PRAMOD: A Privacy-Preserving Framework for Supporting Efficient and Secure Database-as-a-Service

Hung Dang, Tien Tuan 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. However, security in terms of data privacy remains an obstacle because data storage and processing are performed on an untrusted cloud. Strong security under additional constraints of functionality and performance is even more challenging to achieve, for which advanced encryption and recent trusted computing primitives alone prove insufficient.

In this paper, we propose PRAMOD – a novel framework for enabling efficient and secure database-as-a-service. We consider a setting in which data is stored encrypted on the untrusted cloud and data-dependent computations are performed inside a trusted environment. The proposed framework protects against leakage caused by observable data movement between different components (due to limited private memory) by using a special component called *scrambler* running in $O(n)$ time. It supports popular algorithms underlying many data management applications, including sort, compaction, join and group aggregation. The algorithms implemented in PRAMOD are not only privacy-preserving but also asymptotically optimal. They can be used as building blocks to construct efficient and secure query processing algorithms. The experimental study shows reasonable overheads over a baseline system offering a weaker level of security. More remarkably, PRAMOD shows superior performance in comparison with state-of-the-art solutions with similar privacy protection: up to 4.4× speedup over the alternative data-oblivious algorithms.

1. INTRODUCTION

Big data is the driving force behind the database-as-a-service model offered by most 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, 31], data privacy in the big data era faces new challenges. First, the cloud providers have more incentives in extracting content of the outsourced data for its commercial values [45, **?**]. Second, even when the providers are trusted, multi-tenancy, complexity of software stacks, and distributed computing models continue to enlarge the 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, 9].

The first step towards securing the data is to encrypt it before outsourcing to the cloud. Unfortunately, this only protects data at rest [40, 31]. Fully homomorphic encryption allows computation over encrypted data, but it suffers from prohibitive performance [20, 14]. Partially homomorphic encryption schemes [38, 19] are more practical, but they are limited in the range of supported operations [41, 46], and have been shown to be vulnerable to attacks [34]. Consequently, some recent works have advocated an approach combining encryption with trusted computing primitives [11, 42, 7, 9], in which confidentiality and integrity protected execution environments are provisioned by hardware (e.g. Intel SGX [3]) or by hardware-software combination [32, 33]. In such a secure environment, computations are performed on decrypted data, and the results are encrypted before being returned. However, there is a limit on the amount of data that the secure environment can process at any time, the upper bound being the size of physical memory allocated to a process. This results in a data communication channel between the trusted and untrusted parties, and this channel can leak information about the data [44, 16, 18]. For instance, by observing I/O access patterns 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 application-specific data-oblivious algorithms[1] [39, 25, 44]. Both approaches, however, are complex and incur high performance overheads. It is worth noting that complexity of the codebase running inside the secure environment is undesirable for security, because it raises the cost of vetting software for implicit vulnerabilities.

In this paper, we consider a setting in which user data is stored encrypted[2] on the untrusted storage, and all data-

[1] A *data-oblivious* algorithm (or *oblivious* algorithm for short) performs the same sequence of I/O accesses on all inputs of the same size.

[2] We assume data is encrypted using a semantic secure and authenticated encryption scheme.

dependent computations are performed inside trusted computation units (or trusted units). The trusted units are securely provisioned either purely by hardware or by a hardware-software combination. An untrusted worker is responsible for data movements and other house keeping tasks, during these process the worker can observe access patterns. Given this setting, we aim to enable practical, privacy-preserving data management. In particular, the data management algorithms running on the untrusted cloud must not leak any information about the inputs via access patterns, while admitting reasonable performance overheads.

We propose PRAMOD (PRivate dAta Management for Outsourced Databases) – a framework for implementing efficient and privacy-preserving data management algorithms. PRAMOD protects against potential leakage via access patterns. It ensures data privacy in the presence of honest-but-curious adversaries by using a component called *scrambler*. The scrambler randomly and securely permutes input data of size n in $O(n)$ time while requiring $O(\sqrt{n})$ secure memory. It is used to construct four data management algorithms, namely *sort, compaction, group aggregation* and *join*. The first two algorithms can immediately achieve security using the scrambler. They are then composed with other privacy-preserving steps to realize group aggregation and join. These four algorithms form the building blocks for constructing efficient and secure query processing algorithms.

The four algorithms under consideration underlie many data management applications. Sort is fundamental to any database systems. Compaction is vital in many distributed key-value stores where updates are directly appended to disk and compaction is frequently scheduled to improve query performance [6, 30, 5]. Join is arguably one of the most important operators in data management, and commonly used 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. The last two algorithms account for 80 over 99 queries in the TPC-DS benchmarks [43]

In PRAMOD, we make a key observation. In order to prevent leakage from access patterns, for a large class of algorithms including sort and compaction, it is sufficient to randomly permute (or scramble) the input before feeding it to the actual algorithm. Let us consider merge-sort algorithm in which the original input is randomly permuted. During the execution, an adversary observing access patterns will, at best, be able to infer only sensitive information on the scrambled input, which cannot be linked back to that of 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 its overhead factor is additive rather than multiplicative. Second, it generalizes to all algorithms implementing the same application, i.e. we can take advantage of state-of-the-art algorithms to achieve simpler yet more efficient solutions than existing data-oblivious algorithms. For instance, scrambling followed by an optimized merge sort (or any other popular sort algorithms) is simpler than data-oblivious external sort algorithms [25], and it is shown later to have better performance. It is worth noting that the simplicity of this approach implies smaller trusted computing base (TCB) which translates to better security. Also, for a complex algorithm made up of a sequence of sub-steps, there will be no leakage via access patterns when the sub-steps themselves do not leak information. This allows PRAMOD to achieve security for group aggregation and join algorithms by implementing them based on sort and compaction.

We implement the four algorithms in our framework, evaluate their performances and study the costs of security. Compared with a baseline system, PRAMOD offers a stronger privacy protection at a cost of 3.85× overhead on average. Compared with state-of-the-art data-oblivious alternatives [25, 22, 7, 8] which offer similar level of security, PRAMOD demonstrates speedup as high as 4.4×. 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 framework — PRAMOD — for implementing privacy-preserving algorithms. Certain classes of algorithms including sort and compaction immediately achieve security with a prepended scrambler, while other more complex algorithms such as group aggregation and join derive security from their substeps so long as each of which is privacy-preserving.

3. We implement the basic algorithms, namely pSort, pCompact, pAggr and pJoin, and analyze their complexity. These algorithms attain optimal complexity. pSort, pAggr and pJoin run in $O(n \log n)$ time, and pCompact runs in $O(n)$ time.

4. We conduct extensive experiments to benchmark the implementations against a baseline and state-of-the-art data-oblivious alternatives. The results demonstrate reasonable overheads over the less secure (baseline) implementations, and running time speedup of 4.4×, 3.5×, 3.8× and 2.6× over data-oblivious alternatives with similar level of security.

Next section explains the framework, security model and the problem being addressed. Section 3 describes the design of PRAMOD. Section 4 lays out detailed implementation of the four algorithms. Section 5 reports our experimental evaluation of PRAMOD. Section 6 discusses related work before Section 7 concludes.

2. PROBLEM DEFINITION

In this section, we define the problem of privacy-preserving data management of outsourced data. We state the nececssary conditions for data management algorithms to be privacy-preserving using trusted computing primitives. We provide a running example to illustrate our ideas.

Running Example. We consider a user wanting to store her data consisting of integer-value records on the cloud. The user then wishes to sort the data, as a pre-processing step for other tasks such as database loading, ranking, deduplication, etc. To this end, the user encrypts the data so that no untrusted party can learn its content. Although sorting directly over encrypted data is possible, it is highly impractical [4]. To efficiently sort the data, the user relies

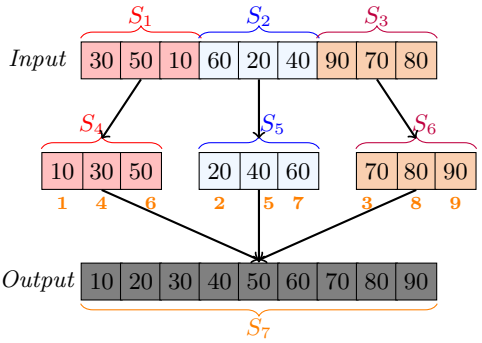


Figure 1: An example of 3-way external merge-sort. The records are encrypted (filled objects), black numbers represent integer-value of the records (invisible to untrusted parties thanks to encryption) while orange numbers denote the order in which each encrypted record is read into the trusted unit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

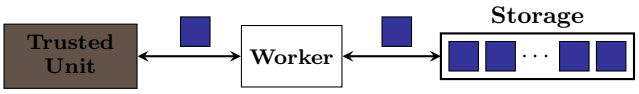


Figure 2: Computation model of a cloud server, consisting of a trusted unit capable of processing a limited of number of records at a time. Storage is untrusted, and its communication with the trusted unit is mediated by an untrusted worker (honest-but-curious). Only the trusted unit can see and compute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

on a trusted unit which decrypts the records and sort them in its secure memory before re-encrypting and sending them back to the cloud storage. Because secure memory is limited in size, the user must employ an 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.

2.1 Baseline System and Adversary Model

We now describe the baseline system, in which the user uploads data to the cloud and relies entirely on the cloud infrastructure to store and execute computations on her data. The data $X = \langle x_1, x_2, \ldots, x_n \rangle$ is a sequence of n equal-sized, key-value records, and let $key(x)$ and $value(x)$ be the key and value components of record x respectively. The data records are protected by a semantically secure encrypted scheme. Figure 2 details the cloud's comptutation model which consists of a *trusted unit*, a *worker* and a storage component. The trusted unit is trusted by the user, and it can only hold up to $m = O(\sqrt{n})$ records at a time, plus a constant number of control variables. The worker is not trusted, and it mediates access to the cloud storage which is also not trusted. The worker is also responsible for other house-keeping tasks. Both the worker and storage see only encrypted data.

Threat Model. The adversary is a curious 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. We consider honest-but-curious (or passive) attackers who try to learn information from what are observable but do 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 the main causes of security breaches (NSA and Target data breach, for example). The model with active attackers who deviate arbitrarily from the expected computation are out of the scope, and we refer readers to recent works demonstrating effective defenses against the active adversaries [18, 42].

We assume that the worker and storage component are under the attacker's control, while the trusted unit is sufficiently protected. Specifically, the trusted unit corresponds to the user's trusted computing base (TCB), and the worker and storage component correspond to the cloud software stack and storage controller. 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 (e.g. power analysis) from the trusted unit. Physical attacks, which could compromise the trusted unit's confidentiality and integrity, such as cold-boot or attacks aiming to subvert the CPU's security mechanisms, are out of scope. Finally, we assume that decryption keys have already been provisioned securely to the trusted unit, and when there are more than one trusted units, they all agree on the same decryption keys.

Leakage of the Baseline System. Let us use the running example to illustrate how the baseline system fails to ensure data privacy. As shown in Figure 1, the encrypted input is divided into three blocks, and the trusted unit execute the algorithm in two phases. In the first phase, the it independently sorts each block in-memory and returns three sorted, encrypted blocks. Next, it performs 3-way merge: 3 encrypted records are kept in memory, each of which is pulled, with help from the worker, from the sorted blocks. Observing this process, the adversary learn that the trusted unit first takes one record from each sorted block, writes one record out, then takes another record from the first block. Although it cannot learn records' content, it can still infer that the smallest record comes from S_1. Such inference in general can reveal relative order of records from different blocks. For algorithms taking data from different sources, this leakage can expose the sources' identities.

2.2 Problem Overview

This paper concerns privacy-preserving data management algorithms using trusted computing with limited private memory. This private memory is limited in a sense that it can hold (and process) only $m = O(\sqrt{n})$ records at any time. Let $\mathcal{P}$ be the algorithm executed on input X. The first goal is to restrict leakage from the execution of $\mathcal{P}$ to only the input and output sizes, i.e. $|X|$ and $|\mathcal{P}(X)|$. The baseline system fails this goal for sort as well as for other algorithms, as summarized in Table 1. One solution is to employ oblivious RAM [44] directly as the storage backend. However, this approach incurs an overhead of 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, the second goal of the paper is to attain a simple design with low performance overhead.

2.3 Security Definition

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

Algorithm	*Baseline*	*Oblivious Algorithms*	PRAMOD
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		

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 access (read/write or I/O) sequence observed by the adversary. In the baseline system, $Q_{\mathcal{P}}(X)$ represents I/O requests made by the trusted units to the worker. q_i is a 5-value tuple $\langle op,\ addr,\ val,\ time$[3]$,\ info \rangle$ where $op \in \{r, w\}$ is the type of the request ("read" or "write"), *addr* and *val* is the address and content accessed by *op* respectively, *time* is the time of request and *info* is the record's metadata ($\perp$ if undefined). The last component is useful when the trusted unit wishes to offload parts of the processing on non-sensitive data fields to the worker. For example, if an algorithm requires arranging records with respect to an order that is not secret, the trusted unit sets *info* to be the record's desired address, thus allowing the worker to complete the arranging step.

Consider the example in Figure 1, the observed read sequence, denoted as $Q_{\mathcal{P}}(X)^{read}$, is as follows (the complete sequence, including write, is similar):

$$Q_{\mathcal{P}}(X)^{read} = \begin{pmatrix} \langle r, S_1, e(S_1), t_1, \perp \rangle,\ \langle r, S_2, e(S_2), t_2, \perp \rangle, \\ \langle r, S_3, e(S_3), t_3, \perp \rangle,\ \langle r, S_1, e(S_4), t_4, \perp \rangle, \\ \cdots \\ \langle r, S_3 + 1, e(S_3 + 1), t_{10}, \perp \rangle, \\ \langle r, S_3 + 2, e(S_3 + 2), t_{11}, \perp \rangle \end{pmatrix}$$

where t_i represents the request time, and $S_i + j, e(S_i + j)$ represents the address and ciphertext of the j^{th} record in block S_i respectively.

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, using the well-accepted notion of *indistinguishability* in the literature [28], dictates that $Q_{\mathcal{P}}(X)$ reveals nothing more than the input and output size. Specifically:

Definition 1 (Privacy-Preserving Algorithm). *An algorithm $\mathcal{P}$ is privacy-preserving with advantage at most ϵ 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 $Q_{\mathcal{P}}(X_2)$ with advantage at most ϵ.*

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

Any algorithm which does not encrypt data fails to meet this definition, because $Q_{\mathcal{P}}(.)$ contains the record content. If not encrypted, the content is visible to the adversary and can be used to distinguish two different inputs. Algorithms which induce fixed access patterns over encrypted data, for instance scanning-based algorithms in the baseline system which read and then immediately write records to the same addresses, are privacy-preserving because the access sequence is the same for all inputs of the same size. On contrary, merge-sort is not privacy-preserving because there exists X_1, X_2 such that $Q_{\mathcal{P}}(X_1) \neq Q_{\mathcal{P}}(X_2)$. For example, $X_1 = \langle 10, 20, 30, 40, 50, 60, 70, 80, 90 \rangle$ and $X_2 = \langle 30, 50, 10, 60, 20, 40, 90, 70, 80 \rangle$.

[3]For simplicity, we assume there exists a global clock

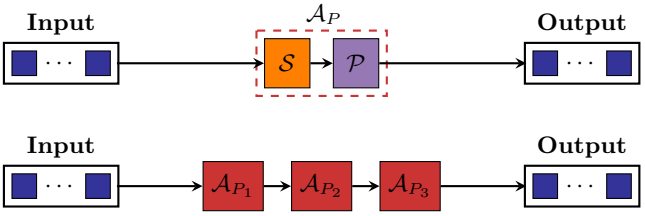


Figure 3: PRAMOD constructs privacy-preserving algorithm $\mathcal{A}_P$ from an algorithm $\mathcal{P}$ by appending it with the scrambler $\mathcal{S}$. PRAMOD can also combine simple privacy-preserving algorithms to build a more complex algorithm which is also privacy-preserving.

Discussion. A similar definition is called *data obliviousness*, which states that an algorithm $\mathcal{P}$ is data-oblivious if the observed I/O sequences are the same for X_1 and X_2 (where $|X_1| = |X_2|$), i.e. $Q_{\mathcal{P}}(X_1) = Q_{\mathcal{P}}(X_2)$. This definition implies perfect zero leakage against all adversaries, as opposed to ours which guarantees negligible leakage against computationally bounded adversaries. Data-obliviousness offers a stronger assurance, both similar to ours, it assumes that data records are always protected and the adversary cannot see their content. This assumption relies on the deployment of secure encryption. However, practical encryption schemes cannot achieve perfect secrecy. As a result, it is reasonable to expect that, in practice, both data-oblivious algorithms and privacy-preserving algorithms satisfying Definition 1 offer a similar level of privacy-protection.

3. PRAMOD

This section describes PRAMOD's design, focusing on the new component — the scrambler. We explain how to design simple privacy-preserving algorithms, namely sort and compaction, using the scrambler. We then discuss how to build more complex algorithms based on simpler, privacy-preserving steps.

PRAMOD is designed on top of the baseline system described earlier in Section 2.1. Specifically, data records are encrypted with semantically secure encryption scheme, and data-dependent computations are done inside the trusted unit with limited memory. Note that algorithms implemented in the baseline system may not be privacy-preserving. For example, merge sort in the baseline system leaks information about the input order. PRAMOD realizes

privacy-preserving algorithms in two ways. First, for algorithms which essentially re-arrange the input, it simply prepends them with a scrambler $\mathcal{S}$ which randomly permutes the input before feeding it to the algorithm (Figure 3[a]). To ensure overall security, however, the scrambler $\mathcal{S}$ must not leak information during its execution. Second, PRAMOD allows for complex, privacy-preserving algorithms to be built from a sequence of privacy-preserving substeps $\mathcal{A}_{P_1}, \mathcal{A}_{P_2}, \ldots$ (Figure 3[b]). We note that for complex algorithms, input scrambling alone is not sufficient to meet our security definition. For example, consider a group aggregation algorithm which first rearranges data into groups and then aggregates the values of each group. Even if the input is scrambled beforehand, the adversary can still learn information about group sizes, which is more than what permitted by our definition (the number of groups).

3.1 The Scrambler

The scrambler $\mathcal{S}$ is responsible for generating the scrambled data $\widetilde{X}$ from X such that their linkage is not revealed via I/O patterns of $\mathcal{S}$ (i.e. $Q_{\mathcal{S}}(X)$). In particular, the scrambler $\mathcal{S}$ first chooses a permutation $\pi : [1..n] \rightarrow [1..n]$ uniformly at random. It then privately realizes the permutation π on the input X, obtaining $\widetilde{X}$ such that $\widetilde{X}[j] = X[\pi[i]]$, while ensuring 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 *Melbourne shuffle* algorithm recently introduced in [36] or a *cascaded mix-network* proposed in [15]. The two approaches have three key differences. First, the former takes X and π as input and outputs $\widetilde{X}$, whereas the latter takes in only X and generates $\widetilde{X}$ without a priori knowledge of π. Second, 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. Since the scrambler is voided if the adversary can correctly guess the permutation π, we wish to make the probability of such even negligible. Regarding this concern, the shuffle algorithm offers higher privacy protection than mix-network, because it reduces the probability of the scrambler being voided to a negligible value (i.e. $\frac{1}{(1-\epsilon)\times n!} \approx \frac{1}{n!}$). Third, time complexity and trusted memory requirement for the shuffle algorithm are $O(n)$ and $O(\sqrt{n})$, respectively. These metrics are lower than those of the cascaded mix-network: $O(n \log n)$ and $O(n^{0.85})$.

We implement the scrambler $\mathcal{S}$ using the Melbourne shuffle algorithm [36]. Specifically, the scrambler $\mathcal{S}$ generates π using a pseudo-random permutation [28] and represents it by a short secret seed. Then, it executes the Melbourn shuffle algorithm with π and X as input, outputting the scrambled data $\widetilde{X}$. The shuffle algorithm is oblivious and consists of two phases: distribution and clean-up. The algorithm is configurable by two variables p_1, p_2 that affect the overall performance and probability of the algorithm restarting[4]. Fortunately, such probability is negligible. Details of the algorithm are provided in Appendix A.

As already discussed, the probability that the adversary can correctly guess the underlying permutation π and thus void the scrambler is negligible. Moreover, since Melbourn shuffle algorithm is oblivious, so long as the encryption scheme in use in not broken, the scrambler $\mathcal{S}$ is secure. Informally, there exists no algorithm $\mathcal{D}$ that can distinguish two inputs X_1, X_2 of the same size by observing the scrambler's access patterns.

[4]due to overflow of memory segments

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 first 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 $Y[i] = X[T[i]]$. The tag T represents the linkage between input and output records. In the running example (Figure 1), $T = \langle 3, 5, 1, 6, 2, 4, 9, 7, 8 \rangle$. Let $Q_{\mathcal{P}}(X)$ be the observable access 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 $\mathcal{S}$ 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 $\mathcal{P}$ 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 $\widetilde{T}_1, \widetilde{T}_2$ tags of the scrambled input $\widetilde{X}_1, \widetilde{X}_2$, respectively. Since the scrambler is data oblivious, $Q_{\mathcal{S}}(X_1) = Q_{\mathcal{S}}(T_1) = Q_{\mathcal{S}}(X_2) = Q_{\mathcal{S}}(T_2)$. Moreover, the security of the scrambler also assures the distributions of $\widetilde{T}_1$ and $\widetilde{T}_2$ are indistinguishable. It follows that $Q_{\mathcal{P}}(\widetilde{T}_1)$ is indistinguishable from $Q_{\mathcal{P}}(\widetilde{T}_2)$. Recall that $Q_{\mathcal{P}}(\widetilde{T}_1) = Q_{\mathcal{P}}(\widetilde{X}_1)$ and $Q_{\mathcal{P}}(\widetilde{T}_2) = Q_{\mathcal{P}}(\widetilde{X}_2)$, $Q_{\mathcal{P}}(\widetilde{X}_1)$ and $Q_{\mathcal{P}}(\widetilde{X}_2)$ are indistinguishable.

Since $\mathcal{A}_P$ executes $\mathcal{S}$ followed by $\mathcal{P}$, $Q_{\mathcal{A}_P}(X_1) = Q_{\mathcal{S}}(X_1)||Q_{\mathcal{P}}(\widetilde{X}_1)$ and $Q_{\mathcal{A}_P}(X_2) = Q_{\mathcal{S}}(X_2)||Q_{\mathcal{P}}(\widetilde{X}_2)$. So far, we have proven that $Q_{\mathcal{S}}(X_1) = Q_{\mathcal{S}}(X_2)$, and $Q_{\mathcal{P}}(\widetilde{X}_1)$ is indistinguishable from $Q_{\mathcal{P}}(\widetilde{X}_2)$. Thus, it follows that $Q_{\mathcal{A}_P}(X_1)$ and $Q_{\mathcal{A}_P}(X_2)$ are computationally indistinguishable. Therefore, $\mathcal{A}_P$ is privacy-preserving.

□

3.2.2 Complex algorithms

We consider complex algorithms which can be decomposed into sequences of substeps. By hybrid arguments [28], PRAMOD enables combining privacy-preserving sub-steps to construct a complex algorithm which is also privacy-preserving. More 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 $\mathcal{A}_{P_{i+1}}$. If every sub-step $\mathcal{A}_{P_i}$ is privacy-preserving, then so is $\mathcal{A}_P$.

COROLLARY 1. *Given two algorithms $\mathcal{P}_1, \mathcal{P}_2$ which are privacy preserving with advantage at most ϵ_1, ϵ_2, respectively. The combined algorithm $\mathcal{P}$ which executes $\mathcal{P}_1$ followed by $\mathcal{P}_2$ is also privacy preserving with advantage at most $\epsilon_1 + \epsilon_2$.*

Proof Sketch: The proof follows from the hybrid arguments. □

3.3 Discussion

One important implication of Theorem 1 is that the output of $\mathcal{A}_P$ is not always the same as that of $\mathcal{P}$, since 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 guarantee the same output, 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 transforms the input, 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 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.

There exists many privacy-preserving algorithms which can be combined into complex algorithms which meet our security definition. As noted earlier, scanning algorithms which go through the input and write the output to the same addresses are privacy-preserving. We will show later that PRAMOD's sort and compaction are privacy-preserving. Existing data-oblivious algorithms proposed in the literature, such as oblivious data expansion [8], are also privacy-preserving. These 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 sub-steps with more efficient privacy-preserving algorithms in PRAMOD. We demonstrate this approach later with the join algorithm (section 4.4).

4. PRIVACY-PRESERVING DATA MANAGEMENT ALGORITHMS

This section describes implementations of four algorithms pSort, pCompact, pAggr and pJoin , illustrating the utilisation of PRAMOD in enabling privacy-preserving data management. Our proposed framework can be applied to derive various efficient and secure query processing algorithms. We leave provisioning a secure full-fledged SQL system as future work.

For clarity, all algorithms assume inputs comprising of key-value records all of which are equal-sized. We denote by $key(x)$ and $value(x)$ the key and value components of record x, respectively. pSort and pCompact achieve security directly using the scrambler, while pAggr and pJoin derive security from that of their sub-steps. For each algorithm, we first explain how it is implemented in the baseline system, then contrast such non-privacy-preserving implementation with the implementation in PRAMOD. Finally, we analyse 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 [29]. 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 is the output buffer. *s-way* merge results in optimal I/O performance because the each record is read only once during merging. This solution, however, leaks the input order as discussed earlier in Section 2.

Algorithm 1 Privacy-Preserving Sort

```
1: procedure SORT(X)
2:     X' ← MakeKeyDistinct(X);
3:     X̃ ← Scramble (X');
4:     Y' ←ExternalMergeSort(X̃);
5:     Y ← 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 pre-processing step appends the address of each record to its key, i.e. $key(x'_i) = key(x_i)||i$, transforming the input X to X' whose keys are distinct. (2) X' is securely permuted by the scrambler, which results in $\widetilde{X}$. (3) $\widetilde{X}$ is sorted into Y'. The comparison function break ties (if any) using the addresses attached to records in step pre-processing step. (4) The post-processing scan through Y' to remove the address information, generating the output Y.

Although this implementation employs merge sort as an underlying sorting algorithm, it can be applied to any other algorithms. This generality is advantageous to our framework: it can adopt the most appropriate and efficient algorithm for the targeted applications.

Performance analysis. The scrambler and merge sort run in $O(n)$ and $O(n \log n)$ time, respectively, therefore pSort runs in $O(n \log n)$. To the best of our knowledge, the most efficient data-oblivious sort algorithms are from Goodrich *et al.* [25, 22]. The deterministic version [25] runs in $O(n \log^2 n)$ time, while the randomized version [22] runs in $O(n \log n)$ time but with large constant factor. pSort attains optimal performance with low constant factor, and is arguably simpler than the data-oblivious alternatives.

4.2 Compaction

The algorithm removes *marked* records from the input. The output contains $n' \leq n$ unmarked records while preserving the original order: if x_i and x_j are to be retained

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

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	$O(n_1 \log n_1 + n_2 \log n_2)$	$O(n_1 \log n_1 + n_2 \log n_2 + l \log l)$	$O(n_1 \log n_1 + n_2 \log n_2 + l \log l)$

and $i < j$, x_i appears before x_j in the output. 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 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 is accessible to the adversary. Keeping n' secret would incur storage overhead and defeat the purpose of compaction.

Baseline solution. We adopt a simple approach which sequentially pulls and decrypts records in the trusted unit, then re-encrypts and writes back to the storage only those marked with 1 while discarding the others. This approach is efficient, but it reveals distribution of the discarded records.

Algorithm 2 Privacy-Preserving Compaction

```
1: procedure COMPACT(X)
2:     X' ← Mark(X)
3:     X̃ ← Scramble (X')
4:     Ỹ ← Filter (X̃)
5:     Y ← Arrange (Ỹ)
                    ▷ 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, it marks retaining records with C_1 and to-be-removed record with C_2. C_1 is incremented by 1 for each retaining record encounters, and C_2 is decremented by 1 for each to-be-removed record met. (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 so that it can arrange 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. Note 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] runs in $O(n \log n)$ time. The proposed pCompact achieves the asymptotically optimal time complexity of $O(n)$ while keeping the constant factor low.

4.3 Grouping and aggregation

The algorithm groups input records based on their keys and then applies an aggregation function, such as summing or averaging, over the group members. Specifically, let $K = \{k_1, k_2, \ldots, k_{n'}\}$ be the set of unique keys, the algorithm outputs $Y = \langle y_1, y_2, \ldots, y_{n'} \rangle$ 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 records based on their keys. Next, the sorted records are scanned, and the aggregate values are accumulated and written out immediately after the last record of each group is encountered. Even if privacy-preserving sort is used, because of this last step, the overall execution reveals the size of each group.

Algorithm 3 Privacy-Preserving Aggregation

```
1: procedure PRIVATESUM(X)
2:     G ← 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 ← v + value(g)
8:             Add ⟨dummy⟩ to V ▷ output dummy records
9:         else
10:            Add ⟨k, v⟩ to V
11:            k ← key(g)
12:            v ← value(g)
13:        end if
14:    end for
15:    Y ← pCompact(V) ▷ Remove all ⟨dummy⟩ from V
16:    return Y
17: end procedure
```

Privacy-preserving solution. Algorithm 3 shows our privacy-preserving algorithm (pAggr) based on a sort, a compaction and a scanning step. pAggr illustrates a special case of grouping with Sum(.) as an aggregation function. It is trivial to modify pAggr to handle all other standard aggregation functions. First, pAggr sorts X using pSort, obtaining G in which records of the same key 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 size of each group, the trusted unit not only outputs a valid aggregation value, but also dummy records. Finally, pCompact is used to remove dummy records in V. Since these 3 steps are all privacy-preserving, from Corollary 1, it follows that pAggr is also privacy-preserving.

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().

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

Performance analysis. pAggr is constructed from pSort and pCompact, thus it runs $O(n \log n)$ time. The data-oblivious algorithm [7] adopts the same workflow, and its time complexity is also $O(n \log n)$.

4.4 Join

Algorithm 4 Privacy-preserving join

```
1: procedure JOIN(X_1, X_2)
2:     X ← X_1||X_2
3:     S ← pSORT(X)
           ▷ tie is broken such that X_1 records always come
   before X_2 records
4:     V_2 ← FRSum(S)
5:     V_1 ← RRSum(S)
6:     W_1 ← pCOMPACT(V_1)
7:     W_2 ← pCOMPACT(V_2)
8:     X_1exp ← OExpand (W_1)
9:     X_2exp ← OExpand (W_2)
10:    Y ← X_1exp · X_2exp
   ▷ stitch expansion of X_1 and X_2 to get the join output
11:    return Y
12: end procedure
```

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 the sake of disposition, we consider a simplified version of join for key-value datasets. It is straight-forward to generalize ours for other standard join algorithms. A record $x_i \in X_1$ matches with another record $x_j \in X_2$ 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 this algorithm can be larger than its input size.

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, the combined leakage from the sorting and 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.* [8]. It consists of two stages. The first stage computes the degree of each record in the join graph. The second stage 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 data-oblivious substeps in [8] for computing record degrees with pSort, pCompact and two linear scans (line 2-7). However, it uses the data-oblivious expansion algorithm without change (line 8-9). For every step is privacy-preserving, it follows from Corollary 1 that pJoin is also 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 which have the same key, 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 a *weight* value of 1 while X_2 record has a weight value of 0. It scans X from left to right and associates with each record the running sum of weights in its 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, assuming weight values of 0 for X_1 records and 1 for X_2 records. At the end of this second pass, X_1's records are associated with theirs 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 [8] directly to implement this step. The results are then stitched together via a linear scan to generate the final output Y. Table 3 gives a detailed 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 as that of the oblivious algorithm [8]. However, we show later in Section 5 that pJoin has lower running time because pSort and pCompact algorithm are more efficient than the corresponding data-oblivious components.

5. PERFORMANCE EVALUATION

This section reports the performance of the four algorithms described in the last section: pSort, pCompact, pAggr and pJoin. We generate input data following Yahoo! TeraSort benchmark [37], in which each record com-

Table 4: Overall running time (in seconds) of our PRAMOD's algorithms in comparison with: (1) implementation in the baseline system with weaker security (2) data-oblivious algorithms with same level of security.

Algorithm	*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×)

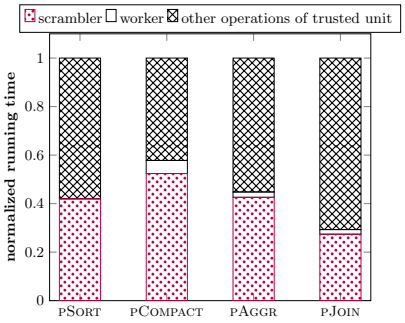


Figure 4: Normalized running time breakdowns for PRAMOD's algorithms. The overall cost consists of the time taken by the scrambler, plus the time required by the worker (if any). The rest is taken by pre-processing, post-processing steps and the (non-privacy-preserving) algorithm itself.

prises of a 10-byte key and a 90-byte value. We encrypt the records with AES-GCM using a 256-bit key, generating 132-byte ciphertexts. Experiments are run on a DELL workstation equipped with an Intel i5-4570 3.2GHz CPU and a 500GB SATA disk. We assign 64MB[5] of secure memory to the trusted unit, and vary the input size from 8GB to 64GB. Our implementations use Crypto++[6]for cryptographic operations. We repeat each experiment 10 times and report the average result.

5.1 Cost of Security

We first compare PRAMOD's algorithms with the alternatives in the baseline system. As noted in Section 4, the baseline implementations reveal information about the input via access patterns. We then compare them with data-oblivious solutions: oblSort for sorting [25], oblCompact for compaction [22], oblAggr for group aggregation [7] and oblJoin for join [8].

Overhead. Table 4 quantifies the execution time

[5]We have also run other sets of experiments with various secure memory capacities (e.g. 128MB and 256MB). We find that varying the secure memory size within the small range does not affect the overall running time of all algorithms. On the other hand, assigning much larger secure memory for the trusted unit, say a few GB, will improve the performance. However, since we consider scenario in which $m = O(\sqrt{n})$, this option is ruled out

[6]Crypto++ library. `http://www.cryptopp.com`

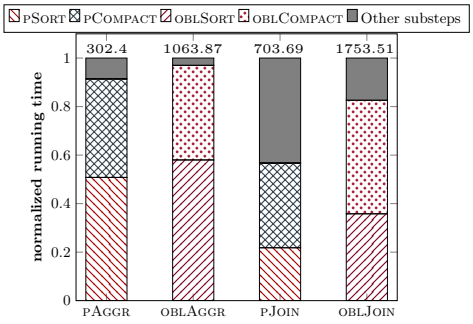


Figure 5: Normalized running time breakdowns for group aggregation and join algorithms. The running time consists of the time taken by sort, compaction. The rest is taken by other substeps. The value on top of each bar indicates the total running time of each algorithm (in minutes).

for 32GB inputs (or $n = 2^{28}$ records[7]). It shows that PRAMOD incurs overheads between 2.43× to 4.99× over the baseline system. This cost of security is admittedly considerable. However, we argue that PRAMOD is still practical, given the overheads of existing data-oblivious solutions which offer the similar security level are between 9.95× to 15.85×.

Cost breakdown. To better understand factors that contribute to the overheads, we measured the time taken by the scrambler, by the worker (if any) and by other operations in the trusted unit. The last factor includes the time spent on pre-processing, post-processing and on the main algorithm logic. Figure 4 shows this breakdown. It can be seen consistently across all algorithms that the cost of scrambling is significant. Particularly, it contributes 42%, 52.3%, 42.6% and 27.4% to the overall cost of pSort, pCompact, pAggr and pJoin, respectively. The untrusted worker accounts for small proportion of the total running time, between 1.8% (for pJoin) to 5.4% (for pCompact). This is because the worker does not perform cryptographic operations which are computationally expensive.

As the security of group aggregation and join algorithms are derived from that of sort and compaction algorithms, we also report cost break-down of the two algorithms with respect to sort, compaction and other sub-steps in Figure 5. The superior performance of pSort and pCompact helps pAggr and pJoin achieve security with low cost. Besides sort and compaction, other sub-steps (e.g. oblivious expansion step in join (see section 4.4)) of the data-oblivious algorithms and ours have the same running time. This reasons why the contributions of sort and compaction steps to the overall running time of pAggr and pJoin are less than that of the oblivious alternatives ($21.7 - 50.8\%$ vs. $36.6 - 58.9\%$ for sort and $34.8 - 40.7\%$ vs. $38.5 - 46.8\%$ for compaction) while the corresponding portion of other sub-steps are larger in ours compared to oblAggr and oblJoin (Figure 5).

Re-Encryptions and I/O complexity. Note that cryptographic operations (re-encryption) are CPU intensive and communication between the trusted unit and the storage is expensive since it involves I/O. Table 5 details the costs of these two operations. Observe that PRAMOD's algorithms require $O(n)$ I/Os with a small con-

[7]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. $n = |X_1| + |X_2|$)

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, $p_1 = p_2 = 2$. 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, $d = 3$.

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]	$(1 + \log n) \cdot n$	$O(n \log n)$
pAggr	$(2p_1 + 2p_2 + 8) \cdot n$	$O(n)$
oblAggregate[7]	$(\sum_{i=1}^{\log s} i + \log s + \log n + 3) \cdot n$	$O(n \log^2 n)$
pJoin	$(3p_1 + 3p_2 + 9 + d) \cdot n$	$O(dn)$
oblJoin[8]	$(\sum_{i=1}^{\log s} i + \log s + 2 \log n + 5 + d) \cdot n$	$O((dn) \log(dn) + n \log^2 n)$

stant factor, whereas all data-oblivious algorithms, except for oblCompact, require $O(n \log^2 n)$ I/Os. For join algorithms, I/O complexity depends on d, the average record degree in the join graph. For uniformly distributed datasets, d can be considered as a constant (we assumed $d = 3$ in our experiments).

Recall that a record is re-encrypted every time it leaves the trusted unit, hence the number of re-encryptions is proportional to the I/O complexity. Table 5 gives the exact number of re-encryptions in each algorithm. These numbers depend on specific configuration of each algorithm. More specifically, the scrambler performs $(p_1+p_2+1) \times n$ re-encryptions in scrambling n records. In our experiments, we find that for the datasets under consideration, with $p_1 = p_2 = 2$, the scrambler achieves optimal running time and negligible probability of restarting[8] . On the other hand, the numbers of re-encryptions of oblivious algorithms depend only on the size of the private memory. We observe that given secure memory size of $m = O(\sqrt{n})$ and the same input, the oblivious algorithms perform a few times more re-encryptions than PRAMOD's privacy-preserving algorithms, which directly translates to considerable running time overheads.

5.2 Efficiency and Scalability

We evaluate how our algorithms scale with larger input sizes. Figure 6 reports the running time in log-scale. Our algorithms outperform the data-oblivious alternatives for all input sizes. More specifically, pSort outperforms oblSort [25] by $2.6 - 4.4\times$ (Figure 6a). Similarly, pCompact is $3 - 3.5\times$ faster than oblCompact [22] (Figure 6b), pAggr outruns oblAggr [7] by $2.7 - 3.8\times$ (Figure 6c), and pJoin is $2 - 2.6\times$ more efficient than oblJoin [8] (Figure 6d). The reason for the outperformning of PRAMOD's algorithms lies in the fact that they requires less number of re-encryptions and I/O accesses than the data-oblivious algorithms (see Table 5). Recall that re-encryptions are computationally extensive and I/O are low, the differences mentioned above significantly affect the running time.

It is worth noting that the speedup becomes more evident with larger inputs: from $2 - 3\times$ for 8GB datasets to $2.6 - 4.4\times$ for 64GB datasets. This suggest PRAMOD's algorithms are more efficient and scalable than the class of data-oblivious alternatives.

[8]From the Appendix A, with $p_1 = p_2 = 2$ and $n = 2^{28}$, the probability that the scrambler need to restart is $Pr_{restart} = 5.3530 \times 10^{-70}$

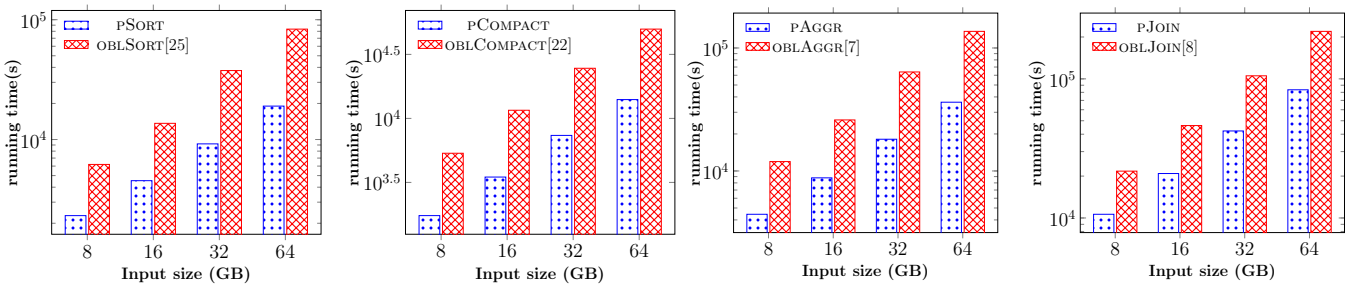
Discussion. Although PRAMOD currently runs on a single machine, we stress that it is straight forward to port it to a distributed environment. The scrambler processes data in blocks independently of each other, which lends itself naturally to distributed setting. We note that distributing the scrambler's workload to multiple nodes could result in substantial speed-up because the scrambling process is CPU intensive. In fact, our implementations are multi-threaded (4 threads), and in comparison with the single-thread version, we observe $1.8\times$ speed-up. Furthermore, most external-memory algorithms are often designed to support parallelism. Therefore, we believe that parallelization will significantly lower the execution time in PRAMOD. On the other hand, it is difficult to parallelize oblivious algorithms due to their complexity.

6. RELATED WORK

Secure Data Management using Trusted Hardware. Several systems have used trusted computing hardware such as IBM 4764 PCI-X [2] or Intel SGX [3] to enable secure data management. TrustedDB [9] presents a secure outsourced database prototype which leverages on IBM 4764 secure CPU (SCPU) for privacy-preserving SQL queries. Cipherbase [7] extends TrustedDB's idea to offer a full-fledged SQL database system with data confidentiality. VC^3 employs Intel SGX processors to build a general-purposed data analytics system. In particular, it supports MapReduce computations, and protects both data and the code inside SGX's enclaves. However, these systems do not meet our security definition, i.e. they offer a weaker security guarantee.

Recent systems [18, 35] adopt a similar approach to this paper's to support privacy-preserving computation. However, they focus on the MapReduce computation model, and specifically use scrambling to ensure security for the shuffling phase (which is essentially a sort algorithm). PRAMOD is a more general framework which supports many other algorithms.

Secure Computation by Data Oblivious Technique. Oblivious-RAM [21] enables secure and oblivious computation by hiding data read/write patterns during program execution. ORAM techniques [39, 13, 25] trust a CPU with limited internal memory, while user programs and data are stored encrypted on the untrusted server. Security is achieved by making data accesses to the server appear random and irrelevant to the true and intended access



(a) Sorting (b) Compaction (c) Group-Aggregation (d) Join

Figure 6: 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).

sequences. A non-oblivious algorithm can be made data-oblivious by adopting ORAM directly, but this approach leads to performance overhead of at least a $O(\log n)$ multiplicative factor. PRAMOD offers a similar level of security with only an $O(n)$ additive overhead.

Another line of works advocate 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 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 breadth-first search [12]. However, these algorithms are application-specific and less efficient than PRAMOD's algorithms.

7. CONCLUSION

In this paper, we have described PRAMOD, a framework for enabling efficient and privacy-preserving data management algorithms using trusted computing with limited secure memory. We show that for many algorithms, prepending them with a scrambling step make the algorithms privacy preserving. PRAMOD achieves security for other complex algorithms by decomposing them into smaller privacy-preserving substeps. We demonstrated four algorithms: pSort for sorting, pCompact for compaction, pAggr for group aggregation and pJoin for join. We showed experimentally that the algorithms are efficient and scalable, outperforming the corresponding data-oblivious algorithms which offer a similar level of privacy protection.

8. REFERENCES

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

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

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

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

[5] 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.

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

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

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

[9] A. D. Barbour, L. Holst, and S. Janson. *Poisson approximation*. Clarendon Press Oxford, 1992.

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

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

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

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

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

[15] 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*.

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

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

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

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

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

[21] M. T. Goodrich. Data-oblivious external-memory algorithms for the compaction, selection, and sorting of outsourced data. SPAA '11.
[22] M. T. Goodrich. Randomized shellsort: A simple data-oblivious sorting algorithm. *J. ACM*, 2011.
[23] M. T. Goodrich. Zig-zag sort: A simple deterministic data-oblivious sorting algorithm running in o(n log n) time. *CoRR*, 2014.
[24] M. T. Goodrich and M. Mitzenmacher. Privacy-preserving access of outsourced data via oblivious ram simulation. *CoRR*, abs/1007.1259, 2010.
[25] M. T. Goodrich, O. Ohrimenko, and R. Tamassia. Data-oblivious graph drawing model and algorithms. *arXiv preprint arXiv:1209.0756*, 2012.
[26] A. Halevy, A. Rajaraman, and J. Ordille. Data integration: The teenage years. In *VLDB*, 2006.
[27] J. Katz and Y. Lindell. *Introduction to modern cryptography*. CRC Press, 2014.
[28] D. Knuth. The art of computer programming 1: Fundamental algorithms 2: Seminumerical algorithms 3: Sorting and searching, 1968.
[29] A. Lakshman and P. Malik. Cassandra: a decentralized structured storage system. *Operating Systems Review*, 44, 2010.
[30] F. Li, M. Hadjieleftheriou, G. Kollios, and L. Reyzin. Dynamic authenticated index structure for outsourced databases. In *SIGMOD*, 2006.
[31] J. M. McCun, B. Parno, A. Perrig, M. K. Reiter, and H. Isozaki. Flicker: An execution infrastructure for tcb minimization. In *EuroSys*, 2008.
[32] 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.
[33] M. Naveed, S. Kamara, and C. V. Wright. Inference attacks on property-preserving encrypted databases. In *CCS*, 2015.
[34] O. Ohrimenko, M. Costa, C. Fournet, C. Gkantsidis, M. Kohlweiss, and D. Sharma. Observing and preventing leakage in mapreduce. In *CCS*, 2015.
[35] 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.
[36] O. OMalley and A. C. Murthy. Winning a 60 second dash with a yellow elephant. *Proceedings of sort benchmark*, 2009.
[37] P. Paillier. Public-key cryptosystems based on composite degree residuosity classes. In *EUROCRYPT*, 1999.
[38] B. Pinkas and T. Reinman. Oblivious ram revisited. In *CRYPTO, 2010*.
[39] 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.
[40] R. A. Popa, C. Redfield, N. Zeldovich, and H. Balakrishnan. Cryptdb: protecting confidentiality with encrypted query processing. In *SOSP*, 2011.
[41] 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*.
[42] S. Specification and T. P. P. C. TPC. Tpc benchmark ds. 2012.
[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.

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 algorithm takes in a randomly chosen permutation π and a data set X of n items. The permutation π is generated using a pseudo random permutation [28], and represented by 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 $p_1, p_2 > 1$. 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 aborts and restarts. Using Poisson Approximation [10] and the result from [36], the probability that the algorithm restarts is:

$$Pr_{restart} \leq 2n^{3/4}\left(\frac{e^{p_1}}{p_1^{p_1\sqrt[4]{n}}} + \frac{e^{p_2}}{p_2^{p_2\sqrt[4]{n}}}\right)$$