of times indicated by its associated weight. We use the oblivious expansion algorithm presented in [9] directly to implement this step. The results are stitched together via a linear scan to generate the final join output Y. Table 3 gives an example for pJoin.

Performance Analysis.. The time complexity of the first stage is $O(n \log n)$, of the second stage is $O(n_1 \log n_1 + n_2 \log n_2 + l \log l)$ where $l = |X_1 \bowtie X_2|$. Overall, pJoin runs in $O(n \log n + l \log l)$, which is the same complexity as that of the oblivious algorithm [9]. However, we show later in Section 5, pJoin has lower running time because of the efficient pSort and pCompact algorithm.

5. PERFORMANCE EVALUATION

In this section, we evaluate the performances of our privacy-preserving algorithms pSort, pCompact, pAgregation and pJoin. Data records are generated following Yahoo! TeraSort benchmark [37], in which each record comprises of a 10 bytes key and 90 bytes value. All the records are encrypted with AES-GCM utilizing 256-bit keys, generating 132 bytes ciphertexts. All experiments are run on a DELL system equipped with Intel i5-4570 Processor running at 3.2GHz frequency and a 500GB SATA hard drive. The trusted unit assumes a 20MB secure memory. Input datasets are assumed to be uniformly distributed. Their sizes are varied from 8 to 64 GB. Our implementation utilizes *Crypto++* library [1] for implementing cryptographic operations. We repeat each experiment 10 times and report average results.

5.1 Cost of Security

Operations	Baseline	PRAMOD	Oblivious Algorithms
Sorting	3782.86	9195.78 (2.43×)	37641.81 (9.95×)
Compaction	1553.88	7364.8 (4.74×)	24636.32 (15.85×)
Group-Aggregation	5336.74	18144.46 (3.39×)	63831.98 (11.96×)
Join	8444.53	42221.43 (4.99×)	105210.44 (12.46×)

Table 4: Overall running time (in seconds) of our PRAMOD's algorithms in comparison with: (1) baseline system protecting data confidentiality at rest and (2) corresponding data-oblivious algorithms.

We first benchmark our algorithms against baseline system that uses conventional external-memory algorithms to process the encrypted data without our proposed privacy-preserving technique. The baseline system implements the external-memory algorithms in a sense that the trusted unit serves a role of main memory while the storage is considered as the secondary storage. This baseline system only protects data confidentiality at rest, while reveals sensitive information on the input via access-pattern as discussed in section 4. Further, we also compare PRAMOD's algorithms with a class of corresponding data-oblivious algorithms for sorting [25], compaction [22], group-aggregation [8] and join [9]. For conciseness, hereafter we shall refer to these oblivious algorithms as oblSort, oblCompact, oblAggregate and oblJoin, respectively.

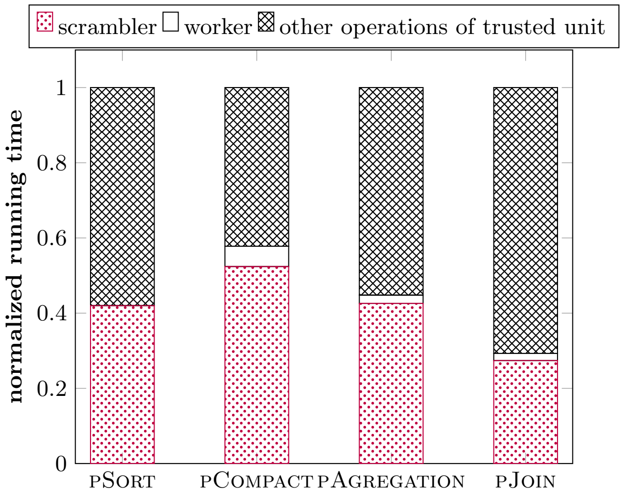


Figure 4: Normalized break-down running time of PRAMOD's algorithms. The running time consists of the time taken by the scrambler, plus the time required by the worker (if any). The rest is incurred by the adopted non-privacy-preserving algorithm together with the pre-processing and/or post-processing steps (if any).

Overhead. Table 4 describes overheads in processing an input database X of size 32GB containing $n = 2^{28}$ records[2] by a trusted unit whose memory is of size 20MB (i.e. it is able to process $m = 2^{17}$ records at once). To guard against the adversary aiming to traverse the execution to learn sensitive information on input, PRAMOD witnesses overheads of between 2.43× to 4.99× over the baseline system. Although this cost of security is admittedly large, PRAMOD is more practical than the relevant data-oblivious solutions, which offers a similar level of privacy-protection, yet incurs 9.95× to 15.85× overheads.

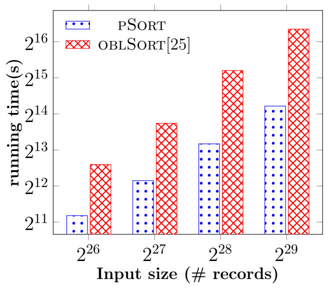
Cost breakdown. To provide an insight into the cost factors that contribute to the overheads, we report time taken by the scrambler, the worker (if any) and other operations performed by the trusted unit, such as pre-processing and/or post-processing steps along with the running time of the adopted non-privacy-preserving algorithms.

Figure 4 depicts this breakdown. As can be seen, the cost of the scrambler is significant. In particular, this primitive contributes 42%, 52.3%, 42.6% and 27.4% the costs of pSort, pCompact, pAgregation and pJoin, respectively. For those algorithms that involve the untrusted worker in processing the intermediate values, its processing time only

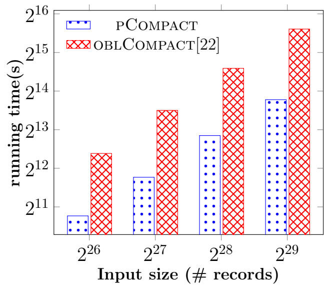
[2]For join algorithms which take as input two dataset X_1 and X_2, we consider the input size to be the total size of X_1 and X_2 (i.e. $n = |X_1| + |X_2|$)

Algorithm	# Re-Encryption	I/O Complexity
pSort	$(p_1 + p_2 + 5) \cdot n$	$O(n)$
oblSort[25]	$(\sum_{i=1}^{\log s} i + \log s + 1) \cdot n$	$O(n \log^2 n)$
pCompact	$(p_1 + p_2 + 2) \cdot n$	$O(n)$
oblCompact[22]	$(1 + \log n) \cdot n$	$O(n \log n)$
pAgregation	$(2p_1 + 2p_2 + 8) \cdot n$	$O(n)$
oblAggregate[8]	$(\sum_{i=1}^{\log s} i + \log s + \log n + 3) \cdot n$	$O(n \log^2 n)$
pJoin	$(3p_1 + 3p_2 + 9 + d) \cdot n$	$O(dn)$
oblJoin[9]	$(\sum_{i=1}^{\log s} i + \log s + 2 \log n + 5 + d) \cdot n$	$O((dn) \log(dn) + n \log^2 n)$

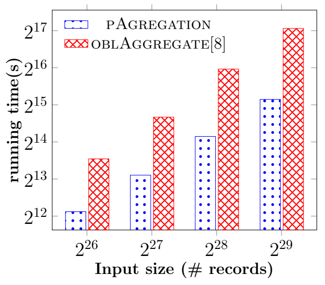
Table 5: Number of re-encryptions and read/write complexity required by PRAMOD's algorithm and relevant oblivious algorithms in processing input of size n. p_1 and p_2 are constant parameters in the scrambler's configuration (see Section 3.1). In our experiments, $p_1 = 2$ and $p_2 = 3$. Let m be the capacity of the secure memory, then $s = n/m$. d is the average degree of records in the join graph. In our experiments, $d = 3$.



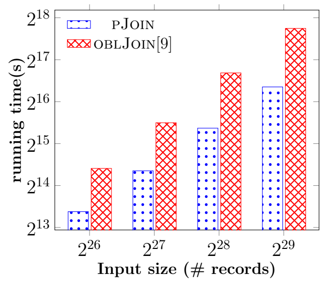
(a) Sorting



(b) Compaction



(c) Group-Aggregation



(d) Join

Figure 5: Performance comparison between our algorithms and the related data-oblivious algorithms. Running time is reported in log-scale (y-axis) for different input sizes (x-axis).

constitutes a small proportion of the the total running time, ranging from 1.8% (for pJoin) to 5.4% (for pCompact). This is because the worker operates directly on the cloud storage in the untrusted environments, thus incurring no I/O request. Moreover, its operations do not involve re-encryptions, which are computationally expensive.

Re-Encryptions and I/O costs. Since cryptographic operation is CPU-intensive and I/O communication between the trusted unit and the storage is slow (in comparison with in-memory computation), these two operations significantly affect the overall running time of the algorithms. Indeed, Goodrich *et al.* measure the complexity of their external-memory algorithms in term of I/O requests [25]. Table 5 details the number of re-encryptions required and I/O complexity of each algorithm.

We observe that on input of size n, PRAMOD's algorithms need $O(n)$ I/Os, while the relevant oblivious algorithms, except for oblCompact, require $O(n \log^2 n)$ I/Os. We note that for join algorithms, the I/O complexity also depends on d, the average degree of each record in the join graph. For uniformly distributed datasets, d can be considered as a constant. In our experiments, we assume $d = 3$ and exclude the situations in which d is proportional n.

Recall that a record is re-encrypted every time it is pulled to the trusted unit and then sent back to the storage, the number of re-encryptions necessitated in each execution is proportional to its I/O complexity. More specifically, PRAMOD's algorithms requires $O(n)$ re-encryptions, while corresponding oblivious algorithms needs $\Theta(n \log n)$ re-encryptions in processing input of size n. In Table 5, we give the actual number of re-encryptions needed in each algorithm. These numbers depend on the configuration and the implementation of the algorithm. More specifically, for PRAMOD's algorithms, the scrambler needs $(p_1+p_2+1) \cdot n$ re-encryptions in scrambling n records. In our experiments, we find that with respect to the datasets under consideration, the scrambler achieves fast running time and zero possibility of failure when $p_1 = 2$ and $p_2 = 3$. On the other hand, the numbers of re-encryptions performed by oblSort, oblCompact, oblJoin are affected by the capacity of the secure memory. With respect to our assumption on the size of the secure memory ($m = O(\sqrt{n})$), given the same input, the oblivious algorithms executes 2 to 3 times more re-encryptions than PRAMOD's algorithms.

5.2 Efficiency and Scalability

We further investigate the efficiency of our algorithms by varying the input sizes between 8 and 64 GB. Figure 5 reports performance of our algorithms (in log-scale) in comparison with the oblivious alternatives. pSort outperforms oblSort [25] by 2.6× to 4.4× (Figure 5a). Similarly, pCompact is 3× to 3.5× faster than oblCompact [22] (Figure 5b), pAgregation outruns oblAggregate [8] by 2.7× to 3.8× (Figure 5c), and pJoin is 2× to 2.6× more efficient than oblJoin [9] (Figure 5d).

It is also worth noting that the speed-up that PRAMOD's algorithms gains over the oblivious algorithms becomes more significant for large input, from 2× to 3× for 8GB dataset to 2.6× to 4.4× for 64GB dataset. The speeding-up is mainly because our algorithms requires less re-encryptions and I/O operations than the oblivious algorithms. With this, we conclude that our privacy-preserving algorithms are efficient and scalable.

Discussion. We remark that it is possible to port PRAMOD to distributed environments. Our algorithms mainly comprise of two components, the scrambler and the underlying non-privacy-preserving external-memory algorithms. The scrambler processes data in blocks independent of each other, thus it is trivial to adapt this primitive to the distributed settings, enabling optimal speed-up for the scrambler with parallelization (i.e. doubling the number of trusted units should halve the scrambler's runtime). Indeed, our implementation multi-threads this primitive and witnesses a speed-up of 1.8×. Besides, most external-memory algorithms are often designed to seamlessly support parallelism. Although we have not yet ported it to distributed computing settings, we believe that parallelization can significantly speed up the execution of PRAMOD. On the other hand, we believe that it is difficult to parallelise the oblivious algorithms oblSort [25], oblCompact [22], oblAggregate [8] and oblJoin [9].

6. RELATED WORK

Secure Data Management using Trusted Hardware. A large body of systems have utilized trusted hardware such as IBM 4764 PCI-X [2] or Intel SGX [3] to enable secure data management. TrustedDB [10] presents a secure outsourced database prototype which leverages on IBM 4764 secure CPU (SCPU) to enable privacy-preserving SQL queries on untrusted servers. Cipherbase [8] follows TrustedDB's idea on combining SCPU with commodity servers to offer a full-fledged SQL database system that assures high data confidentiality. Besides, SCPUs are also employed in secure multi-party computation settings [4]. VC^3 uses another trusted computing primitive, namely Intel SGX processors, as a building blocks in their general-purposed data analytics system. It enables MapReduce computations, and offers privacy guarantee not only for users' data, but also their code. However, these techniques either only protect data confidentiality at rest or incurs large overhead in preserving overall privacy and security.

Secure Computation by Data Oblivious Technique. Oblivious-RAM [21] enables secure and oblivious computation by hiding data read/write patterns during the program execution. ORAM setting places trust on a CPU with limited internal memory, while the users' program and data are both stored encrypted on the untrusted server. Security is achieve by making data accesses to the untrusted server appear random and irrelevant to the true and intended access sequence. The ubiquity of cloud computing and storage services, together with a variety of privacy concerns have revived research interest in ORAM. Various works have been proposed [39, 13, 25]. A non-oblivious algorithm can be made oblivious by utilizing ORAM technique. However, this comes at an expense of performance degradation of at least a $O(\log n)$ multiplicative factor, where n is the data size. PRAMODoffers a similar level of security, with only $O(n)$ additive overhead.

Another line of works advocate for designing data-oblivious algorithms. Goodrich *et al.* present several oblivious algorithms for sorting [25, 23, 24], compaction and selection [22]. The authors also propose approaches to simulate ORAM environment using data-oblivious algorithms [25]. Other interesting data-oblivious algorithms have also been proposed for graph drawing [26], graph-related computations such as maximum flow, minimum spanning tree, single-source single-destination (SSSD) shortest path, and breath-first search [12]. Unfortunately, these algorithms are on the one hand operation-specific, while less efficient than PRAMOD's algorithms on the other hand.

7. CONCLUSION

In this paper, we have introduced a simple yet efficient privacy-preserving approach for enabling computation on large dataset using a trusted unit with limited secure memory. We mainly focus on data management operations whose purpose is to rearrange data items. We show that for various applications, appending a non-privacy-preserving algorithm with a scrambling step gives an efficient privacy-preserving algorithm. To better illustrate our proposed approach, we described four privacy-preserving algorithms for sorting, compaction, aggregation and join. Experiments have shown that our privacy-preserving algorithms are efficient and scalable, outperforming data-oblivious algorithms for the set of operations under consideration upto 3×, while offering the similar level of privacy protection.

8. REFERENCES

[1] Crypto++ library. `www.cryptopp.com/`.

[2] Ibm 4764 pci-x cryptographic coprocessor. `http://www-03.ibm.com/security/cryptocards/pcixcc/overview.shtml`.

[3] Software guard extensions programming reference. `https://software.intel.com/sites/default/files/managed/48/88/329298-002.pdf`.

[4] R. Agrawal, D. Asonov, M. Kantarcioglu, and Y. Li. Sovereign joins. In *ICDE 2006*.

[5] R. Agrawal, J. Kiernan, R. Srikant, and Y. Xu. Order preserving encryption for numeric data. In *ACM SIGMOD*, 2004.

[6] M. Y. Ahmad and B. Kemme. Compaction management in distributed key-value datastores. In *PVLDB*, 2015.

[7] A. Aiyer, M. Bautin, G. J. Chen, P. Damania, P. Khemani, K. Muthukkaruppan, K. Ranganathan, N. Spiegelberg, L. Tang, and M. Vaidya. Storage infrastructure behind facebook messages using hbase at scale. *Data Engineering Bulletin*, 2012.

[8] A. Arasu, S. Blanas, K. Eguro, R. Kaushik, D. Kossmann, R. Ramamurthy, and R. Venkatesan. Orthogonal security with cipherbase. CIDR'13.

[9] A. Arasu and R. Kaushik. Oblivious query processing. *arXiv preprint arXiv:1312.4012*, 2013.

[10] S. Bajaj and R. Sion. Trusteddb: A trusted hardware-based database with privacy and data confidentiality. *TKDE*, 2014.

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

[12] M. Blanton, A. Steele, and M. Alisagari. Data-oblivious graph algorithms for secure computation and outsourcing. ASIACCS '13.

[13] D. Boneh, D. Mazieres, and R. A. Popa. Remote oblivious storage: Making oblivious ram practical. 2011.

[14] Z. Brakerski and Z. Brakerski. Efficient fully homomorphic encryption from (standard) lwe. In *FOCS*, 2011.

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

[16] S. Chen, R. Wang, X. Wang, and K. Zhang. Side-channel leaks in web applications: A reality today, a challenge tomorrow. In *IEEE Security and Privacy, 2010.*

[17] Y. Chen and R. Sion. On securing untrusted clouds with cryptography. *Data Engineering Bulletin*, 36, 2012.

[18] A. Dinh, P. Saxena, E.-C. Chang, B. C. Ooi, and C. Zhang. M2r: Enabling stronger privacy in mapreduce computation. In *USENIX Security,15.*

[19] T. ElGamal. A public key cryptosystem and a signature scheme based on discrete logarithms. In *Advances in Cryptology*, 1985.

[20] C. Gentry et al. Fully homomorphic encryption using ideal lattices. In *STOC*, 2009.

[21] O. Goldreich and R. Ostrovsky. Software protection and simulation on oblivious rams. *J. ACM.*

[22] M. T. Goodrich. Data-oblivious external-memory algorithms for the compaction, selection, and sorting of outsourced data. SPAA '11.

[23] M. T. Goodrich. Randomized shellsort: A simple data-oblivious sorting algorithm. *J. ACM*, 2011.

[24] M. T. Goodrich. Zig-zag sort: A simple deterministic data-oblivious sorting algorithm running in o(n log n) time. *CoRR*, 2014.

[25] M. T. Goodrich and M. Mitzenmacher. Privacy-preserving access of outsourced data via oblivious ram simulation. *CoRR*, abs/1007.1259, 2010.

[26] M. T. Goodrich, O. Ohrimenko, and R. Tamassia. Data-oblivious graph drawing model and algorithms. *arXiv preprint arXiv:1209.0756*, 2012.

[27] A. Halevy, A. Rajaraman, and J. Ordille. Data integration: The teenage years. In *VLDB*, 2006.

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

[29] A. Khoshgozaran, H. Shirani-Mehr, and C. Shahabi. Spiral: A scalable private information retrieval approach to location privacy. In *MDMW 2008.*

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

[31] D. Knuth. The art of computer programming 1: Fundamental algorithms 2: Seminumerical algorithms 3: Sorting and searching, 1968.

[32] A. Lakshman and P. Malik. Cassandra: a decentralized structured storage system. *Operating Systems Review*, 44, 2010.

[33] F. Li, M. Hadjieleftheriou, G. Kollios, and L. Reyzin. Dynamic authenticated index structure for outsourced databases. In *SIGMOD*, 2006.

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

[35] 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.

[36] O. Ohrimenko, M. T. Goodrich, R. Tamassia, and E. Upfal. The melbourne shuffle: Improving oblivious storage in the cloud. In *Automata, Languages, and Programming.* 2014.

[37] O. OMalley and A. C. Murthy. Winning a 60 second dash with a yellow elephant. *Proceedings of sort benchmark*, 2009.

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

[39] B. Pinkas and T. Reinman. Oblivious ram revisited. In *CRYPTO, 2010.*

[40] R. A. Popa, J. R. Lorch, D. Molnar, H. J. Wang, and L. Zhuang. Enabling security in cloud storage slas with cloudproof. In *USENIX Security*, 2011.

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

[42] F. Schuster, M. Costa, C. Fournet, C. Gkantsidis, M. Peinado, G. Mainar-Ruiz, and M. Russinovich. Vc^3: Trustworthy data analytics in the cloud. In *IEEE Security and Privacy, 2014.*

[43] E. Stefanov, M. Van Dijk, E. Shi, C. Fletcher, L. Ren, X. Yu, and S. Devadas. Path oram: An extremely simple oblivious ram protocol. CCS '13.

[44] S. Subashini and V. Kavitha. A survey on security issues in service delivery models of cloud computing. *Journal of network and computer applications*, 2011.

[45] H. Takabi, J. Joshi, and G.-J. Ahn. Security and privacy challenges in cloud computing environments.

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

[47] S. Wang, X. Ding, R. H. Deng, and F. Bao. Private information retrieval using trusted hardware. In *ESORICS 2006.*

A. THE MELBOURN SHUFFLE

The shuffle algorithm shares our assumptions on encryption of data records. Particularly, all records are encrypted using a semantic secure encryption scheme. They are only decrypted inside the trusted unit and re-encrypted before being written back to the storage.

The shuffle takes as input a randomly chosen permutation π and a data set X of n items. The permutation π can be chosen by employing a pseudo random permutation [28], and represented using a short secret seed. It obliviously arranges n items to their final position in $\widetilde{X}$ with respect to π. The shuffling requires two intermediate arrays T_1 and T_2 which are of size $p_1 n$ and $p_2 n$ where p_1 and p_2 are constants and $p_2 \geq p_1$. First, X, T_1, T_2 and $\widetilde{X}$ are divided into $\sqrt{n}$ *buckets*, each contains $O(\sqrt{n})$ records. Every $\sqrt[4]{n}$ buckets constitute a *chunk* and there are $\sqrt[4]{n}$ chunks in total. Each bucket of T_1 holds $p_1\sqrt{n}$ records while each bucket in T_2 stores $p_2\sqrt{n}$.

The algorithm proceeds in two phases: *distribution* and *clean-up*. The first phase comprises of two rounds. Records are moved from X to T_1 in the first round, such that records belonging to the i^{th} chunk of $\widetilde{X}$ will be put in the i^{th} chunk of T_1. In the second round, records in T_1 are distributed among buckets of T_2 such that at the end of this distribution, records are located in their correct buckets. To ensure the obliviousness, data written to T_1 and T_2 are padded to equal size. This implies adding dummy records. There are $(p_1 -$

1)n dummy records in T_1 and similarly $(p_2 - 1)n$ are written to T_2. The second phase, clean-up, removes dummy records and arranges real records to correct positions within their own bucket.

In each round, the trusted unit sequentially process each of $\sqrt{(n)}$ buckets. Recall that each bucket contains $O(\sqrt{n})$ records, the entire bucket can fit in the secure memory of the trusted unit. Records within the bucket, after being read to the secure memory, are divided into $\sqrt[4]{n}$ segments according to their final positions. In distributing records from X to T_1, each segment has at most $p_1 \sqrt[4]{n}$ records and they are written to corresponding chunks in T_1. Similarly, in the second distribution, each segment hold upto $p_2 \sqrt[4]{n}$ records, which are then placed to their corresponding buckets. If a segment contains less records than its capacity, dummy records are added to ensure data-obliviousness. However, if so many records are located to one segment that it becomes overflowed, the algorithm fails. Nevertheless, it is proven that with appropriate value of p_1 and p_2, the probability of failure is negligible [36].