Privacy-preserving computation with trusted computing via Scramble-then-Compute

Hung Dang, Tien Tuan Anh Dinh, Ee-Chien Chang
Beng Chin Ooi, Prateek Saxena, Shruti Tople

School of Computing, National University of Singapore

Abstract. We consider privacy-preserving computation of big data using trusted computing primitives with limited private memory. Simply ensuring that the data remains encrypted outside the trusted computing environment is insufficient to preserve data privacy, because data movement observed during computation could leak information. Designing algorithms that thwart such leakage is challenging. Many known privacy-preserving algorithms are complex and induce large trusted code bases that are unwieldy to vet and verify. In this paper, we make a key observation that many basic algorithms (e.g. sorting) can be made privacy-preserving by adding a step that securely scrambles the data before feeding it to the original algorithms. We call this approach *Scramble-then-Compute* (StC), and give a sufficient condition whereby existing external memory algorithms can be made privacy-preserving via StC. This approach facilitates code-reuse, and its simplicity contributes to a smaller trusted code base. It is also general, allowing algorithm designers to leverage on the rich set of known algorithms for better performance. Our experiments show that StC could offer up to 4.1× speedups over known, application-specific alternatives.

1 Introduction

Big data is the main driving force behind online data storage model offered by incumbent cloud service providers. While these services are cost-effective and scalable, security in terms of data privacy remains a concern, because user data is being handled by untrusted parties. Even when the providers are trusted, other factors like multi-tenancy, complexity of software stacks, and distributed computing models continue to enlarge the attack surface [12, 14]. In addition, there is a tight constraint on the performance overhead since most computations on the data, especially data analytics tasks, consume vast numbers of CPU cycles which are directly billable [13, 5].

The first step towards securing the data is to protect it using encryption. Semantically secure encryption schemes ensure high level of security, but only protect data at rest [25]. Fully homomorphic encryption schemes allow for computations over encrypted data, but suffer from prohibitive overheads [16, 10]. Partially homomorphic encryption schemes [31, 15] are more practical, but limited in the range of supported operations [33, 36].

A line of recent works have advocated an approach of combining encryption with trusted computing primitives which offer a confidentiality and integrity protected execution environment [6, 34, 3]. This trusted environment can be provisioned by either hardware (e.g. Intel SGX processors [1]) or hardware-software combination [26, 27]. Data is then stored in untrusted external memory/storage and protected by a semantically secure encryption. Data is only decrypted and processed in the trusted execution environment, and the outputs are encrypted before being written back to the storage. The trusted environment has a limit on the amount of data it can process at any time – capped by the size of the protected physical memory allocated to a process. This means a communication channel between the trusted and the untrusted environments is necessary to complete the computation. Unfortunately, such a channel could leak information about the data [35, 12, 14]. For instance, by observing I/O access patterns during merge sort, an attacker can infer the order of the original input, i.e. the ranks of input elements. This leakage could be eliminated by using generic oblivious-RAM (ORAM), but with high performance overheads. There exists also application-specific algorithms whose access patterns are data-oblivious, thus eliminating the leakage. However, designing these algorithms are challenging, and existing constructions are complex which indirectly leads to larger trusted code bases (TCB) that are difficult to vet and verify.

Our goal is to design algorithms which are privacy preserving and practical while keeping the TCB lean. To this end, we make a key observation that, for a large class of algorithms (e.g. sorting), randomly permuting (or scrambling) the input before feeding it to the original algorithms is sufficient to prevent leakage from access patterns. For example, consider merge sort algorithm in which the original input is first randomly permuted. During the execution, the adversary observing access patterns will, at best, be able to infer only sensitive information on the scrambled input. If the scrambling is done securely, such information cannot be linked back to that of the original input. Based on this observation, we propose an approach for designing privacy-preserving algorithms, called *Scramble-then-Compute* (StC), which essentially scrambles the input before executing the original algorithm on the scrambled data. This approach not only is applicable to a large number of algorithms, but also incurs only an additive overhead factor (as opposed to a multiplicative factor when using ORAM). Its generality facilitates code reuse, i.e. it allows us to choose among the rich set of known algorithms the most optimized ones thereby reducing the performance overhead. Furthermore, the scrambling step in StC is easily distributed, enabling the algorithms constructed under StC to scale. It is also worth mentioning that the simplicity of our solution promises an ease of implementation.

We note that not all algorithms can be made privacy-preserving by scrambling the input beforehand. Hence, we give a sufficient condition for algorithms derived by StC to be privacy-preserving (Section 3). Many algorithms involving data movement such as merge-sort, quicksort or compaction inherently satisfy

the condition[1]. We demonstrate StC by describing privacy-preserving implementations of five popular algorithms: *sort, compaction, select, group aggregation* and *join* (Section 4). The first three algorithms can be made privacy-preserving by directly applying StC, and the other two by stitching together privacy-preserving sub-steps. We benchmark their performance and and compare them to baseline implementations that are not privacy-preserving. StC offers a stronger privacy protection at a cost of 3.5× overhead on average. We further compare them to state-of-the-art data-oblivious alternatives with similar levels of security [20, 18, 3, 4]. The results show that StC algorithms can achieve speedups as high as 4.1×. The improvement on performance is probably gained from the extensive known results on external memory algorithms. Furthermore, these algorithms can be easily parallelized, thus enabling privacy-preserving computation at scale. Many data-oblivious algorithms, on the other hand, are difficult to parallelize. In summary, we make the following contributions:

1. We define a security model for privacy-preserving algorithms. The definition implies data confidentiality and privacy even when the adversary can observe I/O access patterns.
2. We propose StC – an approach for implementing privacy-preserving algorithms. We give a condition on algorithms whereby scrambling the input beforehand is sufficient to preserve privacy (Theorem 1 & 2). Multiple privacy-preserving sub-steps can be stitched together to realize more complex algorithms. StC's simplicity and generality help reduce the performance overhead and keep the TCB lean.
3. We demonstrate the utility of StC by applying it to five algorithms, all of which achieve asymptotically optimal runtime (Section 4). In particular, our privacy-preserving compaction runs in $O(n)$, and sorting in $O(n \log n)$ using $O(\sqrt{n})$ trusted memory.
4. We conducted extensive experiments to evaluate the privacy-preserving algorithms constructed under StC. The results indicate relatively low overheads over the baseline system which is less secure, and running time speedup of up to 4.1× over data-oblivious alternatives with a similar level of privacy protection. The results also show that StC fits well in distributed settings, allowing for further speedups of up to 7× when running on eight nodes.

The rest of the paper is structured as follows. The next section defines the problem and related challenges. Section 3 presents StC and the rationale behind the approach. Section 4 demonstrates its utility. The experimental evaluation is reported in Section 5. Related work is discussed in Section 6 before we conclude.

[1] There is a subtle issue with non-unique elements. We shall discuss techniques to address it in Section 3.3

2 Problem Definition

In this section, we discuss the problem and challenges of enabling privacy-preserving computation using trusted computing primitives. We also give formal definitions of privacy-preserving algorithms.

Throughout this section, we shall use the following running example to illustrate the problem and its related concepts. Let us consider a user outsourcing her data of integer-value records to the cloud in encrypted form. She then wishes to sort the data, perhaps as a pre-processing step for other tasks such as ranking or de-duplication. Sorting directly over encrypted data is possible, but it is impractical [2]. Instead, the user relies on a trusted unit which sorts and re-encrypts the records in its private memory. Since the private memory is limited in size, a k-way external memory merge sort algorithm is employed. Figure 1 depicts a simple example of three-way merge sort in which the private memory is limited to holding only three records at a time. The input consists of nine records, and sorting involves one merging step.

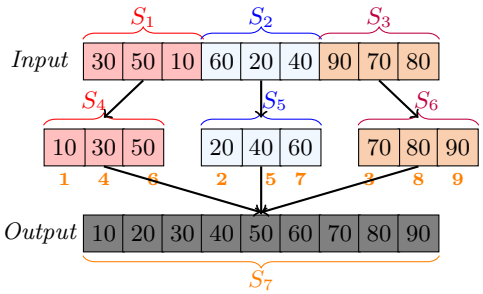


Fig. 1: An example of three-way external merge sort on encrypted records. The subscripts denote the order in which the record is read into the trusted unit during the merging step.

Fig. 2: The system model consists of a trusted unit which can process a limited number of records at a time. The storage and worker are untrusted. Only the trusted unit can see the content of the encrypted records (denoted by blue squares).

2.1 Computation and Adversary Model

Computation model. Let $X = \langle x_1, x_2, \ldots, x_n \rangle$ be the input data of n equal-sized key-value records. Let $key(x)$ and $val(x)$ denote the key and value component of a record x, respectively. An algorithm $\mathcal{P}$, given the input X, computes an output sequence $Y = \mathcal{P}(X) = \langle y_1, y_2, .., y_{n'} \rangle$ of n' key-value records. Unlike input records, the output records need not be of the same size.

We focus on a class of algorithms which are *permutation-invariant*. An algorithm $\mathcal{P}$ is permutation-invariant if $\mathcal{P}(X) = \mathcal{P}(\pi(X))$ for any input X and

permutation $\pi(\cdot)$ over the records in X. In our running example, $\mathcal{P}$ is the three-way merge sort algorithm, $key(x)$ is the value to be sorted, and Y is the sorted output.

The computations are to be carried out by a *trusted unit* and a *worker* with a *storage* (depicted in Figure 2). The trusted unit holds persistent secret data, e.g. secret key used in cryptographic operations that is established prior to the algorithm's execution. It has a limited memory to hold m records. Data is persisted in the long term storage component whose communication with the trusted unit is mediated by the worker. The worker can also carry out computations.

Threat Model. We consider an honest-but-curious adversary who has complete control over the storage and worker. Such adversary can be an insider who has full access to the cloud infrastructure via misuses of privilege, or an attacker gaining access by exploiting vulnerabilities in the software stack. The adversary is able to see the input, output, and access sequences made by the trusted unit. This is a realistic threat model, given recent security breaches (e.g. NSA and Target) being attributed to insider threats. A malicious adversary (aka active adversary), in contrast, can modify data in the storage and deviate the worker from its execution path. Such adversaries are considered by other works [14, 34]. Although our security model considers only honest-but-curious adversary, we believe that our approach remains applicable against malicious adversary. For instance, by incorporating additional integrity check, we can detect malicious tampering. We leave it as an avenue for future work.

We assume that the trusted unit constituting the system's TCB is sufficiently protected, and thus the adversary is unable to observe its states. For TCB based on hardware-software combination, we assume that the software part is free of vulnerabilities and malwares. Furthermore, we assume that there is no side-channel leakage (e.g. power analysis) from the trusted unit. Physical attacks which compromise the trusted unit's protection mechanisms, such as cold-boot attacks that subvert the CPU's hardware protections, are beyond scope. Finally, we assume that some data (e.g. secret keys) can be delivered securely to the trusted unit before the algorithms' execution.

Baseline system. For security and performance analysis, we compare the algorithm in question with an external memory algorithm (which is not necessary secure) which executes under a baseline system. Under this system, records in the storage are protected by a semantically secure encryption scheme with a secret key stored in the trusted unit. Moreover, all records written back to the storage are re-encrypted. Hence, even if the adversary can observe the input and output of the trusted unit, it is still unable to infer content of the records. This baseline system serves as a fair point for comparison, as the overhead incurred by encryption and decryption is arguably minimal. However, as we shall see below, the baseline could leak important information.

Leakage of the baseline system. The baseline system fails to ensure data privacy. In our running example (Figure 1), the encrypted input is divided into

three blocks, each with three records. The trusted unit executes the algorithm in two phases. First, it independently sorts each block 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. The adversary observes in the merging phase that the trusted unit first takes one record from each sorted block, writes one record out, then takes in another record from the first block. From this, he knows that the smallest record is from S_1. Such inference may eventually reveal the distribution of input data. For algorithms taking data from different anonymous sources, this can potentially expose their identities.

Performance Requirements. An important performance requirement is to keep runtime overhead low. To prevent leakage, one could employ oblivious RAM [35] directly on the storage backend, but this approach incurs $\Omega(\log n)$ overhead per access, making it impractical for big data processing. Another option is to use application-specific data-oblivious algorithms such as oblivious sort [20], which are both complex and limited in scope. We note that providing privacy while being able to leverage on state-of-the-art external memory algorithms for improved performance is certainly of great interest.

For security, it is important to keep the overall TCB small. Even though we assume the TCB to be free of vulnerabilities, we remark that it is desirable to keep the TCB small since verifying a large code base is unwieldy. Our proposed approach – StC – essentially adds a *scrambler* to existing external memory algorithms in order to achieve security (Section 3). Its simplicity leads to small TCB and low performance overhead. Furthermore, its ability to support parallelism enables privacy-preserving computation at scale.

2.2 Security Definition

First, let us formalize the information that the adversary can learn by observing the computation. Let $Q^m_{\mathcal{P}}(X) = \langle q_1, q_2, \ldots, q_z \rangle$ be the access (read/write or I/O) sequence the adversary observes during the execution of $\mathcal{P}$ on X, where m is the maximum number of records that the trusted unit can hold at any time. Hereafter, unless stated otherwise, we assume $m > \sqrt{n}$ where n is the number of input records, and omit the superscript m in the notation. Each q_i is an I/O request made by the trusted unit to the worker. It is a 3-value tuple $\langle op,\ addr,\ info \rangle$ where $op \in \{\mathtt{r}, \mathtt{w}\}$ is the type of the request ("read" or "write"), *addr* is the address accessed by *op*, and *info* is metadata ($\perp$ if not applicable) revealed to the worker (the record content is not included in the request because it is encrypted). The metadata is useful when the trusted unit wishes to offload parts of the computation to the worker. For example, if an algorithm sorts records by non-secret indices, the indices can be revealed to the worker via *info*, allowing the latter to complete the sorting. In our running example (Figure 1), the observed sequence of read requests is as follows (the sequence containing write requests is similar):

$$Q_{\mathcal{P}}(X)^{read} = \begin{pmatrix} \langle r, S_1, \perp\rangle,\ \langle r, S_2, \perp\rangle, \langle r, S_3, \perp\rangle,\ \langle r, S_1, \perp\rangle, \\ \cdots \\ \langle r, S_2+2, \perp\rangle, \langle r, S_3+1, \perp\rangle, \langle r, S_3+2, \perp\rangle \end{pmatrix}$$

where $S_i + j$ denotes the address of the j^{th} record in block S_i.

Our security definition requires that $Q_{\mathcal{P}}(X)$ leaks no information on X (except for its size, i.e. the number of records).

Definition 1 (Privacy-Preserving Algorithm). *An algorithm $\mathcal{P}$ is privacy-preserving if for any two datasets X_1, X_2 with the same number of records, $Q_{\mathcal{P}}(X_1)$ is computationally indistinguishable from $Q_{\mathcal{P}}(X_2)$.*

Note that the asymptotic notion of indistinguishability implicitly requires a security parameter κ. In our proposed approach, κ determines the running time (e.g. number of rounds in the scrambler), and the length of the key for cryptographic operations (e.g. the seed that generates pseudorandom permutations for the scrambler).

Relationship to data obliviousness. A related security notion is *data obliviousness* [18], in which $\mathcal{P}$ is data-oblivious if $Q_{\mathcal{P}}(X_1) = Q_{\mathcal{P}}(X_2)$ for any X_1 and X_2 having the same number of records. This definition is stronger than ours in the sense that it implies perfect zero leakage via access patterns. However, in practice, since encryption is involved, we argue that the security of data-oblivious algorithms essentially still relies on indistinguishability.

Permissible leakage. Some applications permit certain leakage of information. For example, consider the problem of grouping records by their keys, the number of unique keys may be considered non-sensitive, and revealing it is permissible. We formulate such leakage by a deterministic function Ψ, and call $\Psi(X)$ the *permissible leakage* on input X. In the group-by-key example above, $\Psi(X)$ is the number of unique keys in X. We say that an algorithm is Ψ-privacy-preserving if it leaks no information on X beyond $\Psi(X)$.

Definition 2 (Ψ-Privacy-Preserving). *An algorithm $\mathcal{P}$ is Ψ-privacy-preserving if for any two datasets X_1, X_2 with the same number of records and permissible leakage (i.e. $\Psi(X_1) = \Psi(X_2)$), $Q_{\mathcal{P}}(X_1)$ is computationally indistinguishable from $Q_{\mathcal{P}}(X_2)$.*

Clearly, when $\Psi(X)$ is a constant for any X, then a Ψ-privacy-preserving algorithm is also privacy-preserving.

3 Scramble Then Compute

In this section, we present StC – an approach for implementing privacy-preserving external memory algorithms. Given an algorithm $\mathcal{P}$ which is not necessarily privacy-preserving, StC derives $\mathcal{A}_{\mathcal{P}}$. We give conditions for $\mathcal{P}$ under which $\mathcal{A}_{\mathcal{P}}$ is privacy-preserving.

StC follows the computation model described earlier in Section 2. For those algorithms that are not inherently privacy-preserving, StC employs a component called *scrambler* which randomly permutes the input without revealing any information of the permutation during its execution. Some algorithms, such as sort, compaction and selection, can be made privacy-preserving immediately via the this approach. Others such as join and group aggregation are made privacy-preserving by exploiting the composition property, i.e. restructuring the original algorithms to be composed of only privacy-preserving sub-steps.

3.1 The Scrambler

The scrambler $\mathbb{S}$ takes input X and outputs a permuted sequence $\widetilde{X} = \pi(X)$ where π is a permutation randomly chosen by $\mathbb{S}$. While there are several variants on the formal security definition of $\mathbb{S}$, we adopt the one based on indistinguishability since it is sufficient in our construction. We say $\mathbb{S}$ is *secure* if it is Ψ-privacy-preserving with respect to a deterministic function Ψ that outputs the sorted sequence of the input X.

We implement the scrambler $\mathbb{S}$ using the Melbourne shuffle algorithm [29]. The algorithm takes input X and a permutation π, and outputs the permuted sequence $\widetilde{X} = \pi(X)$ in a data-oblivious manner. In other words, the access sequence is *the same* for all X of the same size. In fact, the Melbourne shuffle achieves a stronger security guarantee than our requirement on the secure scrambler. It assures that if π is chosen randomly and uniformly, the adversary is unable to distinguish the inputs in the information theoretic sense. Our definition only requires computational indistinguishability. Another construction of $\mathbb{S}$ is Chaum's mix-network [11], which achieves statistical indistinguishability, but the current known provable bounds require large trusted memory [23]. We discuss the implementation of $\mathbb{S}$ and its security parameter κ in Appendix A.

3.2 Deriving Privacy-Preserving Solutions

Recall that $\mathcal{P}$ is permutation-invariant if it always outputs the same result on different input permutations. Examples include sort and group-by-key algorithms. However, hash table lookup or binary search algorithm, which assumes certain structure or order of the input, is not permutation-invariant.

Scramble-then-compute. Given a permutation-invariant algorithm $\mathcal{P}$, StC derives an algorithm $\mathcal{A}_{\mathcal{P}}$ by first scrambling its input X, then forwarding the scrambled data to $\mathcal{P}$. Specifically, $\mathcal{A}_{\mathcal{P}}(X) = \mathcal{P}(\mathbb{S}(X))$. Clearly, by the definition of permutation-invariant, $\mathcal{A}_{\mathcal{P}}$ preserves the correctness of the original $\mathcal{P}$, in the sense that $\mathcal{A}_{\mathcal{P}}(X) = \mathcal{P}(X)$ for any X. For abbreviation, let us call $\mathcal{A}_{\mathcal{P}}$ the combined algorithm.

Before stating our theorem, let us first introduce the following two definitions.

Definition 3 (Tagging Algorithm). *A deterministic algorithm $\mathcal{T}$ operating on X is a* tagging algorithm *if $\mathcal{T}(X)$ is a permuted sequence of $\langle 1, 2, \ldots, n\rangle$, where n is the number of records in X.*

For example, a tagging algorithm can, on input of a sequence of integers $\langle 50, 3, 1, 10\rangle$, output $\langle 4, 2, 1, 3\rangle$ representing the record ranks in the input.

Definition 4 (Imitator). *Given an algorithm $\mathcal{P}$, the pair of two algorithms $\langle \mathcal{P}^*, \mathcal{T}\rangle$ in which $\mathcal{T}$ is a tagging algorithm operating on X is an* imitator *of $\mathcal{P}$ if for any input X and permutation π, the access sequence $Q_{\mathcal{P}}(\pi(X)) = Q_{\mathcal{P}^*}(\pi(\mathcal{T}(X)))$.*

The algorithm $\mathcal{P}^*$ essentially incurs the same observable behaviour as $\mathcal{P}$ when operating on the "downgraded" input. We now give a sufficient condition for $\mathcal{A}_{\mathcal{P}}$ to be privacy-preserving.

Theorem 1. *Given a permutation-invariant algorithm $\mathcal{P}$, if there exists an imitator $\langle \mathcal{P}^*, \mathcal{T}\rangle$ of $\mathcal{P}$, then $\mathcal{A}_{\mathcal{P}}$ is privacy-preserving.*

The proof for this theorem is presented in Appendix B. The theorem provides an easy mean to determine whether an existing external memory algorithm $\mathcal{P}$ can be made privacy-preserving via StC: it suffices to define an imitator of $\mathcal{P}$. If $\mathcal{P}$ is a comparison-based algorithm, the tagging algorithm $\mathcal{T}$ can be the one that outputs the record ranks, and $\mathcal{P}^*$ be a similar comparison-based algorithm as $\mathcal{P}$ but applying the comparison on the ranks. If the records are unique, then $\langle \mathcal{P}^*, \mathcal{T}\rangle$ is an imitator of $\mathcal{P}$.

The above theorem does not consider the case of permissible leakage. Intuitively, when permissible leakage is acceptable, the imitator should has access to such leakage. Hence, given $\mathcal{P}$ and a permissible leakage Ψ, we say that $\langle \mathcal{P}^*, \mathcal{T}\rangle$ is an Ψ-imitator of $\mathcal{P}$ if the access sequence $Q_{\mathcal{P}}(\pi(X)) = Q_{\mathcal{P}^*}(\pi(\mathcal{T}(X)), \Psi(X))$ for any X and permutation π.

Theorem 2. *Given a permutation-invariant algorithm $\mathcal{P}$ and a permissible leakage Ψ, if there exists an Ψ-imitator of $\mathcal{P}$, then $\mathcal{A}_{\mathcal{P}}$ is Ψ-privacy-preserving.*

The proof for this theorem is similar to that of Theorem 1 and is omitted.

Composition. If algorithms $\mathcal{P}_1$ and $\mathcal{P}_2$ are privacy-preserving, using the hybrid argument, their composition, i.e. executing one after another, is also privacy-preserving. However, only a polynomial number of compositions (with respect to κ) are allowed.

We demonstrate the composition property using two examples of group-aggregation and join algorithms in Section 4.4 and 4.5. For now, we note that the condition on the number of sub-steps does not imply usage restriction, because practical algorithms do not contain an exponentially large number of sub-steps.

3.3 Discussion

The condition on being permutation-invariant is strict, as it requires the output of the algorithm on the scrambled input to be *exactly* the same as when running on the original input. Sort algorithms that take duplicate input values do not meet this condition. 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 denote the original positions in the input, it may be the case that $\mathcal{P}(X) = \langle 0_0, 0_3, 0_1, 0_4, 0_2, 0_5 \rangle$ while $\mathcal{A}_{\mathcal{P}}(X) = \langle 0_0, 0_2, 0_1, 0_5, 0_3, 0_4 \rangle$ for a certain permutation generated by the scrambler. This problem can be resolved by adding metadata (e.g. address of the record) to the keys so that the input contains no duplicate. Without loss of generality, some algorithms which are not permutation-invariant can be made so by introducing a pre-processing step which appends metadata to the input, then reversing the effect via a corresponding post-processing step.

We stress the simplicity StC offers in deriving privacy-preserving algorithms from existing algorithms. One immediate benefit is code reuse. For example, there are extensive studies on sorting algorithms, each catered for a specific system configuration and application. With StC, especially with its ability to support parallelism, we can easily adopt the most suitable algorithm with the most well-tuned parameters. Another benefit is the small TCB, as we can choose an algorithm with small codebase. This is as opposed to implementing convoluted algorithms like existing data-oblivious ones. Furthermore, our approach offers an arguably simpler way of implementing data-oblivious algorithms; the composition property allows us to replace the complex data-oblivious sub-steps with more efficient StC alternatives. We demonstrate this advantage in Section 4.5.

Finally, although the algorithms considered so far are deterministic, StC also generalizes to probabilistic instances such as quick sort. Specifically, they can be modified to take the random choices as additional input, making them deterministic and to which our theorems can be applied.

4 Privacy-Preserving Algorithms

In this section, we demonstrate the utility of our approach by showing the implementation of five algorithms in StC: sort, compaction, selection, group aggregation and join. The first three are realized directly through StC, and the other two by stitching together privacy-preserving sub-steps. We provide performance analysis for each algorithm and compare it with the baseline implementation as well as the data-oblivious alternative. Our algorithms offer better privacy protection than the baseline implementations, and similar to the data-oblivious alternatives but with better performance. In Appendix C, we also discuss how StC can be applied on basic operations in Spark[2]. to make them privacy-preserving.

4.1 Sort

The algorithm sorts the input according to a certain order of the record keys. We consider external merge sort algorithm [24], in which the input is divided into $s = n/m$ blocks ($s < m$) and the sorted blocks are combined in one merging step using s-way merge. This algorithm has optimal I/O performance, but leaks the input order when implemented in the baseline system. In StC, we first adds a pre-processing step which appends the address of each record to its key,

[2] Apache Spark. `http://spark.apache.org/docs/latest/programming-guide.html`

Table 1: Comparison of time complexity of different algorithms. For join algorithm, l is the size of the result

Algorithm	*Baseline*	StC	*Oblivious Algorithms*
Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log^2 n)$
Compaction	$O(n)$	$O(n)$	$O(n \log n)$
Selection	$O(n)$	$O(n)$	$O(n)$
Group aggregation	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
Join	$O(n_1 \log n_1 + n_2 \log n_2)$	$O(n_1 \log n_1 + n_2 \log n_2 + l \log l)$	$O(n_1 \log n_1 + n_2 \log n_2 + l \log l)$

i.e. $key(x'_i) = key(x_i)||i$. The result is then forwarded to the scrambler, whose output is used as the input to the original algorithm (the comparison function breaks ties using the address attached to the key). Finally, the post-processing step scans through the output and removes the address information.

This derived algorithm, called pSort, runs in $O(n \log n)$ time. The pre-processing and post-processing steps make the original algorithm permutation-invariant. By Theorem 1, pSort is privacy-preserving. To the best of our knowledge, the most efficient data-oblivious alternatives run in $O(n \log^2 n)$ [20, 18][3].

4.2 Compaction

The algorithm removes $(n-n')$ *marked* records from the input of n records, while preserving the original order of the remaining n' records. The baseline algorithm sequentially reads the input records into the trusted unit and writes back those unmarked records (re-encrypted). This solution is efficient but reveals the distribution of the marked records. In StC, the algorithm pCompact consists of four steps. First, the trusted unit initializes two counters, $C_1 = 0$, $C_2 = n$. While scanning through X, it labels each record with C_1 or C_2 if the record is unmarked (to be retained) or marked (to be removed), respectively. C_1 is incremented while C_2 is decremented after each labeling. The next two steps involve running the labeled input through the scrambler and the baseline algorithm. Finally, the trusted unit reveals the labels to the worker so that the latter can move records to their final positions.

pCompact runs in $O(n)$, while the data-oblivious alternative [18] runs in $O(n \log n)$. The pre-processing and post-processing steps make the baseline algorithm permutation-invariant. The tagging algorithm $\mathcal{T}$ works as follows. First, if x_i is unmarked, then $t_i \leq n'$; otherwise, $t_i > n'$. Second, for any $i < j$, $t_i < t_j$ if both x_i and x_j are unmarked; or $t_i > t_j$ if both x_i and x_j are marked. Here, the permissible leakage $\Psi(X)$ is the number of marked records in X (the data-oblivious algorithm [18] also reveals this information). By Theorem 2, pCompact is Ψ-privacy-preserving.

[3] The randomized version [18] runs in $O(n \log n)$ time with a large constant factor.

4.3 Selection

The algorithm outputs the k^{th} smallest element of the input according to a certain order of the record keys. A straightforward algorithm is to first sort the input data in ascending order and then output the k^{th} record, but its complexity is $O(n \log n)$. Instead, we consider the *median of medians algorithm* [8] which has $O(n)$ runtime complexity even in the worst-case. The baseline implementation of this algorithm, however, partially reveals the distribution of the input records. The algorithm in StC, called pSelect, is the same as pSort, except that merge sort is replaced by the median of medians algorithm. Unlike pSort, pSelect outputs one record instead of a sorted sequence of n records.

pSelect runs in $O(n)$ time, having the same complexity as the existing data-oblivious alternative [18]. We show in the next section, however, that in practice pSelect outperforms its data-oblivious counterpart by a few times. Similar to pSort, the baseline algorithm is permutation-invariant because of the pre-processing and post-processing steps. pSelect is privacy-preserving according to Theorem 1.

4.4 Group aggregation

The algorithm first groups records based on their keys, then applies an aggregation function, such as summing or averaging, over the group members. We consider a baseline algorithm which first sorts the input, then scans the sorted records, accumulates the values and writes out a output record immediately after passing the last record of each group. Because of this last step, the overall execution reveals the size of each group even when a privacy-preserving sorting algorithm is used.

It can be shown that the baseline algorithm does not satisfy the condition in Theorem 1. Thus, we design a new privacy-preserving group aggregation algorithm, called pAggr, and exploit the composition property to derive its security. First, it sorts X using pSort, obtaining G in which records of the same key are next to each other. Second, it scans through G to compute the aggregate. During the process, it outputs one record for every record it encounters in G. Some of these records are real output records, while other are dummy and therefore marked so that they can be removed later. Finally, it uses pCompact to remove the dummy records. Because these 3 steps are privacy-preserving, so is pAggr. The algorithm runs in $O(n \log n)$ — the same complexity as that of the data-oblivious alternative [3].

4.5 Join

The algorithm performs the inner join on two datasets X_1 and X_2. We consider the sort-merge join algorithm (generalizing to other join algorithms is straightforward). It first sorts X_1 and X_2, then performs interleaved linear scans on two sorted sequences to pair matching records. Implemented in the baseline system, the sorting and matching steps reveal the entire join graph.

Similar to the group aggregation algorithm, the baseline join algorithm cannot be transformed using StC. We design a new privacy-preserving algorithm, pJoin, based on the data-oblivious version proposed by Arasu *et al.* [4]. The data-oblivious algorithm consists of two stages: the first stage computes the degree of each record in the join graph, and the second duplicates each record a number of times as indicated by its degree. We refer readers to [4] for more details. pJoin follows the same workflow, but it implements the first stage using pSort and pCompact while reimplementing the data-oblivious expansion step without change for the second stage.

pJoin runs with the same complexity as the data-oblivious algorithm does, i.e. $O(n \log n + l \log l)$ where l is the number of output records. Nevertheless, we show later in Section 5 that pJoin has a lower running time in practice, because pSort and pCompact are more efficient than the corresponding data-oblivious steps. Each and every step in pJoin is privacy-preserving, so is pJoin.

5 Performance Evaluation

We evaluate StC by benchmarking the five algorithms discussed in the last section. We first quantify the cost of security that StC incurs, by comparing the running time of our algorithms with those implemented in the baseline system. In addition, we compare this cost with that of the state-of-the-art data-oblivious alternatives: oblSort for sorting [20], oblCompact for compaction [18], oblSelect for selection [18], oblAggr for group aggregation [3] and oblJoin for join [4]. Next, we evaluate our approach's scalability by measuring the algorithm performance when running on a network of multiple nodes.

We generate the input data using the Yahoo! TeraSort benchmark [30]: each record comprises a 10-byte key and a 90-byte value. We encrypted each record with AES-GCM using a 256-bit key, generating a 132-byte ciphertext. We vary the size trusted memory from 64MB to 256MB (i.e. $m = 2^{19}$ to $m = 2^{21}$), and the input size from 8GB to 64GB. Our implementations[4] use Crypto++ library for cryptographic operations. For the distributed implementations, we use HDFS as the backend storage and Zookeeper to synchronize the processes. We run our experiments on an eight-node cluster of commodity servers, each node has an Intel Xeon E5-2603 CPU, 8GB of RAM, two 500GB hard drives and two 1GB Ethernet cards. We repeat each experiment 10 times and report the average results for 64MB trusted memory (other trusted memory sizes have similar performance).

5.1 Cost of Security

Table 2 compares the running time for various algorithms with 32GB inputs (or $n = 2^{28}$ records) on one node. While StC algorithms can run on multiple nodes, we are not aware of any distributed versions of the five data-oblivious algorithms under consideration. Thus, to make the comparison fair, we ran StC algorithms

[4] Available at `https://github.com/dkhungme/PRAMOD`.

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

Algorithm	*Baseline*	StC	*Oblivious Algorithms*
Sorting	7961	14330 (1.79×)	59628 (7.49×)
Compaction	1678	8253 (4.91×)	25012 (14.89×)
Select	2758	9451 (3.42×)	29365 (16.65×)
Group-Aggregation	10593	24578 (2.32×)	63477 (5.99×)
Join	12400	59610 (4.81×)	105235 (8.49×)

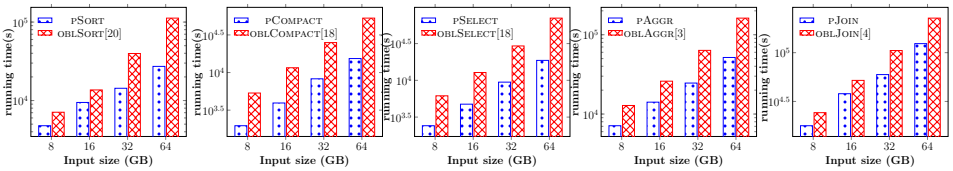
Table 3: Normalized running time breakdowns for StC's algorithms.

Algorithm	*Scrambler*	*Worker*	*Trusted Units*
pSort	41.8%	0.6%	57.6%
pCompact	52.4%	5.4%	42.2%
pSelect	64.1%	3.4%	32.5%
pAggr	42.6%	2.2%	55.2%
pJoin	27.4%	1.9%	70.7%

on a single node. It can be seen that StC algorithms incur overheads between 1.79× to 4.91× over the baseline system. To better understand the factors contributing 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. Table 3 lists the breakdown, showing consistently across all algorithms that the cost of scrambling is significant: from 27.4% (pJoin) to 64.1% (pSelect). The time taken by the untrusted worker accounts for small portions of the total running time, from 0.6% (pSort) to 5.4% (pCompact). This is because the worker does not perform cryptographic operations which are computationally expensive.

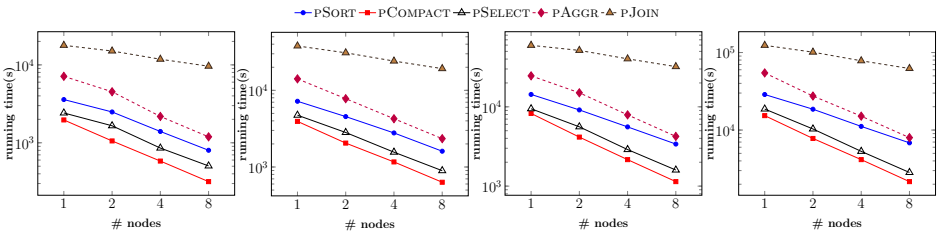
5.2 Comparison with data-oblivious algorithms

The overheads of data-oblivious algorithms are between 5.99× to 16.65× in comparison to the baseline system. Thus, we remark that StC algorithms incur relatively low overhead and therefore is practical. Figure 3 further illustrates that compared to their data-oblivious alternatives, they are consistently more efficient across all input sizes while offering similar privacy protection. More specifically, the privacy-preserving sorting algorithm under StC is up to 4.1× faster than the data oblivious one, compaction is up to 3.4×, selection is up to 3.8×, group aggregation is up to 3.1×, and join is 1.8×. We note that the speedup for join is smaller than for the others because the data-oblivious expansion algorithm,



(a) pSort (b) pCompact (c) pSelect (d) pAggr (e) pJoin

Fig. 3: Performance comparison between our algorithms and the corresponding data-oblivious alternatives. Running time (s) is shown in log-scale.



(a) 8GB (b) 16GB (c) 32GB (d) 64GB

Fig. 4: StCalgorithms performance on multiple nodes with different input sizes. Running time (s) is shown in log-scale.

which pJoin inherits directly from [4], contributes the most to the total running time. It is also worth noting that the speedup becomes more evident with larger inputs: from $1.3 - 2.7\times$ for 8GB datasets to $1.8 - 4.1\times$ for 64GB datasets.

We refer readers to Appendix D for details on the number of cryptographic operations and I/O complexity required by these algorithms.

5.3 Scalability

Figure 4 reports the running time of StC algorithms on multiple nodes. It demonstrates that our proposed approach can leverage resources in distributed environment to achieve significant speedups. In particular, increasing the number of nodes from one to eight results in $4\times$ speedup for sort and up to $7\times$ for compaction, selection and aggregation. This is over an order-of-magnitude better than single-node data-oblivious algorithms. However, pJoin achieves only $2\times$ speedup, because we cannot parallelize the oblivious expansion algorithm. We note that the speedup comes from the distribution of both the scrambler and of the original algorithm itself. Although our current implementations may not be the most efficient, their simplicity and speedup gained when scaling out are compelling evidence of StC's advantages over existing data-oblivious algorithms.

6 Related Work

Secure Computation using Trusted Hardware. Several systems have used trusted computing hardware such as IBM 4765 PCIe Cryptographic Coprocessor (SCPUs)[5] or Intel SGX processors [1] to enable secure computations, especially focusing on query processing. TrustedDB [5] presents a secure outsourced database prototype which leverages on IBM 4765 SCPUs for privacy-preserving SQL queries. Cipherbase [3] 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 [14, 28] 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). StC is a more general solution which supports many other algorithms.

Secure Computation by Data-Oblivious Technique. Oblivious-RAM [17] enables secure and oblivious computation by hiding data read/write patterns during program execution. ORAM techniques [32, 9, 20] trust a CPU with limited internal memory, while storing user programs and data encrypted on the untrusted server. A non-oblivious algorithm can be made data-oblivious by adopting ORAM directly, incurring performance overhead of $\Omega(\log n)$ per each access. StC offers a similar level of security with $O(n)$ additive overhead.

Goodrich *et al.* propose several data-oblivious algorithms [20, 19, 18] which we used for benchmarking StC. The authors also presented approaches to simulate ORAM using data-oblivious algorithms [20]. Other interesting data-oblivious algorithms have also been proposed for graph drawing [21] and graph-related computations such as maximum flow, minimum spanning tree, single-source single-destination (SSSD) shortest path, or breadth-first search [7]. However, these algorithms are application-specific and less efficient than StC algorithms.

7 Conclusion

We have described StC, an approach for implementing practical privacy-preserving algorithms using trusted computing with limited secure memory. We showed that many algorithms can be made privacy-preserving by directly applying StC, and others can be implemented efficiently by rewriting them using only privacy-preserving sub-steps. We demonstrated StC's utility by implementing five algorithms, all of which are not only privacy-preserving but also asymptotically optimal. We showed experimentally that these algorithms are efficient and scalable, outperforming the data-oblivious alternatives with similar privacy protection. StC algorithms can be distributed and therefore able to support privacy-preserving computation at scale.

[5] IBM CryptoCards. `http://www-03.ibm.com/security/cryptocards/`

References

1. Software guard extensions programming reference. https://software.intel.com/sites/default/files/managed/48/88/329298-002.pdf
2. Agrawal, R., Kiernan, J., Srikant, R., Xu, Y.: Order preserving encryption for numeric data. In: ACM SIGMOD (2004)
3. Arasu, A., Blanas, S., Eguro, K., Kaushik, R., Kossmann, D., Ramamurthy, R., Venkatesan, R.: Orthogonal security with cipherbase. In: CIDR (2013)
4. Arasu, A., Kaushik, R.: Oblivious query processing. arXiv preprint arXiv:1312.4012 (2013)
5. Bajaj, S., Sion, R.: Trusteddb: A trusted hardware-based database with privacy and data confidentiality. In: TKDE (2014)
6. Baumann, A., Peinado, M., Hunt, G.: Shielding applications from an untrusted cloud with haven. In: OSDI (2014)
7. Blanton, M., Steele, A., Alisagari, M.: Data-oblivious graph algorithms for secure computation and outsourcing. In: ASIACCS (2013)
8. Blum, M., Floyd, R.W., Pratt, V., Rivest, R.L., Tarjan, R.E.: Time bounds for selection. Journal of computer and system sciences (1973)
9. Boneh, D., Mazieres, D., Popa, R.A.: Remote oblivious storage: Making oblivious RAM practical. MIT-CSAIL-TR-2011-018 (2011)
10. Brakerski, Z., Brakerski, Z.: Efficient fully homomorphic encryption from (standard) lwe. In: FOCS (2011)
11. Chaum, D.L.: Untraceable electronic mail, return addresses, and digital pseudonyms. Communications of the ACM (1981)
12. Chen, S., Wang, R., Wang, X., Zhang, K.: Side-channel leaks in web applications: A reality today, a challenge tomorrow. In: IEEE S&P (Oakland) (2010)
13. Chen, Y., Sion, R.: On securing untrusted clouds with cryptography. Data Engineering Bulletin (2012)
14. Dinh, T.T.A., Saxena, P., Chang, E.C., Ooi, B.C., Zhang, C.: M^2R: Enabling stronger privacy in mapreduce computation. In: USENIX Security (2015)
15. ElGamal, T.: A public key cryptosystem and a signature scheme based on discrete logarithms. In: CRYPTO (1984)
16. Gentry, C., et al.: Fully homomorphic encryption using ideal lattices. In: STOC (2009)
17. Goldreich, O., Ostrovsky, R.: Software protection and simulation on oblivious RAMs. Journal of the ACM (1996)
18. Goodrich, M.T.: Data-oblivious external-memory algorithms for the compaction, selection, and sorting of outsourced data. In: SPAA (2011)
19. Goodrich, M.T.: Zig-zag sort: A simple deterministic data-oblivious sorting algorithm running in o (n log n) time. In: STOC (2014)
20. Goodrich, M.T., Mitzenmacher, M.: Privacy-preserving access of outsourced data via oblivious RAM simulation. CoRR abs/1007.1259 (2010)
21. Goodrich, M.T., Ohrimenko, O., Tamassia, R.: Data-oblivious graph drawing model and algorithms. arXiv preprint arXiv:1209.0756 (2012)
22. Katz, J., Lindell, Y.: Introduction to modern cryptography. CRC Press (2014)
23. Klonowski, M., Kutyłowski, M.: Provable anonymity for networks of mixes. In: Information Hiding (2005)
24. Knuth, D.E.: The art of computer programming: sorting and searching, vol. 3. Pearson Education (1998)
25. Li, F., Hadjieleftheriou, M., Kollios, G., Reyzin, L.: Dynamic authenticated index structure for outsourced databases. In: ACM SIGMOD (2006)

26. McCun, J.M., Parno, B., Perrig, A., Reiter, M.K., Isozaki, H.: Flicker: An execution infrastructure for tcb minimization. In: EuroSys (2008)
27. McCune, J., Li, Y., Qu, N., Zhou, Z., Datta, A., Gligor, V., Perrig, A.: TrustVisor: Efficient tcb reduction and attestation. In: IEEE S&P (Oakland) (2010)
28. Ohrimenko, O., Costa, M., Fournet, C., Gkantsidis, C., Kohlweiss, M., Sharma, D.: Observing and preventing leakage in mapreduce. In: CCS (2015)
29. Ohrimenko, O., Goodrich, M.T., Tamassia, R., Upfal, E.: The Melbourne shuffle: Improving oblivious storage in the cloud. In: ICALP (2014)
30. OMalley, O., Murthy, A.C.: Winning a 60 second dash with a yellow elephant. Tech. rep., Yahoo (2009)
31. Paillier, P.: Public-key cryptosystems based on composite degree residuosity classes. In: EUROCRYPT (1999)
32. Pinkas, B., Reinman, T.: Oblivious RAM revisited. In: CRYPTO (2010)
33. Popa, R.A., Redfield, C., Zeldovich, N., Balakrishnan, H.: CryptDB: Protecting confidentiality with encrypted query processing. In: SOSP (2011)
34. Schuster, F., Costa, M., Fournet, C., Gkantsidis, C., Peinado, M., Mainar-Ruiz, G., Russinovich, M.: VC^3: Trustworthy data analytics in the cloud. In: IEEE S&P (Oakland) (2014)
35. Stefanov, E., Van Dijk, M., Shi, E., Fletcher, C., Ren, L., Yu, X., Devadas, S.: Path ORAM: An extremely simple oblivious RAM protocol. In: CCS (2013)
36. Tu, S., Kaashoek, M.F., Madden, S., Zeldovich, N.: Processing analytical queries over encrypted data. In: PVLDB (2013)

Appendix A The scrambler

The scrambler $\mathbb{S}$ first generates the permutation π using a pseudo-random permutation [22] with a sufficiently long secret seed. Next, it executes the Melbourne shuffle algorithm with π and X as input, outputting the scrambled data $\widetilde{X}$. $\mathbb{S}$ inherits its security from that of the Melbourne shuffle algorithm [29]. We assume that the pseudorandom permutation is secure; i.e. it is indistinguishable from a randomly and uniformly selected permutation.

The Melbourne shuffle consists of two phases: distribution and clean-up, and processes data in blocks independently of one another. This allows distributing the scrambler $\mathbb{S}$ over multiple nodes to achieve substantial speedups.

The performance of Melbourne shuffle algorithm is configurable by two variables p_1, p_2, and it aborts (and restarts) with a negligible probability [29]. If the adversary has observed that the algorithm restarts, he gains some information of the permutation. Such probability can be reduced by always executing the algorithm multiple times, at the expense of longer running time. Note that there is also a negligible probability that the adversary correctly guesses the secret seed and hence the permutation, which can be further reduced by increasing the length of the secret seed. The security parameter κ used in the notion of indistinguishability determines the seed's length and the number of times the Melbourne shuffle algorithm is repeated.

Appendix B Proof of Theorem 1

The proof consists of two parts. We first show that an algorithm constructed from the imitator is privacy-preserving, based on which we then prove that the combined algorithm $\mathcal{A}_{\mathcal{P}}$ is privacy-preserving.

Part 1. Let $\mathcal{A}_{\mathcal{P}^*}$ be the combined algorithm which executes the scrambler $\mathbb{S}$ follows by $\mathcal{P}^*$, we shall show that $\mathcal{A}_{\mathcal{P}^*}$ is privacy-preserving. Recall $\mathcal{P}^*$ is an algorithm indicated in the description of the imitator and its input is a permuted sequence of $\langle 1, 2, \ldots, n\rangle$. Let us call each element in such sequence a *tag*. Under $\mathcal{A}_{\mathcal{P}^*}$, the tags are first scrambled, and then fed to $\mathcal{P}^*$. Intuitively, even if the scrambled tags are revealed, the adversary is still unable to gain information on the input by observing the executions of $\mathcal{A}_{\mathcal{P}^*}$. We formally reason such intuition in the following.

Let $\widetilde{\mathbb{S}}$ be a variant of $\mathbb{S}$ with an additional final step: after the data is scrambled, $\widetilde{\mathbb{S}}$ intentionally reveals the scrambled data. This can be done by revealing the sequence via the meta data *info* in the read/write requests. Let $\widetilde{\mathcal{A}}_{\mathcal{P}^*}$ be the combined algorithm that applies $\widetilde{\mathbb{S}}$ follows by $\mathcal{P}^*$. Clearly, if $\widetilde{\mathcal{A}}_{\mathcal{P}^*}$ is privacy-preserving, then so is $\mathcal{A}_{\mathcal{P}^*}$.

Let us denote by T_1 and T_2 the two permuted sequence $\pi_1(\langle 1, 2, \ldots, n\rangle)$ and $\pi_2(\langle 1, 2, \ldots, n\rangle)$ where π_1 and π_2 are any two permutations. Let us also denote by R_1 and R_2 the scrambled tags $\widetilde{\mathbb{S}}(T_1)$ and $\widetilde{\mathbb{S}}(T_2)$. By the security of the scrambler, the access sequence and revealed tags $\langle Q_{\widetilde{\mathbb{S}}}(T_1), R_1\rangle$ is indistinguishable from $\langle Q_{\widetilde{\mathbb{S}}}(T_2), R_2\rangle$, and hence R_1 and R_2 are indistinguishable. Accordingly, it follows that the access sequences generated by $\mathcal{P}^*$ (i.e. $Q_{\mathcal{P}^*}(R_1)$, $Q_{\mathcal{P}^*}(R_2)$) are indistinguishable from each other. Overall, $Q_{\widetilde{\mathcal{A}}_{\mathcal{P}^*}}(T_1)$ is indistinguishable from $Q_{\widetilde{\mathcal{A}}_{\mathcal{P}^*}}(T_2)$. Therefore $\widetilde{\mathcal{A}}_{\mathcal{P}^*}$ is privacy-preserving, and so is $\mathcal{A}_{\mathcal{P}^*}$.

Part 2. We next show by contradiction that $\mathcal{A}_{\mathcal{P}}$ is privacy-preserving. Suppose $\mathcal{A}_{\mathcal{P}}$ is not privacy-preserving, i.e., there exist X_1, X_2 having the same number of records and an algorithm Adv that can distinguish $Q_{\mathcal{A}_{\mathcal{P}}}(X_1)$ and $Q_{\mathcal{A}_{\mathcal{P}}}(X_2)$. Let $T_1 = \mathcal{T}(X_1)$ and $T_2 = \mathcal{T}(X_2)$ be the corresponding tags. We can construct a distinguisher $\mathcal{D}$ that differentiates $Q_{\mathcal{A}_{\mathcal{P}^*}}(T_1)$ and $Q_{\mathcal{A}_{\mathcal{P}^*}}(T_2)$ by simulating Adv. Recall that $Q_{\mathcal{P}}(\pi(X)) = Q_{\mathcal{P}^*}(\pi(\mathcal{T}(X)))$ for any permutation π, it shall follow that $Q_{\mathcal{A}_{\mathcal{P}}}(X) = Q_{\mathcal{A}_{\mathcal{P}^*}}(\mathcal{T}(X))$. When given access sequences generated by $\mathcal{A}_{\mathcal{P}^*}$, $\mathcal{D}$ can transform them to the corresponding access sequences generated by $\mathcal{A}_{\mathcal{P}}$. By simulating Adv on the transformed sequences, it is able to distinguish $Q_{\mathcal{A}_{\mathcal{P}^*}}(T_1)$ and $Q_{\mathcal{A}_{\mathcal{P}^*}}(T_2)$, contradicting the fact that $\mathcal{A}_{\mathcal{P}^*}$ is privacy-preserving proven in *Part 1*. Therefore, $\mathcal{A}_{\mathcal{P}}$ must be privacy-preserving. □

Appendix C Supporting Spark Operations

To demonstrate the generality of our proposed approach, we apply StC on Spark functions. Since Spark is a general computing framework, supporting these functions in StC makes it easy for developers to build complex privacy-preserving applications. Table 4 summarizes our effort. Some functions benefit immediately from StC, similar to pSort and pSelect, while other func-

Table 4: List of Spark's functions supported in StC.

	Scramble-then-compute	*Composition*
Privacy preserving	map, filter, mapPartition, sample, distinct, sortByKey, cartesian, repartition, count, first, takeOrdered	union, intersection, reduceByKey, aggregateByKey, join, cogroup, reduce, takeSample, countByKey
Ψ-privacy-preserving	flatMap, groupByKey, repartitionAndSortWithinPartitions	

tions require rewriting the original algorithms to be composed of other privacy-preserving steps. Almost all of these functions are privacy-preserving, except for *flatMap*, *groupByKey*, and *repartitionAndSortWithinPartitions* which are Ψ-privacy-preserving. The permissible leakage Ψ of those algorithms is output records' distribution, which conveys a certain information about the distribution of the input records with respect to their key or partition. For example, the *groupByKey* function reveals how many input records sharing the same key.

Table 5: Number of re-encryptions and I/O complexity required by StC's algorithms and relevant data-oblivious algorithms. n is the input size, p_1 and p_2 are constant parameters in the scrambler's configuration. In our experiments, $p_1 = p_2 = 2$. $s = n/m$ and d is the average degree of records in the join graph.

Algorithm	*# Re-Encryptions*	*I/O Complexity*
pSort	$(p_1 + p_2 + 5) \cdot n$	$O(n)$
obliSort[20]	$(\sum_{i=1}^{\log s} i + \log s + 1) \cdot n$	$O(n \log^2 n)$
pCompact	$(p_1 + p_2 + 2) \cdot n$	$O(n)$
obliCompact[18]	$(1 + \log n) \cdot n$	$O(n \log n)$
pSelect	$(p_1 + p_2 + 4) \cdot n$	$O(n)$
obliSelect[18]	$\frac{1}{2}(4 + \log n) \cdot n$	$O(n)$
pAggr	$(2p_1 + 2p_2 + 8) \cdot n$	$O(n)$
obliAggregate[3]	$(\sum_{i=1}^{\log s} i + \log s + \log n + 3) \cdot n$	$O(n \log^2 n)$
pJoin	$(3p_1 + 3p_2 + 9 + d) \cdot n$	$O(dn)$
obliJoin[4]	$(\sum_{i=1}^{\log s} i + \log s + 2 \log n + 5 + d) \cdot n$	$O(n \log^2 n)$

Appendix D I/O complexity of STC algorithms

Table 5 details the numbers of I/O and cryptographic operations. StC algorithms require $O(n)$ I/Os with a small constant factor, whereas all data-oblivious algorithms, except for obliSelect, have super-linear I/O complexity. I/O complexity of the join algorithm 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). The number of re-encryptions of StC algorithms depends on the number of re-encryptions per scrambling step: $n(p_1 + p_2)$. In our experiments, we find that for the datasets under consideration, with $p_1 = p_2 = 2$, the scrambler achieves optimal performance. On the other hand, the numbers of re-encryptions required by data-oblivious algorithms depend only on the size of the secure memory. With secure memory of size $m = c\sqrt{n}$ (where c is a small constant larger than one), the data-oblivious algorithms perform a few times more re-encryptions than StC algorithms, which directly translates to considerable performance overheads.