Hung Dang*, Tien Tuan Anh Dinh, Ee-Chien Chang, and Beng Chin Ooi

Privacy-Preserving Computation with Trusted Computing via Scramble-then-Compute

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, for data movement observed during computation could leak information. While it is possible to thwart such leakage using generic solution such as ORAM [42], designing efficient privacy-preserving algorithms is challenging. Besides computation efficiency, it is critical to keep trusted code bases lean, for large ones are unwieldy to vet and verify. In this paper, we advocate a simple approach wherein 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 an extensive body of known efficient algorithms for better performance. Our experiments show that StC could offer up to 4.1× speedups over known, application-specific alternatives.

DOI 10.1515/popets-2017-0025
Received 2016-11-30; revised 2017-03-15; accepted 2017-03-16.

1 Introduction

Big data is one of the main driving forces 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, for the data is being handled by untrustworthy parties. Even if the providers were trusted, other factors like multi-tenancy, complexity of software stacks, and distributed computing models would continue to enlarge the attack surface [17, 20]. 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 [18].

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 [33]. Fully homomorphic encryption schemes allow for computations over encrypted data, but suffer from prohibitive overheads [15, 22]. Partially homomorphic encryption schemes [21, 41] are more practical, but limited in the range of supported operations [43, 46].

A line of recent works have advocated an approach of combining encryption with trusted computing primitives that offer a confidentiality and integrity protected execution environment [8, 11, 44]. This trusted environment can be provisioned by either hardware (e.g., IBM 4767 PCIeCC2 [3], Intel SGX [4]) or hardware-software combination [34, 35]. Under this approach, data is stored in untrusted external memory/storage and protected by semantically secure encryption. Confidentiality is protected since the data is only decrypted and processed in the trusted execution environment, and the outputs are encrypted before being written back to the storage.

Nevertheless, the trusted environment has a limit on the amount of data it can process at any time. 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 [17, 20, 45]. For instance, by observing I/O access patterns during merge sort, an attacker can infer the order of the input records, i.e., their ranks in the output. This leakage could be eliminated by using generic oblivious-RAM (ORAM), but with high performance overheads [42]. Due to these overheads, ORAM are more suitable for applications

***Corresponding Author: Hung Dang:** National University of Singapore, E-mail: hungdang@comp.nus.edu.sg
Tien Tuan Anh Dinh: National University of Singapore, E-mail: dinhtta@comp.nus.edu.sg
Ee-Chien Chang: National University of Singapore, E-mail: changec@comp.nus.edu.sg
Beng Chin Ooi: National University of Singapore, E-mail: ooibc@comp.nus.edu.sg

which make few accesses in a large dataset, but not necessarily being so for other applications that require accessing the entire dataset multiple times. For those applications, customised data-oblivious algorithms often perform better [39]. However, designing these algorithms are challenging, and existing constructions are complex, which indirectly leads to large trusted code bases (TCB) that are difficult to vet and verify.

Our goal is to design algorithms that are privacy-preserving and practical while keeping the TCB lean. To this end, we observe 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 a 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 such observation, we advocate an approach for designing privacy-preserving algorithms which we call *Scramble-then-Compute* (StC). This approach essentially scrambles the input before executing the original algorithm on the scrambled data. StC not only is applicable to a large number of algorithms, but also incurs only an additive overhead factor (as opposed to a (amortized) multiplicative factor when using ORAM). Its generality facilitates code reuse, i.e., it allows us to harvest an extensive body of existing works on efficient algorithms to achieve desirable performance. For example, built on top of the external merge sort algorithm, the privacy-preserving sorting algorithm implemented under StC outperforms the data-oblivious sorting algorithm specially designed for external memory setting [26] by upto 4.1×. Furthermore, the scrambling step in StC is easily distributed, enabling the algorithms constructed under StC to scale. Of equal importance is the simplicity of our solution, which promises an ease of implementation and vetting.

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). While it may appear that StC has limited use case, we remark that it is capable of supporting an expressive class of privacy-preserving computations. In particular, StC is inherently applicable to various algorithms involving data movement such as merge-sort, quicksort or compaction[1]. Moreover, it is also compatible with Spark - a general computing framework for large-scale data processing [1]. To demonstrate its practicality, we describe privacy-preserving implementations of five popular algorithms: *sort, compaction, selection, aggregation* and *join*, which are the core to various data management applications (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 against baseline implementations that are not privacy-preserving, showing that StC offers a stronger privacy protection at a cost of 3.5× overhead on average. We also compare them with state-of-the-art data-oblivious algorithms that are tailor-made for the above mentioned applications [8, 9, 24, 26]. Experimental results manifest that privacy-preserving algorithms constructed under StC can achieve speedups as high as 4.1× over the data-oblivious alternatives. Furthermore, these algorithms are arguably easier to parallelise. The improvement on performance is probably gained by harvesting an extensively studied body of work on external memory algorithms. In summary, we make the following contributions:

1. We present 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.
2. We demonstrate the utility of StC by applying it to five data management algorithms (including sort, compaction, selection, aggregation and join), 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. We also show that StC is applicable to Spark – a general computing framework [1], enabling developers to build complex privacy-preserving applications at ease.
3. We conduct extensive experiments to evaluate the privacy-preserving algorithms constructed under StC. The results indicate relatively low overheads over the baseline system that is less secure, and run-

1 There is a subtle issue with non-unique elements. We shall discuss techniques to address it in Section 3.5.

ning 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.

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.

We shall use the following running example to illustrate the problem and its related concepts. Let us consider a user outsourcing her data comprising integer-value records to the cloud. The outsourced data is protected by semantically secure encryption. When the user wishes to sort the data, perhaps as a pre-processing step for other tasks such as ranking, she relies on a trusted unit which processes 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 can hold only three records at a time. The input consists of nine records, and sorting involves one merging step.

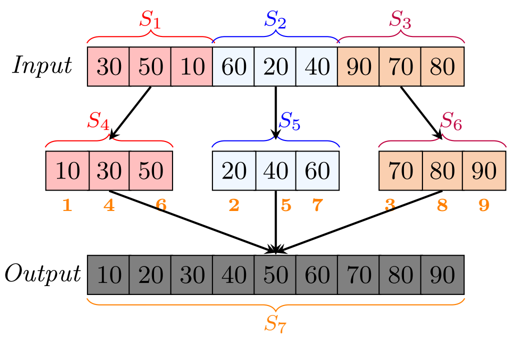


Fig. 1. An example of three-way external merge sort on encrypted records. The subscripts denote the order in which the record is read into the trusted unit during the merging step.

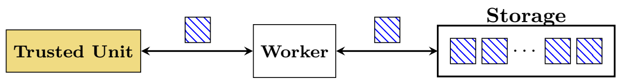


Fig. 2. The system model consists of a trusted unit which can process a limited number of records at a time. The storage and worker are untrusted. Only the trusted unit can see the content of the encrypted records (denoted by hatched squares).

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 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 that 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 records in X. In our running example, $\mathcal{P}$ is the three-way merge sort algorithm, the input X comprises nine records, 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 having 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 threat model is realistic, given recent security breaches (e.g., NSA Target List [5]) being attributed to insider threats. A malicious adversary, in contrast, can modify data in the storage and deviate the worker from its execution path. Such adversaries are considered by other works [20, 44]. 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 that 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 necessarily secure) that 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. 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. It is desirable to keep the runtime overhead low. To prevent leakage, one could employ oblivious RAM [45] directly on the storage backend. This approach incurs $\Omega(\log n)$ (amortized) overhead per access, making it impractical for large scale data processing. Alternatively, one can use application-specific data-oblivious algorithms [26], which are complex yet limited in scope. Therefore, offering privacy protection while enabling adoption of 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 vulnerability-free, small TCB is preferable since verifying a large code base is unwieldy. StC (presented in Section 3) essentially adds a *scrambler* to existing external memory algorithms in order to achieve security. Its simplicity leads to small TCB and low performance overhead. Furthermore, its ability to parallelize 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 first eight observed accesses that the trusted unit made on the external storage during the merging phase of the merge sort is: $\big\langle \langle \mathtt{r}, S_4, \perp\rangle, \langle \mathtt{r}, S_5, \perp\rangle, \langle \mathtt{r}, S_6, \perp\rangle, \langle \mathtt{w}, S_7, \perp\rangle, \langle \mathtt{r}, S_4+1, \perp\rangle, \langle \mathtt{w}, S_7+1, \perp\rangle, \langle \mathtt{r}, S_5+1, \perp\rangle, \langle \mathtt{w}, S_7+2, \perp\rangle \big\rangle$, in which S_i denotes the address of the block S_i (this is also the address of the first record in the block), and $S_i + j$ denotes the address of the j^{th} record in that block. The first five observed accesses have already revealed the fact that the smallest record is from S_4, which in turns implies that it originated from S_1.

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 the definition, the number of records in the dataset is a polynomial of κ (hence the size of the private memory is also a polynomial of κ).

Relationship to data obliviousness. Closely related to our security definition is the notion of *data obliviousness* [24], 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 the same for any X, then a Ψ-privacy-preserving algorithm is also privacy-preserving.

3 Scramble Then Compute

3.1 Overview

In this section, we present StC – an approach for implementing privacy-preserving external memory algorithms that follow the computation model described earlier in Section 2. Given an algorithm $\mathcal{P}$ that 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.

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 this approach. Others such as join and aggregation are made privacy-preserving by exploiting the composition property, i.e., restructuring the original algorithms to be composed of only privacy-preserving substeps.

The privacy protection that $\mathcal{A}_{\mathcal{P}}$ offers can also be achieved by executing the original algorithm $\mathcal{P}$ on an ORAM protocol [42, 45]. In fact, one can derive a privacy-preserving execution of any algorithm using an ORAM protocol, but with a (amortized) *multiplicative* overhead factor of $\Omega(\log n)$ where n is the input size [45]. In other words, for every real access that $\mathcal{P}$ incurs, the ORAM execution will require accessing $\Omega(\log n)$ records to hide its access pattern. Recall that records are to be re-encrypted every time they are accessed, this overhead translates to $10 - 100\times$ slowdown when processing gigabytes of data [20].

StC, on the other hand, adds only an *additive* overhead of $O(n)$ to the execution. Thus, for computations that process the *entire* dataset, such as sort, aggregation, join and other data management algorithms, StC clearly offers better performance, both in terms of asymptotic complexity and practical running time.

One drawback of StC is that it is not applicable to all algorithms. Nevertheless, we would like to remark that the class of permutation-invariant computations that StC covers is expressive and especially relevant in the context of big data management. In fact, StC can support all computations that are available in Spark. We further elaborate on this in Section 4.

For clarity, we summarize notations that are used throughout the rest of the paper in Table 1.

3.2 The Scrambler

We base the scrambler $\mathbb{S}$ on the recently proposed data-oblivious Melbourne shuffle algorithm by Ohrimenko *et al.* [38]. We first describe the Melbourne shuffle algorithm before discussing construction of $\mathbb{S}$.

Table 1. Summary of Notations

Notation	*Description*
X	Input data
$\widetilde{X}$	Scrambled input
Y	Output data
π	Random permutation
$\mathbb{S}$	The Scrambler
n	Input size
m	Trusted memory's size
$\mathcal{T}$	Tagging algorithm
$Q_{\mathcal{P}}(X)$	Access sequence of algorithm $\mathcal{P}$ on input X
Ψ	Leakage function
$\mathcal{A}_{\mathcal{P}}$	Privacy-preserving algorithm derived from a permutation-invariant algorithm $\mathcal{P}$
p_1, p_2	Configurable parameters of the Melbourn shuffle algorithm

3.2.1 Building block: Melbourne Shuffle algorithm

The Melbourne shuffle algorithm [38] follows the computation model described in Section 2. In particular, it assumes a trusted unit with private memory of size $O(\sqrt{n})$ where n is the input size, and records are only decrypted inside the trusted unit and re-encrypted before being written back to the storage.

The algorithm takes as input a data set X comprising n items and a randomly chosen permutation π, and obliviously arranges the n items to their final position in the output $\widetilde{X}$ according to π. The permutation π can be generated using a pseudo random permutation [29], and represented by a short secret seed. The algorithm is data-oblivious in a sense that it incurs *the same* access sequence for all X of the same size.

The shuffling requires two intermediate arrays T_1 and T_2 which are of size $p_1 n$ and $p_2 n$, respectively (p_1 and p_2 are constants larger than one and $p_2 \geq p_1$). Records in X, T_1, T_2 and $\widetilde{X}$ are grouped into *buckets*. Each bucket of T_1 and T_2 can hold upto $p_1\sqrt{n}$ and $p_2\sqrt{n}$ records, respectively. The buckets are further grouped into *chunks*. Each chunk consists of exactly $\sqrt[4]{n}$ buckets, hence there are $\sqrt[4]{n}$ chunks in total.

The algorithm proceeds in three phases: two *distribution* phases and a single *clean-up* phase. In the first distribution phase, the trusted unit reads batches of $\sqrt{n}$ records from X, splits records into $\sqrt[4]{n}$ segments according to their final positions indicated by π, and writes the i^{th} segment to the i^{th} chunk in T_1. If a segment contains less than $p_1\sqrt[4]{n}$ records (i.e., half-full), dummies are added to ensure data-obliviousness. On the other hand, if some segment contains more than $p_1\sqrt[4]{n}$ records, the algorithm fails. At the end of this phase, records are placed in correct chunk, but not the correct bucket within the chunk. In the second distribution phase, the trusted unit reads buckets of $p_1\sqrt{n}$ records in each chunk of T_1, ignores dummies, divides the real records into $\sqrt[4]{n}$ segments according to their final positions, and writes the j^{th} segment to the j^{th} bucket of the same chunk in T_2. If a segment has less than $p_2\sqrt[4]{n}$ records, it is padded with dummies. If some segment contains more than $p_2\sqrt[4]{n}$ records, the algorithm fails. At the end of this phase, records are placed in the correct bucket, but not necessarily at the correct positions within that bucket. Finally, the clean-up phase removes dummies and arranges real records to correct positions within their own bucket.

The Melbourne shuffle algorithm runs in $O(n)$ time, failing with negligible probability:

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

where negl(n) is a negligible function [38].

3.2.2 The Scrambler Construction

The scrambler $\mathbb{S}$ is a probabilistic algorithm that takes as input a dataset X and outputs a permuted sequence $\widetilde{X} = \pi(X)$ where π is a random permutation[2]. We reason about the security of $\mathbb{S}$ using a notion of indistinguishability.

Recall that in our computation model, input and output records are always encrypted. Let us denote by $\widetilde{\mathbb{S}}$ a variant of $\mathbb{S}$ with one additional step that decrypts all records in the output $\widetilde{X}$ at the end. Intuitively, we would like $\mathbb{S}$ to transform X to $\widetilde{X} = \pi(X)$ without revealing any information on X and π. More formally, we say that $\mathbb{S}$ is *secure* if (1) it is privacy-preserving, and (2) $\widetilde{\mathbb{S}}$ is Ψ-privacy-preserving with respect to a deterministic function Ψ that outputs the sorted sequence of the input X. Condition (1) ensures that no information on X is leaked, while condition (2) guarantees that the permutation π is not revealed.

We construct $\mathbb{S}$ based on the Melbourne shuffle algorithm. Another (arguably simpler) construction of $\mathbb{S}$ could be based on Chaum's mix-network [16], which achieves statistical indistinguishability, but requires large trusted memory [30].

The Melbourne shuffle algorithm may fail with a negligible probability, requiring repeating the shuffling

2 It is not necessary that every permutation is equally likely.

with another random seed. This leads to a probabilistic running time. Fortunately, the probability of failure is negligible and thus it is still computationally infeasible for an adversary to distinguish the access patterns of the algorithm on different inputs of the same size via timing attack. Nevertheless, since it is not straightforward to reason about the complexity of combined StC algorithms should $\mathbb{S}$ have probabilistic running time, we implement $\mathbb{S}$ using a variant of the Melbourne shuffle whose running time is deterministic.

$\mathbb{S}$ first generates a permutation π_o using a secure pseudo-random permutation [29]. Next, it executes the Melbourne shuffle algorithm with π_o and X as input. If the underlying Melbourne shuffle completes without failure, $\mathbb{S}$ outputs $\widetilde{X} = \pi_o(X)$. In case there is a segment containing more than an expected number of records (i.e., $p_1 \sqrt[4]{n}$ in the first distribution phase, and $p_2 \sqrt[4]{n}$ in the second distribution phase), instead of failing as in the original algorithm, $\mathbb{S}$ distributes the overflowing records (called *outliers*) among other half-full segment(s). Once the outliers are consumed, segments that are still half-full will be padded with dummies. In the clean-up phase, the trusted unit scans through the intermediate array T_2, removing the dummies and outputting $\widetilde{X} = \pi(X)$ where π is some secret permutation.

The two distribution phases of $\mathbb{S}$ are data-oblivious, while its clean-up phase is privacy-preserving. The access patterns of the clean-up phase (and thus $\mathbb{S}$) are different only if some bucket in T_2 contains more than $\sqrt{n}$ records. Fortunately, such event happens with negligible probability (upper bounded by the probability wherein the Melbourne shuffle fails). It can be proven that $\mathbb{S}$ is secure according to the definition put forth earlier.

Similar to the Melbourne shuffle, $\mathbb{S}$ runs in $O(n)$ time. It is invoked in every run of a privacy-preserving algorithm constructed under StC. If an algorithm comprises privacy-preserving sub-steps, $\mathbb{S}$ is invoked separately in each sub-step. Without loss of generality, we can consider the number of such sub-steps a constant. Hence, $\mathbb{S}$ adds an additive overhead of $O(n)$ to the execution of StC algorithms.

3.3 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. Let us call $\mathcal{A}_\mathcal{P}$ the combined StC 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 it is permutation-invariant, and the output $\mathcal{T}(X)$ is a permuted sequence of $\langle 1, 2, \ldots, n\rangle$, where n is the number of records in X.*

Let us call the output $\mathcal{T}(X)$ the *tags*. A tagging algorithm can, on input of a sequence of integers $\langle 50, 10, 30, 1\rangle$, output tags $\langle 4, 2, 3, 1\rangle$ representing the record ranks in the input according to ascending order.

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 $\mathcal{P}^*$, when operating on the tags, essentially incurs the same observable behaviour as $\mathcal{P}$ does on X. We now give a sufficient condition for combined StC algorithm $\mathcal{A}_\mathcal{P}$ 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.*

Proof. [Main idea] The proof consists of two parts. The first part shows that an algorithm $\mathcal{A}_{\mathcal{P}^*}$ constructed under StC which scrambles the tags before feeding them to $\mathcal{P}^*$ is privacy-preserving. This is to manifest that $\mathcal{A}_{\mathcal{P}^*}$ leaks no information on the tags. The second part extends the result to the original input X, i.e., showing that if $\mathcal{A}_{\mathcal{P}^*}$ is privacy-preserving, then so is $\mathcal{A}_\mathcal{P}$. This is not surprising, for the tags do not reveal content of X, and $\mathcal{P}^*$ incurs the same access pattern as $\mathcal{P}$ does.

Part 1. Let us denote by $\mathcal{A}_{\mathcal{P}^*}$ the algorithm that combines $\mathbb{S}$ and $\mathcal{P}^*$, and by T its input (comprising tags). $\mathcal{A}_{\mathcal{P}^*}$ first scrambles T by $\mathbb{S}$, and then feeds the scrambled tags $R = \mathbb{S}(T)$ to $\mathcal{P}^*$. We note that T and R always remain encrypted outside of the trusted unit. To prove that $\mathcal{A}_{\mathcal{P}^*}$ is privacy-preserving, intuitively, we want to show that even if R are revealed (i.e., by decrypting and revealing the plaintexts to the adversary), and $\mathcal{P}^*$ is not privacy-preserving, the adversary is still unable to obtain sensitive information on T. We formally reason such intuition in the following.

Let $\widetilde{\mathbb{S}}$ be a variant of $\mathbb{S}$ with an additional final step: after scrambling T, $\widetilde{\mathbb{S}}$ intentionally decrypts and reveals plain-text of R via the meta data *info* in the read/write requests. Let $\widetilde{\mathcal{A}}_{\mathcal{P}^*}$ be an algorithm that applies $\widetilde{\mathbb{S}}$ on its input before feeding the output of $\widetilde{\mathbb{S}}$ to $\mathcal{P}^*$. 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 two sequences $\pi_1(\langle 1, 2, \ldots, n\rangle)$ 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 scrambled tags $\widetilde{\mathbb{S}}(T_1)$ and $\widetilde{\mathbb{S}}(T_2)$. By the security of the scrambler, the distribution of the access sequence and revealed tags $\langle Q_{\widetilde{\mathbb{S}}}(T_1), R_1\rangle$ is indistinguishable from that of $\langle Q_{\widetilde{\mathbb{S}}}(T_2), R_2\rangle$, 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 imitating *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 imitating *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 above. Therefore, $\mathcal{A}_{\mathcal{P}}$ must be privacy-preserving.

□

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 $\mathcal{T}$ is the one that outputs the record ranks, and $\mathcal{P}^*$ is a comparison-based algorithm similar to $\mathcal{P}$ but operates on the record 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 have 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. By hybrid argument [29], it is feasible to derive a privacy-preserving algorithm via a composition of other privacy-preserving algorithms. In specific, if algorithms $\mathcal{P}_1$ and $\mathcal{P}_2$ are privacy-preserving, their composition, i.e., executing one after another wherein the output of $\mathcal{P}_1$ is the input of $\mathcal{P}_2$, is also privacy-preserving. We note subtly that only a polynomial number of compositions (with respect to κ) are allowed. In another words, the number of combined privacy-preserving algorithms cannot be arbitrarily large. Interested readers can refer to [29] for a detailed discussion. Nevertheless, this restriction does not affect the utilization of StC in practice, for practical algorithms do not contain a very large number of substeps. We demonstrate the composition property using two examples of aggregation and join algorithms in Section 4.4 and 4.5.

3.4 Handling Duplicates

The permutation-invariant condition is strict, for it requires the output of the algorithm operating on the scrambled data to be *exactly* the same as that when it operates on the original input. For example, a merge sort algorithm operating on duplicate records does not meet this condition. Consider $X = \langle 0_0, 0_1, 0_2, 0_3, 0_4, 0_5\rangle$ 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 $\mathbb{S}$. 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 that are not permutation-invariant can be made so by introducing a pre-processing step that appends metadata to the input, then reversing the effect via a corresponding post-processing step. We use this technique in deriving privacy-preserving implementations of sort and selection algorithms (Section 4).

3.5 Discussion

We stress the simplicity StC offers in deriving privacy-preserving algorithms from existing algorithms. One immediate benefit is code reuse. For example, there exist 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 for a particular problem setting at hand. Another benefit is the small TCB, for we can choose an algorithm with small codebase. This is in contrast 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 Computations with StC

We demonstrate the utility of StC by showing privacy-preserving implementations of five algorithms: *sort, compaction, selection, aggregation* and *join*. These algorithms are the core to various data management applications. Sort is fundamental to any database systems. Compaction is vital in many distributed key-value stores where updates are directly appended to disk and compaction is frequently scheduled to improve query performance [6, 7, 32]. Selection is essential in order statistic. Aggregation is widely used in decision support systems to summarize data, making it an integral part of data warehouse systems. Join is arguably one of the most important operations in data management, and commonly used for data integration that is becoming more important given the variety of data sources [28]. By showing that StC is applicable to these algorithms, we would like to remark that it can be generalised to support a wide range of data management applications.

Among the five algorithms under consideration, the first three are realized directly through StC, and the other two are constructed 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. We summarize in Table 2 the time complexities of the StC algorithms in comparison with the baseline algorithms that are not privacy-preserving and the corresponding oblivious algorithms offering similar level of privacy protection.

We further illustrate how StC can be generalized to support generic privacy-preserving computations at scale by applying it on basic operations in Spark [1] (Section 4.6). Owing to the facts that Spark is a general computing framework for scalable data processing and that it witnesses a steady adoption in various application domains [36, 49], supporting Spark computations in StC promises a capability of building complex and general privacy-preserving applications with ease of design and implementation.

4.1 Sort

The algorithm sorts the input according to a certain order of the record keys. We consider the ExternalMergeSort algorithm [31], 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 (i.e., MakeKeyDistinct()) that appends the address of each record to its key, i.e., $key(x'_i) = key(x_i)||i$. The result is then forwarded to $\mathbb{S}$, 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 (i.e., RevertKey()) scans through the output and removes the address information.

Table 2. 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	$O(n)$	$O(n)$	$O(n)$
Aggregation	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
Join	$O(n_1 \log n_1 + n_2 \log n_2)$	$O(n_1 \log n_1 + n_2 \log n_2 + l \log l)$	$O(n_1 \log n_1 + n_2 \log n_2 + l \log l)$

This derived algorithm, called pSort, is detailed in Algorithm 1. pSort runs in $O(n \log n)$ time. The pre-processing and post-processing steps make the original algorithm permutation-invariant. We can construct its imitator by specifying the two algorithm $\mathcal{P}^*$ and $\mathcal{T}$. The tagging algorithm $\mathcal{T}$ on input X outputs the sequence of record ranks, and $\mathcal{P}^*$ is essentially the external merge sort algorithm operating on the record ranks. For example, if $X = \langle 50, 30, 10, 1\rangle$, then $\mathcal{T}(X) = \langle 4, 3, 2, 1\rangle$; and $\mathcal{P}^*$ is executed on $\mathcal{T}(X)$. It is trivial to see that $Q_{\mathcal{P}}(\pi(X)) = Q_{\mathcal{P}^*}(\pi(\mathcal{T}(X)))$ for any permutation π and input X, where $\mathcal{P}$ is the underlying ExternalMergeSort algorithm. Thus, by Theorem 1, pSort is privacy-preserving. To the best of our knowledge, the most efficient data-oblivious sorting algorithms run in $O(n \log^2 n)$ [26]. We note that there also exists a randomized oblivious sort algorithm that runs in $O(n \log n)$ time [24]. However, it features a large constant factor and is not necessarily faster than the $O(n \log^2 n)$ version.

```
procedure pSort($X$)
    $X' \leftarrow$ MakeKeyDistinct($X$);
    $\widetilde{X} \leftarrow \mathbb{S}(X')$;
    $Y' \leftarrow$ ExternalMergeSort($\widetilde{X}$);
    $Y \leftarrow$ RevertKey($Y'$);
    return $Y$;
end procedure
```

We emphasize the efficiency of the pSort algorithm. Given a number of privacy-preserving solutions relying on the sorting primitive [39, 42], having an efficient implementation of a privacy-preserving sorting algorithm is certainly of significant interest.

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 – Filer() – 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. In the first step (i.e., Mark(X)), 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 labelling. The next two steps involve running the labelled input through $\mathbb{S}$ and then the baseline algorithm. Finally, the trusted unit reveals the labels to the worker so that the latter can move records to their final positions (i.e., Arrange()).

pCompact runs in $O(n)$, while the data-oblivious alternative [24] runs in $O(n \log n)$. The sub-procedure Mark() and Arrange() make the original algorithm permutation-invariant. We show the security of pCompact by considering the following imitator $\langle \mathcal{P}^*, \mathcal{T}\rangle$. The tagging algorithm $\mathcal{T}$ associate with each record x_i in X a counter t_i according to the following two rules. 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. The algorithm $\mathcal{P}^*$ is essentially the same baseline algorithm operating on $\mathcal{T}(X)$. Here, the permissible leakage $\Psi(X)$ is the number of marked records in X (the data-oblivious algorithm [24] also reveals this information), which is accessible to $\mathcal{P}^*$. Since $Q_{\mathcal{P}}(\pi(X)) = Q_{\mathcal{P}^*}(\pi(\mathcal{T}(X)), \Psi(X))$ for any X and π, where $\mathcal{P}$ is the underlying filter algorithm, by theorem 2, pCompact is Ψ-privacy-preserving, where $\Psi(X)$ reveals the output size.

```
1: procedure pCompact($X$)
2:     $X' \leftarrow$ Mark($X$)
3:     $\widetilde{X} \leftarrow \mathbb{S}(X')$
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
```

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 MedianOfMedians algorithm [13] that 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.

```
1: procedure pSelect($X, k$)
2:     $X' \leftarrow$ MakeKeyDistinct($X$);
3:     $\widetilde{X} \leftarrow \mathbb{S}(X')$
4:     $Y' \leftarrow$ MedianOfMedians($\widetilde{X}$);
5:     $Y \leftarrow$ RevertKey($Y'$);
6:     return $Y$;
7: end procedure
```

pSelect runs in $O(n)$ time, having the same complexity as the existing data-oblivious alternative [24]. We show in the next section, however, that in practice pSelect outperforms its data-oblivious counterpart by a few times. The original algorithm is made permutation-invariant because of the preprocessing and post-processing steps. Its imitator comprises the tagging algorithm $\mathcal{T}$ that outputs the sequence of record ranks, and $\mathcal{P}^*$ which is essentially the MedianOfMedians algorithm being executed on the record ranks. It is straightforward to observe that $Q_{\mathcal{P}}(\pi(X)) = Q_{\mathcal{P}^*}(\pi(\mathcal{T}(X)))$ for any permutation π and input X, where $\mathcal{P}$ is the underlying median of medians algorithm. Thus, pSelect is privacy-preserving according to Theorem 1.

4.4 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 that first sorts the input, then scans the sorted records, accumulates the values and writes out an output record immediately after passing the last record of each group. Because of this last step, the overall execution reveals number of records in each group even if 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 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, outputting one record for every record it encounters in G. Some of these records are real output records, while other are dummies and therefore marked so that they can be removed later. Finally, it uses pCompact to remove the dummies. Because these 3 steps are privacy-preserving, so is pAggr (i.e., it does not reveals number of records in each group). The algorithm invokes the scrambler $\mathbb{S}$ twice. The overall running time is $O(n \log n)$ — having the same complexity as that of the data-oblivious alternative [8].

```
1: procedure pAggr($X$)
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
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 dummies from $V$
16:    return $Y$
17: end procedure
```

4.5 Join

The algorithm performs the inner join on two datasets X_1 and X_2. We consider the sort-merge join (generalizing to other join algorithms is straightforward), which 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 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.* [9]. The data-oblivious algorithm consists of two stages: the first stage computes the degree of each record in the join graph, and 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. In the nutshell, pJoin follows the workflow of Arasu *et al.* data-oblivious join algorithm [9], but improves its overhead by implementing the first stage using one pSort, two linear-scan and two pCompact steps (line 2-7), while reimplementing the data-oblivious expansion step without change for the second stage. (line 8-9).

```
 1: procedure pJoin($X_1, X_2$)
 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
```

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 using pSort, 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 traverses X in forward direction (i.e., from left to right), associating 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. Similarly, the second pass, RRSum(), assumes weight values of 0 for X_1 records and 1 for X_2 records, scans X backwards (i.e., from right to left) and associates with each record the running sum of weights in its group. At the end of this pass, X_1 records are associated with theirs degree in the join graph. After these two passes, pCompact is invoked twice to privately 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 OExpand() presented in [9] for this step. Finally, it performs a linear scan to stitch records together and generate the final output Y. Table 3 gives a detailed example for pJoinwith two input sequences $X_1 = \{\langle a, fde\rangle, \langle a, tol\rangle, \langle b, lxv\rangle, \langle b, xdj\rangle\}$ and $X_2 = \{\langle a, maj\rangle, \langle b, med\rangle, \langle c, tfn\rangle, \langle d, kbs\rangle\}$.

pJoin runs with the same complexity as the data-oblivious version does, i.e., $O(n \log n + l \log l)$ where l is the output size. Nevertheless, we show later in Section 5 that pJoin has a lower running time in practice, for pSort and pCompact are more efficient than the corresponding data-oblivious steps. Since each and every step in pJoin is privacy-preserving, following the composition property, pJoinis also privacy-preserving.

4.6 Supporting Spark Operations

Spark [1] is a general computing framework that has been widely adopted in various application domains including machine learning and data analysis [50]. Thus, by supporting Spark functions in StC, our solution enables developers to build complex privacy-preserving applications. While StC covers only a certain class of computations (e.g., those that are invariant to input permutation), its compatibility with Spark has proved that such a class is expressive enough to enable a wide range of privacy-preserving computations. We remark that most, if not all, of functions in Spark require accessing the entire dataset during their execution, rendering ORAM protocols' amortized multiplicative overhead significantly prohibitive when processing a large volume of data (e.g., translating to $10 - 100\times$ slowdown).

Table 3. 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
$\langle a, fde\rangle_{X_1}$	$\langle a, fde\rangle_{X_1}(1)$	$\langle d, kbs\rangle_{X_2}(1)$	$\langle b, xdj\rangle(1)$	$\langle a, maj\rangle(2)$	$\langle b, xdj\rangle$	$\langle b, med\rangle$	$\langle b, xdjmed\rangle$
$\langle a, tol\rangle_{X_1}$	$\langle a, tol\rangle_{X_1}(2)$	$\langle c, tfn\rangle_{X_2}(1)$					
$\langle a, maj\rangle_{X_2}$	$\langle a, maj\rangle_{X_2}(2)$	$\langle b, med\rangle_{X_2}(1)$	$\langle b, lxv\rangle(1)$	$\langle b, med\rangle(2)$	$\langle b, lxv\rangle$	$\langle b, med\rangle$	$\langle b, lxvmed\rangle$
$\langle b, lxv\rangle_{X_1}$	$\langle b, lxv\rangle_{X_1}(1)$	$\langle b, xdj\rangle_{X_1}(1)$					
$\langle b, xdj\rangle_{X_1}$	$\langle b, xdj\rangle_{X_1}(2)$	$\langle b, lxv\rangle_{X_1}(1)$	$\langle a, tol\rangle(1)$	$\langle c, tfn\rangle(0)$	$\langle a, tol\rangle$	$\langle a, maj\rangle$	$\langle a, tolmaj\rangle$
$\langle b, med\rangle_{X_2}$	$\langle b, med\rangle_{X_2}(2)$	$\langle a, maj\rangle_{X_2}(1)$					
$\langle c, tfn\rangle_{X_2}$	$\langle c, tfn\rangle_{X_2}(0)$	$\langle a, tol\rangle_{X_1}(1)$	$\langle a, fde\rangle(1)$	$\langle d, kbs\rangle(0)$	$\langle a, fde\rangle$	$\langle a, maj\rangle$	$\langle a, fdemaj\rangle$
$\langle d, kbs\rangle_{X_2}$	$\langle d, kbs\rangle_{X_2}(0)$	$\langle a, fde\rangle_{X_1}(1)$					

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	

We summarize our effort in Table 4. Some functions benefit immediately from StC, similar to pSort and pSelect, while other functions 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 these three 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.

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 [26], oblCompact for compaction [24], oblSelect for selection [24], oblAggr for aggregation [8] and oblJoin for join [9]. 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 [40]: 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 input size from 8GB to 64GB (i.e., 2^{26} to 2^{29} records). Our implementations 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. In order to simulate a trusted hardware (e.g., IBM 4767-002 PCIeCC2 [3]), we limit the CPU clock to 233MHz and use 64MB of RAM to represent its private memory (i.e., $m = 2^{19}$). We repeat each experiment 10 times and report the average results.

5.1 Cost of Security

Table 5 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

Table 5. 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×)
Aggregation	10593	24578 (2.32×)	63477 (5.99×)
Join	12400	59610 (4.81×)	105235 (8.49×)

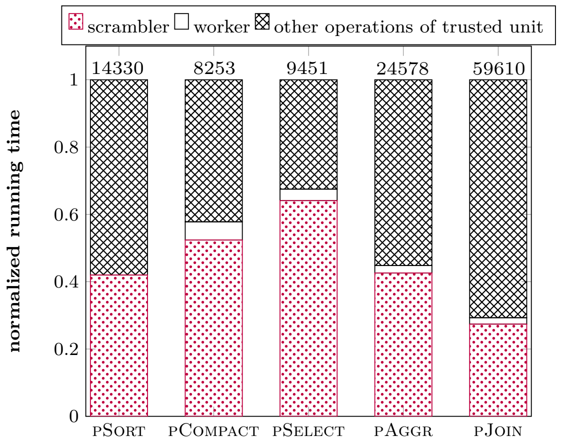


Fig. 3. Normalized running time breakdowns for StC's algorithms. The total running time (in seconds and displayed on top of each bar) comprises of the time taken by the scrambler, by the worker, by the pre-processing and post-processing steps, and by the main algorithm logic.

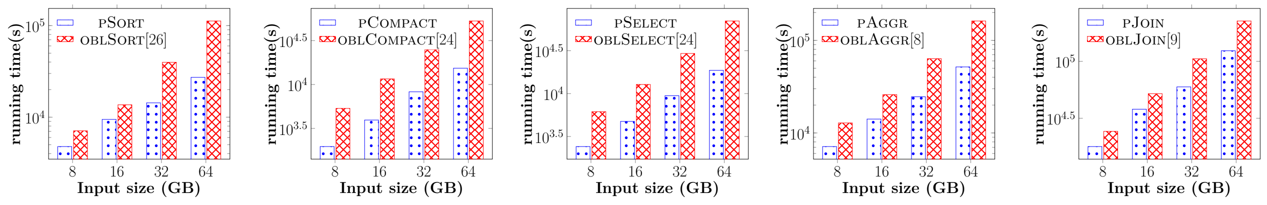
data-oblivious algorithms under consideration. Thus, to make the comparison fair, we ran StC algorithms on a single node. It can be seen that StC algorithms incur overheads between 1.79× to 4.91× over the baseline system. To better understand the factors contributing to the overheads, we measured the 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. Figure 3 depicts 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 are practical. Figure 4 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×, 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, which pJoin inherits directly from [9], 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.

5.3 I/O complexity of STC algorithms

Table 6 details the numbers of I/O and cryptographic operations required by StC algorithms and their data-oblivious counterpart. StC algorithms require $O(n)$ I/Os with a small constant factor, whereas all data-oblivious algorithms, except for oblSelect, 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.



(a) Sort **(b)** Compaction **(c)** Selection **(d)** Aggregation **(e)** Join

Fig. 4. Performance comparison between our algorithms and the corresponding data-oblivious alternatives. Running time (s) is shown in log-scale.

Table 6. 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	$(p_1 + p_2 + 5) \cdot n$	$O(n)$
oblSort[26]	$(\sum_{i=1}^{\log s} i + \log s + 1) \cdot n$	$O(n \log^2 n)$
pCompact	$(p_1 + p_2 + 2) \cdot n$	$O(n)$
oblCompact[24]	$(1 + \log n) \cdot n$	$O(n \log n)$
pSelect	$(p_1 + p_2 + 4) \cdot n$	$O(n)$
oblSelect[24]	$\frac{1}{2}(4 + \log n) \cdot n$	$O(n)$
pAggr	$(2p_1 + 2p_2 + 8) \cdot n$	$O(n)$
oblAggregate[8]	$(\sum_{i=1}^{\log s} i + \log s + \log n + 3) \cdot n$	$O(n \log^2 n)$
pJoin	$(3p_1 + 3p_2 + 9 + d) \cdot n$	$O(dn)$
oblJoin[9]	$(\sum_{i=1}^{\log s} i + \log s + 2\log n + 5 + d) \cdot n$	$O(n \log^2 n)$

5.4 Scalability

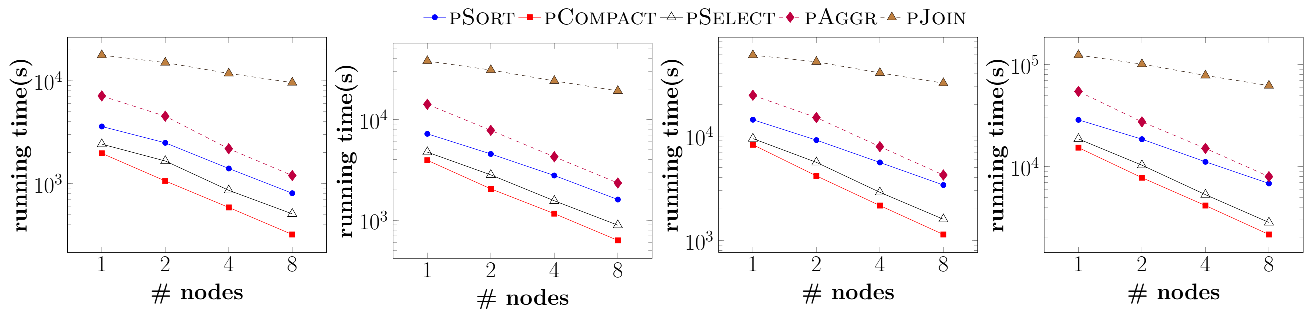
Figure 5 reports the running time of StC algorithms on multiple nodes. It demonstrates that StC can leverage resources in distributed environment to achieve significant speedups. In particular, increasing the number of nodes from one to eight results in $4\times$ speedup for sort and up to $7\times$ 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 4764 PCI-X[2] or Intel SGX [4] to enable secure computations, especially focusing on query processing. TrustedDB [10] presents a secure outsourced database prototype that leverages IBM 4764 secure CPU (SCPU) for privacy-preserving SQL queries. Cipherbase [8] extends TrustedDB's idea to offer a full-fledged SQL database system with data confidentiality. $VC3$ [44] 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 [20, 37] 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 that supports many other algorithms.

Ohrimenko *et al.* presented a system for oblivious multi-party machine learning, supporting various training and prediction methods [39]. StC, on the other hand, focuses on data management operations.



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

Fig. 5. StC algorithms performance on multiple nodes with different input sizes. Running time (s) is shown in log-scale.

Secure Computation by Data-Oblivious Technique. Oblivious-RAM [23] enables secure and oblivious computation by hiding data access patterns during program execution. ORAM techniques [14, 26, 42] 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.* proposed several data-oblivious algorithms [24–26] which we used for benchmarking StC. The authors also presented approaches to simulate ORAM using data-oblivious algorithms [26]. Other interesting data-oblivious algorithms have also been proposed for graph drawing [27] and graph-related computations such as maximum flow, minimum spanning tree, single-source single-destination (SSSD) shortest path, or breadth-first search [12]. However, these algorithms are application-specific and less efficient than StC algorithms.

Access Confidentiality via shuffling. The use of scrambling process in hiding data access patterns has been discussed in the literature [19, 42, 47, 48]. While these works are ideal for applications that make few accesses in a large dataset, they may not necessarily be so for other applications that potentially require accessing the entire dataset multiple times, for example data management tasks. For such applications, customized algorithms are likely to perform better (e.g., [37]). StC offers a simple way for implementing those 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.

Acknowledgements

We would like to thank the anonymous reviewers of the paper for their helpful feedback and insightful suggestions. This research has been supported by the National Research Foundation, Prime Minister's Office, Singapore under its National Cybersecurity R&D Program (Award No. NRF2015-NCR-NCR002-001) administered by the National Cybersecurity R&D Directorate, and its Competitive Research Programme (CRP Award No. NRF-CRP8-2011-08). All opinions and findings expressed in this work are those of the authors and do not necessarily reflect the views of any of the sponsors.

References

[1] *Apache Spark*. http://spark.apache.org/.

[2] *IBM 4764 PCI-X Cryptographic Coprocessor*. http://www-03.ibm.com/security/cryptocards/pcixcc/overview.shtml.

[3] *IBM PCIe Cryptographic Coprocessor Version 2 (PCIeCC2)*. http://www-03.ibm.com/security/cryptocards/pciecc2/overview.shtml.

[4] *Software Guard Extensions Programming Reference*. https://software.intel.com/sites/default/files/managed/48/88/329298-002.pdf.

[5] *WikiLeaks Publishes NSA Target List*. https://www.schneier.com/blog/archives/2016/03/wikileaks_publi.html.

[6] Ahmad, Muhammad Yousuf, and Kemme, Bettina 2015. *Compaction management in distributed key-value datastores*. In: *PVLDB*.

[7] Aiyer, Amitanand, Bautin, Mikhail, Chen, Guoqiang Jerry, Damania, Pritam, Khemani, Prakash, Muthukkaruppan, Kannan, Ranganathan, Karthik, Spiegelberg, Nicolas, Tang, Liyin, and Vaidya, Madhuwanti 2012. Storage infrastructure behind facebook messages using hbase at scale. *Data Engineering Bulletin*.

[8] Arasu, Arvind, Blanas, Spyros, Eguro, Ken, Kaushik, Raghav, Kossmann, Donald, Ramamurthy, Ravi, and Venkatesan, Ramaratnam 2013. *Orthogonal Security With Cipherbase*. In: *CIDR*.

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

[10] Bajaj, Sumeet, and Sion, Radu 2014. *TrustedDB: A Trusted Hardware-Based Database with Privacy and Data Confidentiality*. In: *TKDE*.

[11] Baumann, Andrew, Peinado, Marcus, and Hunt, Galen 2014. *Shielding applications from an untrusted cloud with haven*. In: *OSDI*.

[12] Blanton, Marina, Steele, Aaron, and Alisagari, Mehrdad 2013. *Data-oblivious graph algorithms for secure computation and outsourcing*. In: *ASIACCS*.

[13] Blum, Manuel, Floyd, Robert W, Pratt, Vaughan, Rivest, Ronald L, and Tarjan, Robert E 1973. Time bounds for selection. *Journal of computer and system sciences*.

[14] Boneh, Dan, Mazieres, David, and Popa, Raluca Ada 2011. Remote oblivious storage: Making oblivious RAM practical. *MIT-CSAIL-TR-2011-018*.

[15] Brakerski, Zvika, and Brakerski, Zvika 2011. *Efficient Fully Homomorphic Encryption from (Standard) LWE*. In: *FOCS*.

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

[17] Chen, Shuo, Wang, Rui, Wang, XiaoFeng, and Zhang, Kehuan 2010. *Side-Channel Leaks in Web Applications: A Reality Today, a Challenge Tomorrow*. In: *IEEE S&P (Oakland)*.

[18] Chen, Yao, and Sion, Radu 2012. On securing untrusted clouds with cryptography. *Data Engineering Bulletin*.

[19] di Vimercati, Sabrina De Capitani, Foresti, Sara, Paraboschi, Stefano, Pelosi, Gerardo, and Samarati, Pierangela 2013. *Distributed shuffling for preserving access confidentiality*. In: *ESORICS*.

[20] Dinh, Tien Tuan Anh, Saxena, Prateek, Chang, Ee-Chien, Ooi, Beng Chin, and Zhang, Chunwang 2015. *M^2R: Enabling Stronger Privacy in MapReduce Computation*. In: *USENIX Security*.

[21] ElGamal, Taher 1984. *A public key cryptosystem and a signature scheme based on discrete logarithms*. In: *CRYPTO*.

[22] Gentry, Craig, et al. 2009. *Fully homomorphic encryption using ideal lattices*. In: *STOC*.

[23] Goldreich, Oded, and Ostrovsky, Rafail 1996. Software protection and simulation on oblivious RAMs. *Journal of the ACM*.

[24] Goodrich, Michael T. 2011. *Data-oblivious External-memory Algorithms for the Compaction, Selection, and Sorting of Outsourced Data*. In: *SPAA*.

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

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

[27] Goodrich, Michael T, Ohrimenko, Olga, and Tamassia, Roberto 2012. Data-oblivious graph drawing model and algorithms. *arXiv preprint arXiv:1209.0756*.

[28] Halevy, Alon, Rajaraman, Anand, and Ordille, Joann 2006. *Data Integration: The Teenage Years*. In: *VLDB*.

[29] Katz, Jonathan, and Lindell, Yehuda 2014. *Introduction to modern cryptography*. CRC Press.

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

[31] Knuth, Donald Ervin 1998. *The art of computer programming: sorting and searching*, Vol. 3. Pearson Education.

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

[33] Li, Feifei, Hadjieleftheriou, Marios, Kollios, George, and Reyzin, Leonid 2006. *Dynamic authenticated index structure for outsourced databases*. In: *ACM SIGMOD*.

[34] McCun, Jonathan M., Parno, Bryan, Perrig, Adrian, Reiter, Michael K., and Isozaki, Hiroshi 2008. *Flicker: An Execution Infrastructure for TCB Minimization*. In: *EuroSys*.

[35] McCune, J.M., Li, Y., Qu, N., Zhou, Z., Datta, A., Gligor, V., and Perrig, A. 2010. *TrustVisor: Efficient TCB Reduction and Attestation*. In: *IEEE S&P (Oakland)*.

[36] Meng, Xiangrui, Bradley, Joseph, Yuvaz, B, Sparks, Evan, Venkataraman, Shivaram, Liu, Davies, Freeman, Jeremy, Tsai, D, Amde, Manish, Owen, Sean, et al. 2016. Mllib: Machine learning in apache spark. *JMLR*.

[37] Ohrimenko, Olga, Costa, Manuel, Fournet, Cedric, Gkantsidis, Christos, Kohlweiss, Markulf, and Sharma, Divya 2015. *Observing and Preventing Leakage in MapReduce*. In: *CCS*.

[38] Ohrimenko, Olga, Goodrich, Michael T, Tamassia, Roberto, and Upfal, Eli 2014. *The Melbourne shuffle: Improving oblivious storage in the cloud*. In: *ICALP*.

[39] Ohrimenko, Olga, Schuster, Felix, Fournet, Cédric, Mehta, Aastha, Nowozin, Sebastian, Vaswani, Kapil, and Costa, Manuel 2016. *Oblivious Multi-Party Machine Learning on Trusted Processors*. In: *USENIX Security*.

[40] O'Malley, Owen, and Murthy, Arun C 2009. *Winning a 60 second dash with a yellow elephant*. Tech. rep., Yahoo.

[41] Paillier, Pascal 1999. *Public-key cryptosystems based on composite degree residuosity classes*. In: *EUROCRYPT*.

[42] Pinkas, Benny, and Reinman, Tzachy 2010. *Oblivious RAM revisited*. In: *CRYPTO*.

[43] Popa, Raluca Ada, Redfield, Catherine, Zeldovich, Nickolai, and Balakrishnan, Hari 2011. *CryptDB: Protecting confidentiality with encrypted query processing*. In: *SOSP*.

[44] Schuster, Felix, Costa, Manuel, Fournet, Cédric, Gkantsidis, Christos, Peinado, Marcus, Mainar-Ruiz, Gloria, and Russinovich, Mark 2014. *VC3: Trustworthy DATA analytics in the cloud*. In: *IEEE S&P (Oakland)*.

[45] Stefanov, Emil, Van Dijk, Marten, Shi, Elaine, Fletcher, Christopher, Ren, Ling, Yu, Xiangyao, and Devadas, Srinivas 2013. *Path ORAM: An extremely simple oblivious RAM protocol*. In: *CCS*.

[46] Tu, Stephen, Kaashoek, M Frans, Madden, Samuel, and Zeldovich, Nickolai 2013. *Processing analytical queries over encrypted data*. In: *PVLDB*.

[47] Vimercati, Sabrina De Capitani Di, Foresti, Sara, Paraboschi, Stefano, Pelosi, Gerardo, and Samarati, Pierangela 2015. Shuffle index: efficient and private access to outsourced data. *TOS*.

[48] Wang, Shuhong, Ding, Xuhua, Deng, Robert H, and Bao, Feng 2006. Private information retrieval using trusted hardware. In: *ESORICS*.

[49] Xin, Reynold S, Gonzalez, Joseph E, Franklin, Michael J, and Stoica, Ion 2013. *Graphx: A resilient distributed graph system on spark*. In: *GRADES*.

[50] Zaharia, Matei, Chowdhury, Mosharaf, Franklin, Michael J, Shenker, Scott, and Stoica, Ion 2010. Spark: Cluster computing with working sets. *HotCloud*.