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
Prateek Saxena, Shruti Tople
School of Computing, National University of Singapore
{hungdang,dinhtta,changec,oiibc,prateeks,shruti90}@comp.nus.edu.sg

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. Achieving strong security under additional constraints of functionality and performance is even more challenging, 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 secure 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 assuring a weaker level of security. More remarkably, PRAMOD shows superior performance in comparison with the state-of-the-art solutions offering 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, especially in the context of outsourced database in the past [45, 46, 26], data privacy in this era of Big Data faces new challenges. First, the cloud providers have more incentives in extracting content of the outsourced data for its commercial values [42, 43]. Second, even when the providers are trusted, multi-tenancy, complexity of software stacks, and distributed computing models continue to enlarge the attack surface [15, 17]. 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 [16, 8].

The first step towards securing the data is to encrypt it before outsourcing to the cloud. Unfortunately, this only protects data at rest [30]. Fully homomorphic encryption allows computation over encrypted data, but it suffers from prohibitive performance [20, 13]. Partially homomorphic encryption schemes [37, 19] are more practical, but they are limited in the range of supported operations [39, 44], and have been shown to be vulnerable to attacks [33]. Consequently, some recent works have advocated an approach of combining encryption with trusted computing primitives [10, 40, 6], in which confidentiality and integrity protected execution environments are provisioned by hardware (e.g. Intel SGX [1]) or by hardware-software combination [31, 32]. In such a secure environment, computations are performed on decrypted data, and the results are encrypted before being returned. However, the remedy is yet to come. 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, which can leak information about the data [41, 15, 17]. 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] [38, 24, 41]. 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 the 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 processes, 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 utilizing a component named *scrambler*. The scrambler randomly and securely permutes input data of size n in $O(n)$ time. We demonstrate PRAMOD by constructing four popular data management algorithms: *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 [5, 29, 4]. Join is arguably one of the most important operations 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 [2].

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 the original input. This approach to security — scrambling the input before executing the algorithm — leads to two important results. First, its performance is superior to that of generic ORAM solutions, because its overhead factor is additive rather than multiplicative. Second, it generalizes to all algorithms implementing the same application. This allows PRAMOD to 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 sorting algorithms) is simpler than data-oblivious external sort algorithms [24], 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. For a complex algorithm made up of a sequence of sub-steps, there will be no access pattern leakage if none of the sub-steps leak information. This allows PRAMOD to achieve security for group aggregation and join algorithms by implementing them based on sort and compaction.

We implement pSort, pCompact, pAggr and pJoin, evaluate their performances and study the costs of security. Compared with the 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 [24, 22, 6, 7] 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 PRAMOD – the framework for deriving 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 sub-steps so long as each of which is privacy-preserving.

3. We demonstrate PRAMOD by constructing pSort, pCompact, pAggr and pJoin, and analysing their complexity. These algorithms are not only privacy-preserving but also 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 PRAMOD's algorithms against the baseline and state-of-the-art data-oblivious alternatives. The results demonstrate reasonable overheads over the less secure (baseline) approach, and running time speedup of upto 4.4× over corresponding data-oblivious alternatives with similar level of security.

Next section describes the security model and defines the problem that we are solving and rationale behind the proposed framework. Section 3 presents PRAMOD and its associated components. Section 4 demonstrates the proposed framework by describing constructions of four data management algorithms and discusses their theoretical analysis. Our experimental evaluation of PRAMOD is reported in Section 5. After that, we discusses related work in Section 6 before concluding our work in Section 7.

2. PROBLEM DEFINITION

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

Running Example. Let us consider a user storing her data consisting of integer-value records on the cloud. Due to privacy concerns, the user encrypts the data so that no untrusted party can learn its content. She then wishes to sort

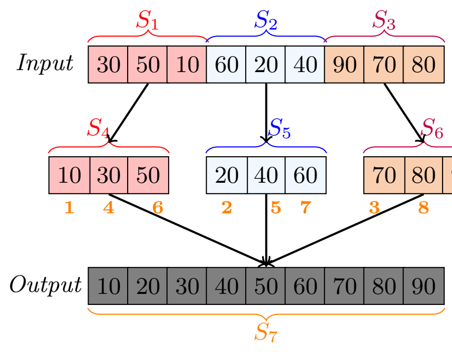


Figure 1: An example of three-way external merge sort. Each square denotes an encrypted record. Black numbers represent integer-value of the records which are invisible to untrusted parties due to the deployment of secure encryption. Orange numbers denote the order in which each encrypted record is read into the trusted unit during the merging.

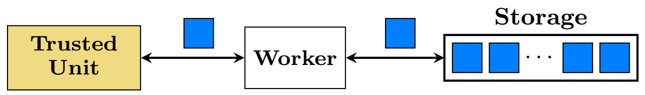


Figure 2: Computation model of a cloud server, consisting of a trusted unit capable of processing a limited number of records at a time. Storage is untrusted, and its communication with the trusted unit is mediated by an untrusted worker (honest-but-curious). Only the trusted unit can see and compute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

the data, as a pre-processing step for other tasks such as database loading, ranking, de-duplication, etc. Although sorting directly over encrypted data is possible to a certain extent, it is highly impractical [3]. To efficiently sort the data, the user relies on 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 the secure memory is limited in size, the user must employ an external, k-way merge sort algorithm. Figure 1 depicts a simple example of three-way merge sort in which the secure memory is limited to holding only three records at a time. The input consists of nine records, and sorting involves one 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. Let $key(x)$ and $value(x)$ denote the key and value component of a record x, respectively. The data records are protected by a semantically secure encryption scheme. Figure 2 details the cloud's computation model which consists of a *trusted unit*, a *worker* and a storage component. The user grants her trust on the trusted unit, which can only hold up to m records at a time. We assume that m is at least $\sqrt{n}$. This is a reasonable assumption since in practical scenarios, the size of the secure memory can be in orders of megabytes. The worker, which mediates access to the untrusted cloud storage and is also responsible for other house-keeping tasks, is not trusted. Both the worker and the 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) adversary who tries 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 adversaries who deviate arbitrarily from the expected computation is a complex and different problem by itself. We refer readers to recent works demonstrating effective defences against the active adversaries [17, 40].

We assume that the worker and storage component are under the adversary's control, while the trusted unit is sufficiently protected. Specifically, the trusted unit corresponds to the user's TCB, while 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 unit, 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 executes the algorithm in two phases. In the first phase, it independently sorts each block in-memory and returns three sorted, encrypted blocks. Next, it performs three-way merge: at most three records are kept in the secure memory at any time. They are pulled from the sorted blocks with help from the worker. In this the running example, the adversary observes that the trusted unit first takes one record from each sorted block, writes one record out, then takes another record from the first block in. Although it cannot learn the records' content, it can still infer that the smallest record comes from S_1. Such inference in general can reveal the relative order of the records from different blocks. For algorithms taking data from different sources, this leakage can expose the sources' identities.

2.2 Problem Definition

In this paper, we concern privacy-preserving data management algorithms using trusted computing with limited secure memory. This secure memory is limited in a sense that it can hold (and process) at most only m 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. One solution is to employ oblivious RAM [41] directly on the storage backend. However, this approach incurs an overhead of at least $O(\log n)$ per each access, rendering it impractical for big data processing. Another option is to use application-specific data-oblivious algorithms such as oblivious sort [24]. Nevertheless, they are convoluted and do not generalize well to other algorithms. Thus, the second goal of our work is to attain a simple design with low performance overhead.

2.3 Security Definition

We now describe our formal security definition that admits only the disclosure of the input and output sizes. Let $Q^m_{\mathcal{P}}(X) = \langle q_1, q_2, \dots, q_z \rangle$ be the access (read/write or I/O) sequence observed by the adversary during the computation of $\mathcal{P}$ on X using the trusted unit with secure memory of size m. In the baseline system, $Q^m_{\mathcal{P}}(X)$ represents the sequence of I/O requests made by the trusted units to the worker. Hereafter, unless stated otherwise, we assume $m > \sqrt{n}$ and simply denote the trusted unit's access sequence by $Q_{\mathcal{P}}(X)$. Each q_i is a 5-value tuple $\langle op,\ addr,\ val,\ time,\ info \rangle$ 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[3] 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):

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

where t_i represents the request time, and $S_i + j, e(S_i + j)$ denote 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 adversary can learn information about X. Using the well-accepted notion of *indistinguishability* in the literature [28], our security definition dictates that $Q_{\mathcal{P}}(X)$ reveals nothing beyond the input and output sizes. Specifically:

DEFINITION 1 (PRIVACY-PRESERVING ALGORITHM). *An algorithm $\mathcal{P}$ is privacy-preserving if for any two datasets X_1, X_2 of the same size, $Q_{\mathcal{P}}(X_1)$ is computationally indistinguishable from $Q_{\mathcal{P}}(X_2)$.*

Informally, the definition requires that for any two equal-sized inputs, the observed I/O sequences are *similar*, and thus reveal no sensitive information about their inputs. This also implies the two inputs induce equal-sized outputs, otherwise the I/O sequences would have been different. In addition, note subtly that if the records are not encrypted, $Q_{\mathcal{P}}(X_1)$ and $Q_{\mathcal{P}}(X_2)$ are immediately distinguishable for their tuples contain the records in clear-text.

Discussion. A similar definition, *data obliviousness*, requires a data-oblivious algorithm $\mathcal{P}$ to incur the same access sequence on any two inputs X_1, X_2 of the same size,

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

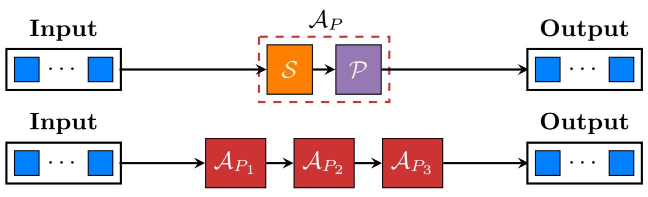


Figure 3: PRAMOD constructs privacy-preserving algorithm $\mathcal{A}_{\mathcal{P}}$ from an algorithm $\mathcal{P}$ by appending it with the scrambler $\mathcal{S}$. The proposed framework can also combine simple privacy-preserving algorithms to build a more complex algorithm which is also privacy-preserving.

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 assuring negligible leakage against computationally bounded adversaries. Data-oblivious algorithms share with ours the assumption that data records are always protected and the adversary cannot see their content, relying on the deployment of secure encryption. However, practical encryption schemes cannot achieve perfect secrecy. Therefore, in practice, both data-oblivious algorithms and privacy-preserving algorithms satisfying Definition 1 are expected to offer similar levels of privacy-protection.

Oblivious algorithms, such as one which sequentially scans through the entire input, reads and then immediately writes records to the same addresses for instance, are inherently privacy-preserving. On contrary, the k-way merge sort in the running example is not privacy-preserving because there exists X_1, X_2 such that $Q_{\mathcal{P}}(X_1) \neq Q_{\mathcal{P}}(X_2)$ ((e.g. $X_1 = \langle 1, 2, 3, 4, 5, 6, 7, 8, 9 \rangle$ and $X_2 = \langle 3, 5, 1, 6, 2, 4, 9, 7, 8 \rangle$).

3. PRAMOD

In this section, we present the PRAMOD framework, focusing on a new component — the *scrambler*. We explain how to design simple privacy-preserving algorithms, for examples sort and compaction, using this component. We then discuss how to build more complex yet still secure 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 a 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, such as merge sort in the running example, may not be privacy-preserving. PRAMOD realizes privacy-preserving algorithms in two ways. First, for algorithm $\mathcal{P}$ which essentially re-arranges the input, it simply prepends $\mathcal{P}$ with the scrambler $\mathcal{S}$ which randomly permutes the input before applying $\mathcal{P}$ on the output of the scrambler (Figure 3[a]). To ensure overall security, however, the scrambler $\mathcal{S}$ must not leak any information during its execution. Second, PRAMOD allows for complex, privacy-preserving algorithms to be built from a sequence of privacy-preserving sub-steps $\mathcal{A}_{P_1}, \mathcal{A}_{P_2}, \dots$ (Figure 3[b]). We note that for complex algorithms, input scrambling alone is not sufficient to assure our security definition. For example, consider a group aggregation algorithm which first arranges data records into groups based on their keys and then aggregates the values of the records in each group. Even if the input is scrambled beforehand, the adversary can still learn information about size of each group, which is beyond the leakage admitted by our definition (i.e. 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, $\mathcal{S}$ 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 $\widetilde{X}[\pi[i]] = X[i]$, 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, as the adversary is able to observe all read/write accesses on the storage.

The scrambler can be implemented using either the *Melbourne shuffle* algorithm recently introduced in [35] or a *cascaded mix-network* proposed in [14]. 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. Although the cascaded mix-network seems able to produce the scrambled data whose distribution is statistically close to random permutation within constant number of rounds, the best known bound requires $O(n^{0.85})$ trusted memory [14]. In contrast, the Melbourne shuffle algorithm is able to successfully scramble the data with high probability within constant number of rounds, using only $O(\sqrt{n})$ trusted memory [35].

We implement $\mathcal{S}$ using the Melbourne shuffle algorithm. Specifically, $\mathcal{S}$ first privately generates the permutation π using a pseudo-random permutation [28]. Then, it executes the Melbourne 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 its overall performance. With a negligible probability, the algorithm needs to aborts its execution and restarts. Note that this does not imply failure, but rather longer running time. Details of the shuffle algorithm are provided in Appendix A, which also shows the probability of restarting is negligible.

Let us now consider the security of the scrambler. Since the adversary's guess of π in unlikely to be correct[4], it has to rely on the access pattern $Q_{\mathcal{S}}(X)$ to reveal the linkage between X and $\widetilde{X}$. However, from the security of the Melbourne shuffle algorithm, no adversary can reveal π by observing $Q_{\mathcal{S}}(X)$. Therefore, the scrambler is secure.

3.2 Privacy-Preserving Algorithms

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

3.2.1 Basic algorithms

Let us now consider basic 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 and $\mathcal{A}_P$ denote the algorithm derived from $\mathcal{P}$ by prepending it with the scrambler. More specifically, $\mathcal{A}_P$ first scrambles X using $\mathcal{S}$ and then applies $\mathcal{P}$ on the scrambled input $\widetilde{X}$ to obtain the desired output. We show that if $Q_{\mathcal{P}}(X) = Q_{\mathcal{P}}(T)$, then $\mathcal{A}_P$ is privacy-preserving.

THEOREM 1. *Given an algorithm $\mathcal{P}$, if for any input X, the read/write sequence generated by $\mathcal{P}$ on X is the same as the read/write sequence generated by $\mathcal{P}$ on T where T is the corresponding tag of X, then the derived algorithm $\mathcal{A}_P$ is privacy-preserving.*

Proof Sketch: Let $\mathcal{A}^*_P$ be an algorithm that is the same as $\mathcal{A}_P$, except for one step which reveals the corresponding tag $\widetilde{T}$ of the scrambled input $\widetilde{X}$. We shall prove by contradiction that $\mathcal{A}^*_P$ is privacy-preserving. Since $\mathcal{A}^*_P$ reveals more than $\mathcal{A}_P$, if $\mathcal{A}^*_P$ is privacy-preserving, then so is $\mathcal{A}_P$.

Let us consider two equal-sized inputs X_1, 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. Assume that $\mathcal{A}^*_P$ is not privacy-preserving, there exists an algorithm $\mathcal{D}^*$ that distinguishes $Q_{\mathcal{A}^*_P}(X_1)$ and $Q_{\mathcal{A}^*_P}(X_2)$. We then construct another algorithm $\mathcal{D}$ that breaks the security of the scrambler $\mathcal{S}$ by distinguishing $\widetilde{X}_1$ and $\widetilde{X}_2$.

Given $\widetilde{X}_1$ and $\widetilde{X}_2$, $\mathcal{D}$ first executes $\mathcal{P}$ on $\widetilde{X}_1$ and $\widetilde{X}_2$. Since $\mathcal{P}$ is not privacy-preserving, $\mathcal{D}$ can learn the scrambled tags $\widetilde{T}_1, \widetilde{T}_2$ from $Q_{\mathcal{P}}(\widetilde{X}_1)$ and $Q_{\mathcal{P}}(\widetilde{X}_1)$. $\mathcal{D}$ combines $Q_{\mathcal{S}}(X_1)$ with $Q_{\mathcal{P}}(\widetilde{X}_1)$ and $Q_{\mathcal{S}}(X_2)$ with $Q_{\mathcal{P}}(\widetilde{X}_2)$, getting $Q_{\mathcal{A}_P}(X_1)$ and $Q_{\mathcal{A}_P}(X_2)$. Recall that $\mathcal{A}^*_P$ is the same as $\mathcal{A}_P$ except for the step revealing $\widetilde{T}$, $\mathcal{D}$ can executes $\mathcal{D}^*$ with two inputs $(Q_{\mathcal{A}_P}(X_1), \widetilde{T_1})$ and $(Q_{\mathcal{A}_P}(X_2), \widetilde{T_2})$ and relies on $\mathcal{D}^*$'s output to distinguish $\widetilde{X}_1$ and $\widetilde{X}_2$. However, as discussed in Section 3.1, the scrambler is secure and the assumption of the existence of $\mathcal{D}$ cannot hold. By contradiction, we conclude that $\mathcal{A}^*_P$ is privacy-preserving and so is $\mathcal{A}_P$. □

3.2.2 Complex algorithms

We consider complex algorithms which can be decomposed into sequences of sub-steps. By hybrid argument [28], PRAMOD allows a complex algorithm to derive security from that of its sub-steps. Specifically, let $\mathcal{A}_P = (\mathcal{A}_{P_1}; \mathcal{A}_{P_2}; ..; \mathcal{A}_{P_k})$ be the algorithm consisting of k sub-steps, 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 privacy-preserving algorithms $\mathcal{P}_1$ and $\mathcal{P}_2$, the combined algorithm $\mathcal{P}$ which first executes $\mathcal{P}_1$ on the input and then applies $\mathcal{P}_2$ on the output of $\mathcal{P}_1$ is also privacy-preserving.*

Proof Sketch: This can be proved using the hybrid argument [28]. □

Note subtly that this corollary can only be applied a polynomial number of times as opposed to repeated arbitrarily. In other words, the number of combined sub-steps k cannot be arbitrarily large. Interested readers can refer to [28] for a detailed discussion. Nevertheless, this restriction does not affect the utilization of PRAMOD in practice, for practical algorithms do not contain an exponentially large number of sub-steps.

[4]The probability of such event is $Pr = 1/((1 - negl(n)) \times n!)$ in which $negl(n)$ is a negligible function in n

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 with $X = \langle 0_0, 0_1, 0_2, 0_3, 0_4, 0_5 \rangle$ where the subscripts indicate the original positions in the input. The output $\mathcal{P}(X) = \langle 0_0, 0_3, 0_1, 0_4, 0_2, 0_5 \rangle$, and $\mathcal{A}_P(X) = \langle 0_0, 0_2, 0_1, 0_5, 0_3, 0_4 \rangle$ for a certain permutation generated by the scrambler. It can be seen that when $\mathcal{P}$ is invariant to input permutation (i.e. $\mathcal{P}(X) = \mathcal{P}(X')$ where X' is a permutation of X), the outputs of $\mathcal{P}$ and $\mathcal{A}_P$ are the same. In practice, this condition 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.

There are many privacy-preserving algorithms which can be combined to build complex algorithms that meet our security definition. A set of algorithms which scan through the entire input and write the output to the same addresses is one example. Sort and compaction implemented in PRAMOD are also privacy-preserving, as we will show later. Other examples include existing data-oblivious algorithms, such as oblivious data expansion [7], which can be ported directly to PRAMOD. In fact, for complex algorithms for which data-oblivious implementations exist, we can re-use the 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).

Although the algorithms considered so far are deterministic, Theorem 1 also generalizes to probabilistic algorithms such as quick sort. Essentially, they can be modified to take the random choices as additional input, thus making them deterministic and to which Theorem 1 is applicable.

4. PRIVACY-PRESERVING DATA MANAGEMENT ALGORITHMS

In this section, we provides examples of privacy-preserving data management algorithms constructed using PRAMOD framework: pSort, pCompact, pAggr and pJoin. pSort and pCompact achieve security directly using the scrambler (as discussed in Section 3.2.1), while pAggr and pJoin derive security from that of their sub-steps (as presented in Section 3.2.2).

For each algorithm, we first explain its variant in the baseline system, then contrast it to the PRAMOD's version. Finally, we analyse the performance of different alternatives, the results of which are summarized in Table 1.

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 [18] algorithm. First, the input is divided into $s = n/m$ blocks ($s < m$). Each block is sorted entirely inside the trusted unit. Next, all s sorted blocks are combined in 1 merge step using s-way merge. In this process, 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 the output buffer. *s-way* merge results in optimal I/O performance because each record is read only once during merging. This implementation, however, leaks the input order as discussed earlier in 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 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 and becomes Y'. The comparison function break ties (if any) using the addresses attached to record keys in the pre-processing step. (4) The post-processing step scans through Y' and removes the address information, generating the final output Y.

Although pSort employs merge sort as an underlying sorting algorithm, it can use other sorting algorithms as well. This generality is advantageous to PRAMOD: 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 sorting algorithms are from Goodrich *et al.* [24, 22], among which the deterministic version [24] runs in $O(n \log^2 n)$ time, and the randomized version [22] runs in $O(n \log n)$ time with a 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 and $i < j$, x_i appears before x_j in the output. For simplicity, 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 leakage is acceptable, because the purpose of the algorithm is to reduce the number of records stored on the storage accessible to the adversary. Keeping n' secret would incur storage overhead and defeat the purpose of compaction.

Baseline solution. A straightforward implementation sequentially pulls and decrypts records in the trusted unit, then re-encrypts and writes back to the storage only those

Table 1: Comparison of time complexity of different algorithms. 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	$O(n_1 \log n_1 + n_2 \log n_2)$	$O(n_1 \log n_1 + n_2 \log n_2 + l \log l)$	$O(n_1 \log n_1 + n_2 \log n_2 + l \log l)$

marked with 1 while discarding the others. This solution is efficient, but it reveals distribution 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 initializes two counters, $C_1 = 0$ and $C_2 = n$. Scanning through X, it marks each record with C_1 or C_2 if it is to be retained or removed, respectively. C_1 is then incremented by 1, while C_2 is decremented by 1 for each mark. (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 move records to their desired positions. The worker can finish this step by one linear scan through $\widetilde{Y}$, instead of sorting $\widetilde{Y}$. Although the worker can learn the marking of records, such leakage does not compromise privacy as we argued earlier.

Performance analysis. Every step of pCompact runs in linear time, thus the algorithm runs in $O(n)$. The data-oblivious algorithm for compaction [22] runs in $O(n \log n)$ time. Interestingly, 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 $K = \{k_1, k_2, \dots, k_{n'}\}$ be the set of unique keys in X, the algorithm outputs $Y = \langle y_1, y_2, \dots, y_{n'} \rangle$ in which $key(y_i) = k_i$ and $value(y_i) = \text{Agg}(value(x) : x \in X; key(x) = k_i)$.

Baseline solution. First, records are sorted by their keys. Then, the sorted records are scanned, and the aggregate values are accumulated and written out immediately after the last record of each group is encountered. Because of this last step, the overall execution reveals the size of each group even if a privacy-preserving sorting algorithm is used.

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 record
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
```

Privacy-preserving solution. Algorithm 3 shows the 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 easy to modify pAggr to accommodate other aggregation functions. First, it sorts X using pSort, obtaining G in which records of the same key are inherently grouped together. Next, it scans through G, processes each record and computes an intermediate result V. To prevent the worker from inferring the size of each group, this step outputs not only valid aggregation values, but also dummy records. Finally, it uses pCompact to remove dummy records in V. Since these 3 steps are all privacy-preserving, it follows from Corollary 1 that pAggr is also privacy-preserving.

Performance analysis. pAggr is constructed from pSort and pCompact, thus it runs $O(n \log n)$ time. The data-oblivious alternative described in [6] 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 the sake of disposition, let us consider a simplified version of the join algorithm, although generalizing to other join algorithms is straight forward. 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 the algorithms considered so far, the output size of join can be larger than the input size.

Table 2: Example of pJoin for inputs $X_1 = \{\langle a, fde\rangle, \langle a, tol\rangle, \langle b, lxv\rangle, \langle b, xdj\rangle\}$ 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 of weights in each group.

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

Baseline solution. We consider the sort-merge join algorithm. It first sorts X_1 and X_2, then performs interleaved linear scans to find matching records. These sorting and matching steps may reveal the entire join graph.

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 4 shows the privacy-preserving join algorithm — pJoin — which is based on the data-oblivious algorithm proposed by Arasu *el al.* [7]. 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 sub-steps in [7] with a pSort, two linear-scan and two pCompact steps (line 2-7). However, it uses the data-oblivious expansion step without change (line 8-9). Because 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 of size $n = n_1 + n_2$, then privately sorts X, ensuring that for those records having the same key, tie is broken by placing X_1's records before X_2's. Next, it 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 pass, X_1 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, pJoin duplicates each record in X_1 and X_2 a number of times indicated by its associated weight. It directly uses the oblivious expansion algorithm presented in [7] for this step. Finally, it performs a linear scan to stitch records together and generate the final output Y. Table 2 gives a detailed example for pJoin.

Performance analysis. The time complexity of the first stage is $O(n \log n)$, and that 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 the complexity of the oblivious alternative [7]. Nevertheless, we show later in Section 5 that pJoin has lower running time because pSort and pCompact are more efficient than the data-oblivious sub-steps.

5. PERFORMANCE EVALUATION

We report the performance of the four algorithms described in the last section: pSort, pCompact, pAggr and pJoin. We generate input data following Yahoo! TeraSort benchmark [36]. Each record comprises 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. We assign 64MB of secure memory to the trusted unit[5] (i.e. $m = 2^{19}$), and vary the input size from 8GB to 64GB. Our implementations[6] use Crypto++ library[7] for cryptographic operations. Experiments are run on a DELL workstation equipped with a Intel i5-4570 3.2GHz CPU and a 500GB SATA disk. We repeat each experiment 10 times and report the average result.

5.1 Cost of Security

[5]We have run other sets of experiments with various secure memory capacities (e.g. 128MB and 256MB) and found that varying the secure memory within the small range does not affect the the algorithms' performance. On the other hand, much larger secure memory, say a few GB, will improve the running time. However, since we consider settings in which m is in orders of $\sqrt{n}$, we rule out this option.

[6]Our implementations are available on github at `https://github.com/dkhungme/PRAMOD`.

[7]`http://www.cryptopp.com`.

Table 3: Leakage of the baseline system, oblivious solution and the proposed PRAMOD for different 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		

Table 4: Overall running time (in seconds) of PRAMOD's algorithms in comparison with: (1) implementations in the baseline system with weaker security and (2) data-oblivious algorithms offering the similar level of pirvacy protection.

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×)

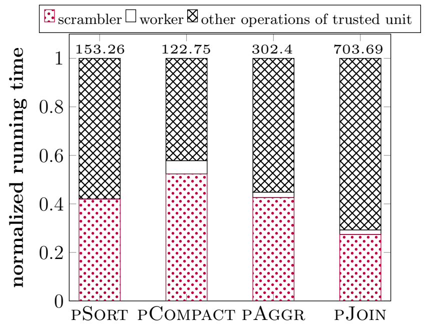


Figure 4: Normalized running time breakdowns for PRAMOD's algorithms. The overall cost consists of the time taken by the scrambler, plus the time required by the worker (if any). The rest is taken by pre-processing, post-processing steps and the main algorithm logic. The value on top of each bar indicates the running time of the corresponding algorithm (in minutes).

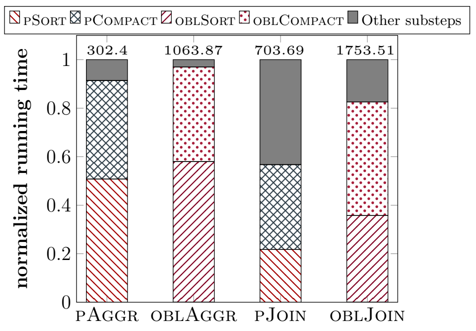


Figure 5: Normalized running time breakdowns for group aggregation and join algorithms. The running time consists of the time taken by sort and compaction steps, as well as by other sub-steps. The value on top of each bar indicates the running time of the corresponding algorithm (in minutes).

We first compare PRAMOD's algorithms with the alternatives in the baseline system. As noted in Section 4, the baseline implementations reveal more information about the input via access patterns than what are admitted by our security definition. We then compare them with data-oblivious algorithms: oblSort for sorting [24], oblCompact for compaction [22], oblAggr for group aggregation [6] and oblJoin for join [7]. For the sake of completeness, we summarize leakage of different approaches in Table 3.

Overhead. Table 4 quantifies the execution time for 32GB inputs (or $n = 2^{28}$ records[8]). It shows that PRAMOD incurs overheads between 2.43× to 4.99× over the baseline system. This cost of security is considerable. Nevertheless, it is still practical, considering the overheads of data-oblivious algorithms offering 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 steps and on the main algorithm logic. Figure 4 depicts this breakdown. It can be seen consistently across all algorithms that the cost of scrambling is significant. Particularly, the scrambler contributes 42%, 52.3%, 42.6% and 27.4% to the overall cost of pSort, pCompact, pAggr and pJoin, respectively. The time taken by the untrusted worker accounts for small proportion of the total running time, from 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 sorting and compaction algorithms, we also report cost break-down of the algorithms over the 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. Apart from sort and compaction, other sub-steps (e.g. oblivious expansion step in join) in the data-oblivious algorithms and ours have the similar running time. This explains why the contributions of sort and compaction 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).

Re-Encryptions and I/O complexity. Table 5 details the costs of cryptographic operations (which are CPU intensive) and of communication between the trusted unit and the storage (which is I/O intensive). Observe that PRAMOD's algorithms require $O(n)$ I/Os with a small constant 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

[8]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 I/O complexity required by PRAMOD's algorithms and relevant data-oblivious solutions on input of size n. p_1 and p_2 are constant parameters in the scrambler's configuration (see Section 3.1). In our experiments, $p_1 = p_2 = 2$. $s = n/m$ and d is the average degree of records in the join graph (we assume $d = 3$ in our experiments).

Algorithm	# Re-Encryption	I/O Complexity
pSort	$(p_1 + p_2 + 5) \cdot n$	$O(n)$
oblSort[24]	$(\sum_{i=1}^{\log s} i + \log s + 1) \cdot n$	$O(n \log^2 n)$
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[6]	$(\sum_{i=1}^{\log s} i + \log s + \log n + 3) \cdot n$	$O(n \log^2 n)$
pJoin	$(3p_1 + 3p_2 + 9 + d) \cdot n$	$O(dn)$
oblJoin[7]	$(\sum_{i=1}^{\log s} i + \log s + 2\log n + 5 + d) \cdot n$	$O((dn) \log(dn) + n \log^2 n)$

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. For PRAMOD algorithms, these numbers depend on their specific configurations, i.e. p_1, p_2. 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[9] . On the other hand, the numbers of re-encryptions of data-oblivious algorithms depend only on the size of the secure memory. We observe that given secure memory of size $m = c\sqrt{n}$ (where c is a small constant larger than one) and the same input, the data-oblivious algorithms perform a few times more re-encryptions than PRAMOD's privacy-preserving algorithms, which directly translates to considerable performance overheads.

5.2 Efficiency and Scalability

Figure 6 illustrates how the algorithms scale in running time with larger input sizes. PRAMOD's algorithms outperform the data-oblivious alternatives for all input sizes. More specifically, pSort is faster than oblSort [24] by $2.6 - 4.4\times$ (Figure 6a). Similarly, pCompact is faster than oblCompact [22] by $3 - 3.5\times$(Figure 6b), pAggr is faster than oblAggr [6] by $2.7 - 3.8\times$ (Figure 6c), and pJoin is faster than oblJoin [7] by $2 - 2.6\times$ (Figure 6d). The speedup is due to the fewer numbers of re-encryptions and I/O accesses, both are expensive operations.

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 data-oblivious alternatives.

Discussion. PRAMOD is currently running on a single machine, but we stress that porting it to a distributed environment is straight forward for two reasons. First, the scrambler processes data in blocks independently of each other, which lends itself naturally to distributed setting. Consequently, distributing the scrambler's workload to multiple nodes could result in substantial speed-up because the scrambling process is CPU intensive. In fact, the current implementations are multi-threaded (4 threads), and in comparison with the single-thread version, we observed $1.8\times$ speed-up. Second, most external-memory algorithms are often designed to support parallelism, making it simple to port them to a distributed setting. On the other hand, data-oblivious algorithms are difficult to parallelize because of their complexity.

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

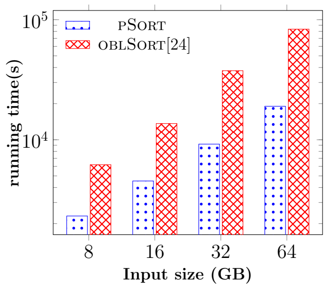
6. RELATED WORK

Secure Data Management using Trusted Hardware. Several systems have used trusted computing hardware such as IBM 4764 PCI-X [10] or Intel SGX [1] to enable secure data management. TrustedDB [8] presents a secure outsourced database prototype which leverages on IBM 4764 secure CPU (SCPU) for privacy-preserving SQL queries. Cipherbase [6] 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.

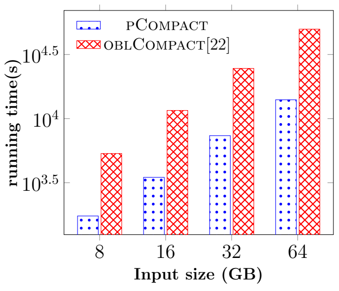
Recent systems [17, 34] adopt a similar approach to this paper's to support privacy-preserving computation. However, they focus on the MapReduce computation model, and specifically use scrambling to ensure security for the shuffling phase (which is essentially a sorting algorithm). PRAMOD is a more general 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 [38, 12, 24] 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 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.

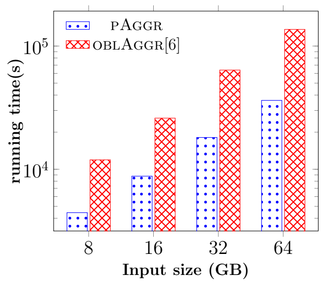
[10]`http://www-03.ibm.com/security/cryptocards/pcixcc/overview.shtml`



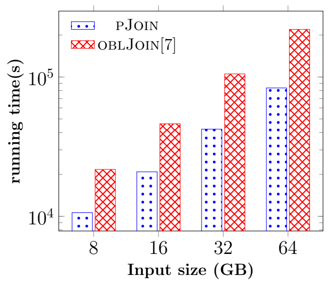
(a) Sorting



(b) Compaction



(c) Group-Aggregation



(d) Join

Figure 6: Performance comparison between our algorithms and the corresponding data-oblivious alternatives. Running time (in seconds) is reported in log-scale (y-axis) for different input sizes (x-axis).

Another line of works advocate designing data-oblivious algorithms. Goodrich *et al.* present several oblivious algorithms for sorting [24, 23], compaction and selection [22]. The authors also propose approaches to simulate ORAM using data-oblivious algorithms [24]. Other interesting data-oblivious algorithms have also been proposed for graph drawing [25], graph-related computations such as maximum flow, minimum spanning tree, single-source single-destination (SSSD) shortest path, and breadth-first search [11]. 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 showed that for many algorithms, prepending them with the scrambling step make the algorithms privacy preserving. Moreover, PRAMOD achieves security for other complex algorithms by decomposing them into smaller privacy-preserving sub-steps. We demonstrated four algorithms: pSort for sorting, pCompact for compaction, pAggr for group aggregation and pJoin for join, all of which are not only privacy-preserving but also optimal. We showed experimentally that the algorithms are efficient and scalable, outperforming the corresponding data-oblivious algorithms offering a similar level of privacy protection.

8. REFERENCES

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

[2] Tpc benchmark ds. `http://www.tpc.org/tpcds/spec/tpcds_1.1.0.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. In *CIDR, 2013.*

[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. In *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. In *ASIACCS, 2013.*

[12] D. Boneh, D. Mazieres, and R. A. Popa. Remote oblivious storage: Making oblivious ram practical. *MIT-CSAIL-TR-2011-018*, 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*, 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] E. K. Donald. The art of computer programming. *Sorting and searching*, 3:426–458, 1999.

[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. 1996.*

[22] M. T. Goodrich. Data-oblivious external-memory algorithms for the compaction, selection, and sorting of outsourced data. In *SPAA 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] V. Gupta, G. Miklau, and N. Polyzotis. Private database synthesis for outsourced system evaluation. In *AMW*, 2011.

[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. 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 *ACM 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 Security and Privacy, 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, C. Redfield, N. Zeldovich, and H. Balakrishnan. Cryptdb: protecting confidentiality with encrypted query processing. In *SOSP*, 2011.

[40] 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.*

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

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

[43] H. Takabi, J. Joshi, and G.-J. Ahn. Security and privacy challenges in cloud computing environments. In *IEEE Security and Privacy, 2010.*

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

[45] J. Vaidya and C. Clifton. Privacy preserving association rule mining in vertically partitioned data. In *ACM SIGKDD*, 2002.

[46] L. Zou, L. Chen, and M. T. Özsu. K-automorphism: A general framework for privacy preserving network publication. In *VLDB*, 2009.

A. THE MELBOURNE 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 as input 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. The algorithm then 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 [9] and the result from [35], the probability that the algorithm restarts is:

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