PermuteRam: Optimizing Oblivious Computation for Efficiency

Shruti Tople, Hung Dang, Prateek Saxena and Ee-Chien Chang

National University of Singapore
{shruti90, hungdang, prateeks, changec} @comp.nus.edu.sg

Abstract. Privacy preserving computation is gaining importance. Along with secure computation guarantees, it is essential to hide information leakage through access patterns. Input-oblivious execution is a security property that is crucial to guarantee complete privacy preserving computation. In this work, we present an algorithm-specific approach to achieve input-oblivious execution. We call this class of algorithms PermuteRam. PermuteRam algorithms satisfy a specific patterns in their execution profile called Perpat— patterns that can be realized using permutation as a primitive. Next, we claim that algorithms having Perpat pattern execute in an input-oblivious manner. Further, we show that PermuteRam is expressive and includes various categories of algorithms like sorting, clustering, operating on tree data structures and so on. PermuteRam algorithms incur only an additive overhead of $O(N)$ and a private storage of $O(\sqrt{N})$. Hence, PermuteRam algorithms demonstrate optimal performance for linear or super-linear complexities.

1 Introduction

Privacy preserving computation on sensitive data is gaining importance. Several secure computation and isolated execution techniques have emerged that support operating on encrypted data [11, 1]. However, it is well-known that only encryption of data is not sufficient to guarantee privacy. Observing address access patterns while executing on encrypted data can leak sensitive information such as secret keys [18]. Therefore, it is crucial to prevent information leakage through address access patterns in order to guarantee complete privacy preserving computation.

One approach to prevent leakage through address access patterns is to use Oblivious RAM (ORAM). Goldreich and Ostrovsky [6] proposed ORAM to hide memory access patterns by shuffling data in memory. ORAM is a generic approach and is applicable to any algorithm irrespective of its structure. It hides the actual memory address by replacing a single access with *polylog N* accesses. Although being algorithm-agnostic, it incurs a multiplicative overhead (polylog N) to the algorithm complexities. Despite of the tremendous improvements in this area, the multiplicative factor causes a performance slowdown in many applications [14, 16, 9, 15].

A second line of research is to specifically re-design existing algorithms to not leak information about the input during execution. Such algorithms are called as input-oblivious algorithms. Blanton et al. provides input-oblivious algorithms for "dense" graphs such as SSSD, minimum spanning tree and maximum flow to hide the graph structure [2]. More recently M^2R system allows oblivious computation for map-reduce framework by leveraging on its algorithmic structure of write followed by read operations [5]. Previous work shows that some algorithms exhibit specific patterns that allows the use of simple techniques like permutation to make their execution input-oblivious. In addition, leveraging on the algorithm structure provides better performance, almost equal to the complexity of original algorithm in most cases [2].

The key question is whether their exists any general pattern that allows designing of such input-oblivious algorithms? In this paper, we take a step ahead to identify and generalize such patterns that guarantee input-obliviousness in various algorithms. Such a generalization is useful in optimizing the performance of various privacy preserving computations that are currently not feasible with generic ORAM techniques. With the knowledge of these patterns, developers can write efficient and practical input-oblivious algorithms using simple techniques. This is a rational objective as the developers are aware of the algorithm structure and the ways to modify them while retaining the functionality.

Prior work has used existence of specific patterns in algorithms to prevent information leakage through address access profiling. In this paper, as our first contribution, we identify such patterns and generalize them into a class called Perpat patterns. These patterns are classified based on the read-write profiles and the address access profiles of an algorithm. Permutation is an important primitive to achieve these patterns in most algorithms, hence the name Perpat. We refer to the class of algorithms that satisfy these patterns as PermuteRam algorithms.

Next, we present a formal definition for our security property of input-oblivious execution in presence of an "honest-but-curious" adversary. To state informally, input-oblivious execution prevents information leakage about the input by observing the execution of an algorithm. As our second contribution, we claim that all PermuteRam algorithms having Perpat pattern guarantee input-oblivious execution in our threat model. Further, as a part of security analysis, we provide proofs to support the above claim.

Lastly, we investigate the expressiveness of PermuteRam algorithms in the presence of a moderate size private but trusted storage and a large public untrusted storage. We show that a broad class of applications such as operations on data structures, sorting algorithms, clustering algorithms fall under PermuteRam. Further exploration might result in even larger class of applications falling in this category. However, we present this paper with the hope that future work on optimizing input-oblivious algorithms will benefit from this work.

Results. We demonstrate the expressiveness of PermuteRam algorithms by manually transforming various classes of algorithms to PermuteRam. We show that most of them demonstrate a performance improvement as compared to the

best known ORAM techniques. It includes several sorting algorithms, clustering algorithms and operations on tree structures. We compare the complexities of PermuteRam algorithms to the best known ORAM scheme by Kushilevitz et al. [9]. PermuteRam algorithms perform better in case studies that have more than linear complexity, in most cases retaining the original complexity. This is due to the fact that PermuteRam algorithms have an additive overhead of $O(N)$ for permutation operation where N is the size of input data whereas all ORAM schemes have a multiplicative overhead. PermuteRam algorithms requires $O(\sqrt{N})$ private storage which is practical in scenarios such as cloud storage or secure processors.

Contributions. The contributions as follows :

- Perpat **pattern**: We present a class of pattern called Perpat to hide information leakage through access patterns. Perpat patterns can be realized using a simple technique of permutation.
- **Security Property**: We define the security property of input-oblivious execution and prove that any algorithm with Perpat pattern is input-oblivious.
- **Expressiveness**: Perpat patterns are applicable to a variety of algorithms such as sorting, clustering and operations on tree structures. We refer to them as PermuteRam algorithms.
- **Complexity Analysis** : The algorithm complexities of PermuteRam algorithms is close to the original algorithms. PermuteRam algorithms perform better as compared to the best known ORAM techniques.

2 Overview

Our aim is to generalize patterns in various algorithms such that their presence guarantees the security property of input-obliviousness when executed in real world setting where the attacker is usually a passive observer.

2.1 Problem Setting.

We envision a setting with some moderate private storage and a huge amount of public but untrusted storage. The untrusted public region stores the data in encrypted form. The private storage is trusted and isolated from the public region. Any operations or data accesses performed in the private memory are inaccessible to the adversary. The data is decrypted only in the private memory in order to compute on it. This setting is practical and observed in many real world scenarios such as in cloud storage where the memory on client side can be considered private and the storage space at the cloud provider is huge but untrusted. Similarly, enclaved execution environments such as SGX or Overshadow suit this setting where a process has its own isolated and protected address space as the private memory and the disk storage as the public memory [11, 4]. Finally, a more granular setting can be observed in secure processors where cache is private and RAM memory is public.

Scope and Assumptions. In this setting, our main goal is to hide information leakage in various algorithms due to address access profiling. Leakage through other channels like timing are not in the scope of this paper [3]. However, alternate solutions to prevent information leakage through these side channels can be used simultaneously [8]. We assume the encryption scheme used to protect the confidentiality of sensitive data in the public region is semantically secure. The security of the permutation operation which is used as a primitive in PermuteRam is assumed. Further, the implementation of private storage is assumed to be secure and tamper-proof. The adversary does not have access to any secret keys or address patterns in the private storage.

2.2 Attacker Model

In the above setting, an attacker is anybody that can observe the access patterns on encrypted data, for example, a curious administrator at the cloud provider or a compromised / untrusted operating system. In our threat model, the adversary is "honest-but-curious" i.e., it can passively observes the algorithm execution profile but cannot tamper or alter the execution process. An algorithm during its execution performs several read / write operations accessing various addresses in the memory. A passive adversary can trace memory access patterns in the public region and may even log them for future analysis. Such an adversary can observe the execution profile of an algorithm with the intention to infer information about the inputs. Thus, the attacker's knowledge set consists of the algorithm execution profile as defined below.

Definition 1. *(***Execution Profile***). The execution profile* $\boldsymbol{P}$ *of an algorithm* $\mathcal{AG}$ *is a vector* $\boldsymbol{P}(\mathcal{AG}) = (P_1, P_2, P_3...P_n)$ *where each* P_i *is denoted as a tuple:*

$$P_i :=< Operation, Address, Value > \tag{1}$$

The `Operation` parameter in P is either a read (R) or a write (W) operation executed by the algorithm on the untrusted memory. `Address` parameter denotes the location (Loc) accessed by the corresponding read / write operation in public memory. `Value` is any ciphertext which is read / written from / to the public memory. We assume that `Value` parameter is always encrypted with a semantically secure encryption scheme and is therefore secure. Since our focus in this paper is to hide information leakage through address access pattern, henceforth, we consider only operation and address parameters in the execution profile i.e, $P = (RW, Loc)$. Further, we divide this $\boldsymbol{P}$ into two separate profiles namely read-write profile ($\boldsymbol{RWP}$) and address access profile ($\boldsymbol{LocP}$) which is used later for explanation of the different Perpat patterns. Note that, the adversary can distinguish between a read and write operation unlike previous work [10]. Roughly speaking, $\boldsymbol{RWP} = (RW_1, RW_2, ...RW_n)$ contains the sequence of read-write operations that an algorithm performs and $\boldsymbol{LocP} = (Loc_1, Loc_2...Loc_n)$ is the sequence of addresses accessed during the execution of the algorithm.

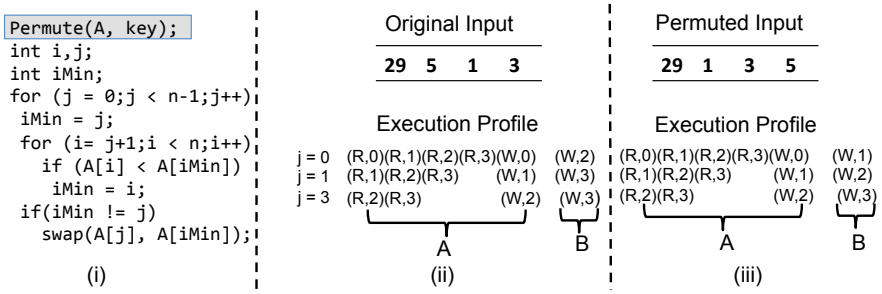

Fig. 1. Selection sort and its execution profiles

2.3 Our Approach : Running Example

Let us consider an example of selection sort algorithm. It sorts the input array by finding the minimum element and placing it in the correct position in the output array. Figure 1(i) shows the pseudo code for selection sort algorithm. It also shows the execution profile $\boldsymbol{P}$ of selection sort for an input array (ii). In such sorting algorithms, our goal is to hide the ordering of elements in the original input array. For example, the position of the largest or smallest element in the input. With this example, we generate an insight for PERPAT patterns and how their existence guarantees input-obliviousness.

Analysis. The execution profile for selection sort is divided into two sub-profiles — A and B. For a fixed length input (n = 4 in Figure), profile A has a deterministic pattern while profile B differs based on the input ordering. This profile reflects the arrangement of the input values thereby leaking information about their positions. Careful readers may notice that we can protect this information leakage if the execution profile in part B is randomized (or unlinked) from the original input. The key insight is to make the execution profile in part B access random locations on every execution of this algorithm. Since the adversary observes random locations being accessed, it learns nothing about the original input.

Transformation. To introduce randomness, we insert a shuffling or permutation function before the start of the algorithm (line 1 in Figure 1). Now the profile in part B is dependent on the permuted input rather than the original input (See Figure 1(iii)). If the permutation function is perfectly pseudo-random than the attacker cannot infer information about original input by observing the execution profile of the permuted input. Selection sort is a simple case where inserting a single permutation operation is sufficient to randomize the access profile. This is due to the algorithm characteristics where the execution profile in one part does not effect the profile in other part. However, this is not true for all cases. Several algorithms require addition of intermediate permutation step to completely hide the information leakage through address accesses. The permutation step guarantees that the address accessed in each part is equally likely and is not influenced by other read / write operations in the algorithm. With

this observation we define a class of PERPAT patterns in Section 3 that prevents information leakage through execution profile. We present these patterns with the intention that developers can generate input-oblivious algorithms by transforming algorithms to satisfy PERPAT pattern. As this transformation requires manual analysis and understanding of the algorithms, we expect the developer of an algorithm to be the appropriate candidate for this task.

Permutation. Note that, the permutation step introduced in the above transformation also needs to be performed obliviously. For performing oblivious permutation, we use the Melbourne shuffle which requires $O(\sqrt{N})$ private memory that satisfies our requirement of moderate size private storage, message size of $O(\sqrt{N})$ and $O(N)$ I/O operations [13]. This is a oblivious shuffling algorithm which is practical and efficient.

2.4 Problem Definition

In this work, we advocate the idea that algorithms having execution profiles with PERPAT patterns guarantee the security propery of input-obliviousness.

Security Property. We define input-obliviousness as a property that guarantees computational indistinguishability of inputs to an algorithm in the presence of an adversary with access to the execution profile $\boldsymbol{P}$. We model the definition as a game between an adversary and a challenger. The adversary provides two encrypted inputs $\boldsymbol{I_0}$ and $\boldsymbol{I_1}$ to the challenger. The challenger randomly selects one of the inputs $\boldsymbol{I_b}$ where $b \in \{0, 1\}$ and executes algorithm $\mathcal{AG}$ over the selected input. The challenger further sends the execution profile $\boldsymbol{P_b(AG)}$ to the adversary.

Definition 2. *(***Input-obliviousness).** *Given a computationally bounded adversary $\mathcal{A}$ with access to execution profile $\boldsymbol{P_b(AG)}$ where $b \in \{0, 1\}$ and encrypted inputs $\boldsymbol{I_0}$ and $\boldsymbol{I_1}$ having the same length i.e, $|\boldsymbol{I_0}| = |\boldsymbol{I_1}|$, algorithm $\mathcal{AG}$ is input-oblivious only if the advantage of p.p.t $\mathcal{A}$ in distinguishing $\boldsymbol{I_0}$ and $\boldsymbol{I_1}$ is negligible by learning $\boldsymbol{P_b(AG)}$ i.e,*

$$Adv(A) := |Pr[A(\boldsymbol{I_0}) = 1] - Pr[A(\boldsymbol{I_1}) = 1]| \leq \epsilon \tag{2}$$

where ϵ is negligible.

Section 4 provides a security proof that any algorithm having PERPAT pattern is input-oblivious as per above definition. The example of selection sort algorithm explains how developers can manually analyze algorithmic structures and transform them to satisfy PERPAT patterns. The transformed algorithms are input-oblivious if they satisfy PERPAT patterns in their execution profile i.e, $\boldsymbol{P}(\mathcal{AG}) \in \{\text{PERPAT }\}$.

3 PERPAT Pattern.

On the basis of whether the read-write ($\boldsymbol{RWP}$) and address access profile($\boldsymbol{LocP}$) of a given algorithm show deterministic or randomized behaviour, we classify the PERPAT pattern into four main types that guarantee input-obliviousness.

1. `deterRWP-and-deterLocP` **(PDD).** The first pattern is where both the read-write ($\boldsymbol{RWP}$) and address ($\boldsymbol{LocP}$) profiles of an $\boldsymbol{AG}$ show a deterministic behavior. This is the most simplest pattern that provides input-obliviousness. This pattern is observed and used in many previous works that aim to hide information leakage through address accesses. We say that an algorithm exhibits (PDD) pattern if given two inputs of the same length, the execution profile of the algorithm for the two inputs can be divided into exactly same ($\boldsymbol{RWP}$) and ($\boldsymbol{LocP}$). Intuitively, this indicates that the execution profile is not a function of the input values and hence leaks nothing about the original inputs.

Examples of algorithms showing this pattern include finding minimum or maximum value in a given input array, matrix multiplication, Rabin-Karp substring match and so on.

Definition 3. *(*`deterRWP-and-deterLocP`*.) If* $\forall \boldsymbol{I}_1, \boldsymbol{I}_2 \in \boldsymbol{I}$ *where* $|\boldsymbol{I}_1| = |\boldsymbol{I}_2|$ *and* $\boldsymbol{P}(\mathcal{AG}_{\boldsymbol{I}_1})$,$\boldsymbol{P}(\mathcal{AG}_{\boldsymbol{I}_2})$ *are the execution profile of algorithm* $\mathcal{AG}$ *over inputs* $\boldsymbol{I}_1, \boldsymbol{I}_2$ *respectively, then algorithm* $\mathcal{AG}$ *exhibits* `deterRWP-and-deterLocP` *pattern only if* $\boldsymbol{P}(\mathcal{AG}_{\boldsymbol{I}_1}) \equiv \boldsymbol{P}(\mathcal{AG}_{\boldsymbol{I}_2})$ *i.e,* $\boldsymbol{RWP}_{\boldsymbol{I}_1} \equiv \boldsymbol{RWP}_{\boldsymbol{I}_2}$ *and* $\boldsymbol{LocP}_{\boldsymbol{I}_1} \equiv \boldsymbol{LocP}_{\boldsymbol{I}_2}$.

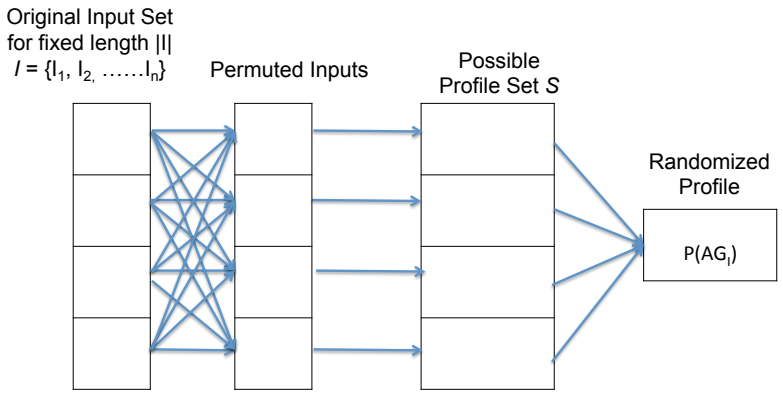

Fig. 2. Mapping of input to permuted values and their execution profiles

2. `deterRWP-and-randLocP` **(PDR).** The second pattern which guarantees input-obliviousness is called `deterRWP-and-randLocP`. In this pattern, similar to PDD, the $\boldsymbol{RWP}$ is deterministic across all inputs of the same length. However, the address corresponding to the read-write operations i.e, the $\boldsymbol{LocP}$ profile is always randomized.

By randomized profile, we mean that $\boldsymbol{LocP}$ is not dependent on the original input $\boldsymbol{I}$ and has equal probability of occurring as any one of the possible $\boldsymbol{LocP}$ profiles for input of a given length. Figure 2 pictorially explains what it means for a profile to show randomized behaviour. Let $\mathcal{I} = \boldsymbol{I_1}, \boldsymbol{I_2}....\boldsymbol{I_n}$ be the set of all

the possible original inputs of fixed length l to an algorithm $\mathcal{AG}$. Let $\mathcal{S}$ be the set of all possible profiles $P(\mathcal{AG})$ that an algorithm can generate given inputs $\boldsymbol{I} \in \mathcal{I}$.

Definition 4. *(***Randomized Profile.***) A profile is randomized if the probability of occurence of that profile after executing algorithm $\mathcal{AG}$ on any input $\boldsymbol{I}_i \in \mathcal{I}$ is equally likely from the profile set $\mathcal{S}$ i.e,*

$$P[\boldsymbol{P}(\mathcal{AG}_{\boldsymbol{I_i}})] = \frac{\mathbf{1}}{|\mathcal{S}|} \tag{3}$$

Definition 5. *(*`deterRWP-and-randLocP`*.) If $\forall \boldsymbol{I}_1, \boldsymbol{I}_2 \in \boldsymbol{I}$ where $|\boldsymbol{I}_1| = |\boldsymbol{I}_2|$ and $\boldsymbol{P}(\mathcal{AG}_{\boldsymbol{I_1}})$,$\boldsymbol{P}(\mathcal{AG}_{\boldsymbol{I_2}})$ are the execution profile of algorithm $\mathcal{AG}$ over inputs $\boldsymbol{I}_1, \boldsymbol{I}_2$ respectively, then algorithm $\mathcal{AG}$ exhibits `deterRWP-and-randLocP` pattern only if $\boldsymbol{RWP}_{\boldsymbol{I_1}} \equiv \boldsymbol{RWP}_{\boldsymbol{I_2}}$ and $\boldsymbol{LocP}_{\boldsymbol{I_1}}, \boldsymbol{LocP}_{\boldsymbol{I_2}}$ are randomized i.e, the probability of their occurence is equally likely.*

Examples of algorithms were this pattern appears is sorting algorithm such as bubble sort, selection sort, merge sort, heapsort and so on. Here the algorithms have a deterministic $\boldsymbol{RWP}$ mainly reading an element and followed by writing it to its appropriate location. However, the address from where the element is read and written is randomized.

3. `randRWP-and-randLocP` **(PRR).** The last profile pattern whose presence provides input-obliviousness is `randRWP-and-randLocP`. It includes both randomized $\boldsymbol{RWP}$ and $\boldsymbol{LocP}$ in the algorithm execution profile for every input of the same length. In addition to $\boldsymbol{LocP}$ in PDR, in PRR the probability that $\boldsymbol{RWP}$ belongs to one of the original inputs is equally likely. Quicksort algorithm falls in this category.

Definition 6. *(*`randRWP-and-randLocP`*.) If $\forall \boldsymbol{I}_1, \boldsymbol{I}_2 \in \boldsymbol{I}$ where $|\boldsymbol{I}_1| = |\boldsymbol{I}_2|$ and $\boldsymbol{P}(\mathcal{AG}_{\boldsymbol{I_1}})$,$\boldsymbol{P}(\mathcal{AG}_{\boldsymbol{I_2}})$ are the execution profile of algorithm $\mathcal{AG}$ over inputs $\boldsymbol{I}_1, \boldsymbol{I}_2$ respectively, then algorithm $\mathcal{AG}$ exhibits `randRWP-and-randLocP` pattern only if both $\boldsymbol{RWP}_{\boldsymbol{I_1}}, \boldsymbol{RWP}_{\boldsymbol{I_2}}$ and $\boldsymbol{LocP}_{\boldsymbol{I_1}}, \boldsymbol{LocP}_{\boldsymbol{I_2}}$ are randomized i.e, the probability of their occurence is equally likely.*

4. Composite pattern. It is possible to find combination of one or more of the above patterns in certain applications. If the execution of an algorithm can be partitioned such that each of the partition satisfies the requirements for either one of the above patterns then we say that the algorithm demonstrates a composite pattern. If the execution profile of an algorithm satisfies composite pattern then that algorithm guarantees the property of input-obliviousness.

Definition 7. *(***Composite pattern.***) An algorithm $\mathcal{AG}$ exhibits composite pattern if its execution profile $\boldsymbol{P}(AG)$ can be efficiently divided into n sub-profiles such that $P(AG) = \bigcup_{i=1}^{n} p_i$ and each sub-profile $p_i \in \{PDD, PDR, PRR\}$.*

Insertion sort algorithm exhibits composite pattern. In insertion sort, the read operation is performed always at a fixed address but the address of write operations is random depending on the input. In our case studies we manually transform clustering algorithms such as K means clustering and hierarchical clustering algorithms such that the transformation of these algorithms $\mathcal{AG}$ demonstrate composite pattern.

4 Security Guarantees

In this section, we guarantee that any program that exhibits one of the PERPAT pattern guarantees the security property of input-obliviousness as described in Section 2.4. The definition is based on computational indistinguishability of inputs in the presence of adversary with access to execution profile of the algorithm. Our security guarantees rely on the following assumptions about the underlying cryptographic schemes used in our solution.

- The encryption scheme used to provide confidentiality of data is semantically secure.
- The permutation scheme (Melbourne Shuffle [13]) is secure i.e, an adversary cannot construct the original input array given a permutation of the array.

Lemma 1. *An algorithm having execution profile with* `deterRWP-and-deterLocP` *pattern is input-oblivious.*

Proof. For execution profiles of algorithm having `deterRWP-and-deterLocP` pattern, the read-write profile and address profile are always deterministic for input of fixed length as per Definition 3. It is straightforward to calculate that the probability of execution profile $\boldsymbol{P}(\mathcal{AG})$ being generated from either input $\boldsymbol{I}_0$ or $\boldsymbol{I}_1$ is exactly the same i.e, $1/2$. Hence, simply observing the execution profile does not give any significant advantage to an adversary in distinguishing between two inputs to the algorithm. This ensures computational indistinguishability of inputs and thereby guaranteeing our security property of input-obliviousness.

Lemma 2. *An algorithm having execution profile with* `deterRWP-and-randLocP` *pattern is input-oblivious.*

Proof. For algorithms that demonstrate `deterRWP-and-randLocP` pattern in their execution profiles, we know by Definition 5 that the $\boldsymbol{RWP}$ are always deterministic i.e, they are exactly the same for any input of fixed length supplied to the algorithm. Thus, the adversary has no additional advantage in distinguishing the inputs by observing the $\boldsymbol{RWP}$ profile. Now let us consider what benefit the adversary gets from the address profile. The address profile $\boldsymbol{LocP}$ is randomized in PDR pattern. According to the definition of randomized profile (Definition 4), the probability $\boldsymbol{LocP}$ is equally likely to be generated from the possible set of address profiles for every run of the algorithm. Thus, the advantage of an adversary in guessing the input is equivalent to making a random

guess from the possible input space. Therefore, given two inputs $\boldsymbol{I_0}$ or $\boldsymbol{I_1}$ a p.p.t adversary cannot distinguish between them with more than negligible advantage even by observing the ***RWP*** profile and ***LocP*** profile. Note that, PERPAT pattern guarantees input-obliviousness as long as the underlying permutation scheme is secure. By observing the execution profile, the adversary guesses the input to be a random value from the possible input space. The security guarantee relies on the assumption that the adversary cannot reverse the permuted input to original input.

Lemma 3. *An algorithm having execution profile with* `randRWP-and-randLocP` *pattern is input-oblivious.*

Proof. In `randRWP-and-randLocP` pattern, both the ***RWP*** and ***LocP*** have randomized behaviour. Similar to `deterRWP-and-randLocP` pattern, the adversary can guess the input to be a random value from the input space depending on the profiles generated during that run of the algorithm. Hence, for two original inputs $\boldsymbol{I_0}$ or $\boldsymbol{I_1}$, the probability of the a specific profile being generated during the execution of an algorithm exhibiting `randRWP-and-randLocP` pattern is equally likely for both the inputs by Definition 6. Thus, the advantage of a p.p.t adversary in distinguishing between the two inputs based on their execution profiles is negligible.

Theorem 1. *An algorithm having execution profile with composite pattern is input-oblivious.*

Proof. An algorithm with composite pattern has its execution profile composed of sub-profiles where each profiles has either PDD, PDR or PRR patterns. Lemma 1, 2 and 3 guarantee that each of these sub-profiles is input-oblivious. We have to prove that the composition of these sub-profiles is also input-oblivious. Let us consider that an algorithm has 2 sub-profiles with a `deterRWP-and-deterLocP` pattern followed by a `deterRWP-and-randLocP` pattern. At the end of first profile, the adversary's advantage in distinguishing between two inputs $\boldsymbol{I_0}$ or $\boldsymbol{I_1}$ is negligible (by Lemma 1). The input to the PDR pattern is the output of the PDD pattern. However, as the PDR sub-profile is input-oblivious by itself, an adversary observing this sub-profile cannot distinguish whether the input to this sub-profile was generated from either $\boldsymbol{I_0}$ or $\boldsymbol{I_1}$. Therefore, the advantage of an adversary in distinguishing between two inputs given to an algorithm that contains composition of two sub-profiles is negligible. Similar argument holds for a profile consisting of n sub-profiles. Thus, an algorithm with n sub-profiles is input-oblivious as long as each sub-profile is itself input-oblivious i.e, belongs to either PDD, PDR or PRR patterns. Such an algorithm is said to exhibit composite pattern according to Definition 7. Hence, we prove that an algorithm having execution profile with composite pattern is input-oblivious.

5 Case Studies

We present the expressiveness of PermuteRam by elaborating on various categories of algorithms that satisfy Perpat patterns and hence perform input-oblivious execution.

5.1 Sorting Algorithms

Sorting algorithms especially comparison-based sorting operate on a given input array and produce sorted output. An adversary can learn about the relation of the elements in the original array by observing the execution profile of these algorithms such as the position of the smallest / largest element in the array. We discuss the method to transform these algorithm to PermuteRam and the type of Perpat patterns exhibited by their execution profiles.

Quicksort. Quicksort algorithm divides an input array into smaller sub-arrays and sorts the smaller arrays. It choses a pivot element in each iteration to partition the array. To satisfy Perpat pattern, we insert a permute function at the beginning of the algorithm. The algorithm in the Figure 3 shows `randRWP-and-randLocP` pattern. Due to the permutation, the pivot element is chosen randomly in each iteration. Hence, the sequence of read and writes varies across different inputs of the same size. However, since the permuted value cannot be converted to original array, the access pattern on the permuted input does not leak any information about the original input.

```
1  // Insert permute operation on the input array
2  Permute(A, K);
3  quicksort(A, lo, hi)
4    if lo < hi
5      p = partition(A, lo, hi)
6      quicksort(A, lo, p - 1)
7      quicksort(A, p + 1, hi)
8
9  partition(A, lo, hi)
10 // A random pivot element is selected in every
11 // execution of quicksort due to permutation step
12     pivot = A[hi]
13     i = lo //place for swapping
14     for j = lo to hi - 1
15         if A[j] <= pivot
16             swap A[i] with A[j]
17             i = i + 1
18     swap A[i] with A[hi]
19     return i
```

Fig. 3. Quicksort algorithm with `randRWP-and-randLocP` pattern

Heapsort. Heapsort algorithm involves building a heap data structure from a given input array and then performing sort operation on it. The heapsort function follows `deterRWP-and-deterLocP` pattern thus making it oblivious. However, building the heap data structure from a given array input is what leaks information. By performing a permute operation on the input array, we prevent the information leakage from building a heap step. Permuting the input array and then building the heap follows the `randRWP-and-randLocP` pattern. Figure 4 shows the algorithm for heapsort that performs input-oblivious execution.

```
1   Permute(A. key)
2   Heapsort(A)
3     Build_Max_Heap(A)
4     for i = A.length downto 2
5       exchange A[1] with A[i]
6       A.size = A.size - 1
7       Max_heapify(A, 1)
8
9   Build_Max_Heap(A)
10    A.size = A.length
11  for i = [A.length/2] downto 1
12    Max_heapify(A, i)
```

```
1   Max_heapify(A, i)
2     l = LEFT(i)
3     r = RIGHT(i)
4     if l <= A.size and A[l] > A[i]
5   largest = l
6     else
7   largest = i
8     if r <= A.size && A[r]>A[largest]
9   largest = r
10    if largest != i
11  exchange A[i] with A[largest]
12  Max_heapify(A, largest)
```

Fig. 4. Psuedo code for Heapsort and Building a Maximum Heap with `deterRWP-and-deterLocP` pattern. The Max_heapify function shows a random patten due to the permutation operation on the input array.

Mergesort. Mergesort algorithm divides the input array into small arrays of single element and then merges the smaller arrays. The order in which the merge step operates leaks information about the original array. Permuting the original array and then performing merge operation on it randomizes the execution profile. Such a merge-sort algorithm with permutation function displays the pattern `deterRWP-and-randLocP`. Hence, for any permuted input the address profile of mergesort is different and indistinguishable from the original input.

Bubblesort / Insertion sort. Bubblesort and Insertion sort also fall in the category where the algorithm operates on the complete input array and the output is an sorted array. Modifying these algorithms and inserting a permutation function at the beginning transforms them such that the execution profile satisfies is a combination of patterns, thus exhibiting composite pattern. Selection sort as shown in the running example follows the `deterRWP-and-randLocP` pattern and thus can be executed with PermuteRam.

5.2 Clustering Algorithms

In clustering algorithms, various input elements are grouped together into several clusters. Normal execution of these algorithm leaks the elements that are grouped together to form a cluster and even the size of each cluster. We describe how these algorithms can be transformed using permutation as a building block to satisfy one of Perpat patterns.

K means clustering. K-means clustering algorithm partitions n input elements into k clusters where each input element belongs to a cluster with the closest mean value. We manually modify this algorithm to fit into PERPAT patterns. We use the euclidean distance as the metric for computation. Figure 5 shows our transformed algorithm. A permutation function is appended at the beginning of the algorithm. After every iteration, we permute the K mean value, thus the read values are always deterministic whereas the write values occurring is equally likely, and it does not leak how many numbers belong to the same cluster. Since the permutation cost is O(K) it gets subsumed in the cost for accessing the K means. Thus the complexity for PERMUTERAM for K means clustering is equal to the original cost of the algorithm i,e $O(N.K)$.

```
1   Permute(A, key)
2  K–means(A)
3    for i = 0 to A.length
4     A[i].dist = infinity
5     A[i].cluster = 0
6
7    for i = 0   to K.length
8     K[i].mean = random(A)
9     K[i].new_mean = K[i].total = 0
10
11
12   for i = 0 to A.length
13     Permute(K, key)
14       for j = 0 to K.length
15        dis = distance(A[i].val,K[j].mean)
16        if A[i].dis > dis
17       A[i].dis = dis
18       A[i].cluster = K[j].mean
19       K[j].new_mean += A[i].val
20       K[j].total++
21
22   for i = 0 to K.length
23    K[i].mean = K[i].new_mean / K[i].total
```

Fig. 5. Modified K-means algorithm to satisfy PERPAT patterns.

Hierarchical clustering. Hierarchical clustering is an algorithm that groups input elements to build a hierarchy of clusters. We transform the original algorithm such that it exhibits PERPAT patterns. We consider a matrix representation to denote the distance between two points. Similar to K-means clustering, we use the Euclidean distance. After every iteration we permute the matrix to update the clustering information. Since the outer loops proceeds for n iterations, and in every iteration we permute the matrix which hides the points that are combined

together to form the next cluster. The original algorithm has cost of $O(n^3)$. Every matrix permutation has a cost of $O(n^2)$. Thus our modified program retains the original cost of the algorithm.

5.3 Building a data structure from an array

Input data is often represented in form of ordered data-structures such as heaps, priority queue, AVL trees and so on to compute queries on them efficiently. The execution profile while constructing such data structures from given input values leaks information about the values. These algorithms are transformed to PermuteRam by adding a permute function at the beginning of the algorithm. The permutation causes the access patterns in the execution profile of algorithm to be independent of the original input. Constructing of such data structures using PermuteRam makes the execution input-oblivious thereby preventing information leakage about the input values. The transformed PermuteRam algorithms preserve the property of data-structure that supports efficient queries.

5.4 Complexity analysis

Table 1 gives the execution complexity for PermuteRam algorithms with respect to original algorithm and using the best known generic ORAM solutions. Kushilevitz et al. scheme for oblivious RAM is the best known scheme which requires a constant private storage and has a overhead of $O(\frac{log^2 N}{loglogN})$ for each memory access. We use this scheme for comparison purpose. PermuteRam algorithms require a private storage of $O(\sqrt{N})$ and additive overhead of $O(N)$. The table gives the complexity comparison for the case studies discussed above using these two schemes. Using the big O notation, we observe that PermuteRam algorithms demonstrate complexities almost equal to the complexity of original algorithms. On the other hand, the case studies incur a multiplicative overhead for generic ORAM solution. This result is because of the fact that PermuteRam algorithms has an additive overhead of $O(N)$ incurred due to the permutation step. This additive overhead gets subsumed in the complexity for algorithms which perform in linear or super-linear time. Thus in all the sorting, clustering and creating data structures algorithms, PermuteRam performs better than general ORAM schemes. However, PermuteRam algorithms might not be an efficient approach for algorithms with sub-linear complexities where the permutation cost will dominate the overall complexity.

6 Related Work

Blanton et. al design input-oblivious algorithms for “dense" graphs for various applications like SSSD, minimum spanning tree, breadth-first-search and so on [2]. They use permutation primitive to hide the access patterns in these algorithms. Though this work does not intentionally design algorithms to satisfy Perpat pattern, we observe that all their algorithms meet the requirement of

Algortihms	Technique (Private Storage)		
	Original Algorithm	**Generic ORAM** $O(1)$	**PermuteRAM** $O(\sqrt{N})$
Constructing a tree data structure from a given array :			
Heap	$O(NlogN)$	$O(\frac{Nlog^3N}{loglogN})$	$O(NlogN)$
Priority-queue	$O(NlogN)$	$O(\frac{Nlog^3N}{loglogN})$	$O(NlogN)$
AVL tree	$O(NlogN)$	$O(\frac{Nlog^3N}{loglogN})$	$O(NlogN)$
Sorting Algortihms :			
Heapsort	$O(NlogN)$	$O(\frac{Nlog^3N}{loglogN})$	$O(NlogN)$
Quicksort	$NlogN$	$O(\frac{N^2log^2N}{loglogN})$	$O(NlogN)$
Selection sort	$O(N^2)$	$O(N^2 + \frac{Nlog^2N}{loglogN})$	$O(N^2)$
Clustering algorithms:			
K means	$O(N.K.)$	$\frac{O(N.Klog^2N.}{loglogN}$	$O(N.K)$
Hierarchical	$O(N^3)N$	$\frac{O(N^3)log^2N}{loglogN}$	$O(N^3)$

Table 1. Overhead comparison for different algorithms

PERPAT patterns. Goodrich et. al use a similar shuffling technique to design data-oblivious graph drawing algorithms for outsourced computation scenario with a small trusted private storage. M_2R system uses mixed network as a primitive to shuffle or permute the data in memory and thereby prevent leakage through access patterns. It leverages the specific pattern in map-reduce operations where all the data is first written to a fixed memory location, followed by a read operation. Thus, this paper intuitively converts the map-reduce functionality to demonstrate `deterRWP-and-deterLocP` pattern followed by `deterRWP-and-randLocP` pattern. Therefore, map-reduce operations are also a part of PERMUTERAM algorithms.

Recently, Ohrimenko et. al propose the use of shuffling data in memory to hide leakage in map-reduce jobs in presence of secure hardware like SGX [12]. The key observation is similar as in this paper i.e, access patterns during execution should be independent of the input data. Even their work uses Melbourne shuffle as a primitive to prevent leakage through access patterns.

Wang et. al design data-oblivious algorithms for programs and data-structures which exhibit a certain amount of predictability in their access patterns [17]. Similar to our work, they identify some specific behaviour in applications that allow designing efficient oblivious counterparts of the original algorithms. However, instead of permutation, their work uses ORAM as a primitive for hiding access patterns which incurs a multiplicative overhead. Keller and Scholl pro-

vide oblivious algorithms specifically for data structures but in the multi-party computation model [7].

All these work focus on specific applications like dense graphs algorithms, map-reduce jobs, graph drawing model or data structures whereas in this paper, we generalize the patterns and presents PermuteRam algorithms exhibiting Perpat patterns that aims to encompass a broader class of applications.

7 Conclusion

In this we work, we present a new class of algorithms called PermuteRam. PermuteRam algorithms satisfy the security property of input-obliviousness in presence of honest-but-curious adversary with access to algorithm execution profiles. The key contribution is the generalization of specific patterns in PermuteRam algorithms called Perpat that prevent information leakage through address access profiling. PermuteRam algorithms use a simple operation like permutation as its basic primitive. Several classes of algorithms like sorting, clustering and so on fall under PermuteRam. Knowledge of Perpat pattern allows developers to design efficient and practical input-oblivious algorithms.

8 Future Work

In this work, we rely on developer's skill to transform algorithms into PermuteRam. An interesting future work is to build a compiler that can automatically transform original algorithms into PermuteRam algorithms. Such a tool will be useful in scaling the expressiveness of PermuteRam algorithms. In addition to this, a promising direction is to use techniques like probabilistic symbolic execution to verify whether transformed algorithms adhere to the requirements of Perpat patterns.

References

1. Private Core. `https://privatecore.com/`
2. Blanton, M., Steele, A., Alisagari, M.: Data-oblivious graph algorithms for secure computation and outsourcing. In: Proceedings of the 8th ACM SIGSAC symposium on Information, computer and communications security. pp. 207–218. ACM (2013)
3. Brumley, D., Boneh, D.: Remote timing attacks are practical. In: USENIX Security Symposium 2003
4. Chen, X., Garfinkel, T., Lewis, E.C., Subrahmanyam, P., Waldspurger, C.A., Boneh, D., Dwoskin, J., Ports, D.R.: Overshadow: a virtualization-based approach to retrofitting protection in commodity operating systems. In: ACM SIGOPS Operating Systems Review. vol. 42, pp. 2–13. ACM (2008)
5. Dinh, A., Saxena, P., Chang, E.C., Ooi, B.C., Zhang, C.: M 2 r: Enabling stronger privacy in mapreduce computation. In: 24th USENIX Security Symposium (USENIX Security 15). USENIX Association (2015)
6. Goldreich, O., Ostrovsky, R.: Software protection and simulation on oblivious rams. J. ACM (1996)

7. Keller, M., Scholl, P.: Efficient, oblivious data structures for mpc. In: Advances in Cryptology–ASIACRYPT 2014, pp. 506–525. Springer (2014)
8. Kopf, B., Durmuth, M.: A Provably Secure And Efficient Countermeasure Against Timing Attacks. In: IEEE Computer Security Foundations Symposium (2009)
9. Kushilevitz, E., Lu, S., Ostrovsky, R.: On the (in) security of hash-based oblivious ram and a new balancing scheme. In: Proceedings of the twenty-third annual ACM-SIAM symposium on Discrete Algorithms. pp. 143–156. SIAM (2012)
10. Liu, C., Wang, X., Nayak, K., Huang, Y., Shi, E.: Oblivm: A programming framework for secure computation. In: IEEE Symposium on Security and Privacy (S & P) (2015)
11. McKeen, F., Alexandrovich, I., Berenzon, A., Rozas, C.V., Shafi, H., Shanbhogue, V., Savagaonkar, U.R.: Innovative Instructions and Software Model for Isolated Execution. In: Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy. HASP (2013)
12. Ohrimenko, O., Costa, M., Fournet, C., Gkantsidis, C., Kohlweiss, M., Sharma, D.: Observing and preventing leakage in mapreduce. In: Proceedings of the 2015 CCS. ACM
13. Ohrimenko, O., Goodrich, M.T., Tamassia, R., Upfal, E.: The melbourne shuffle: Improving oblivious storage in the cloud. In: Automata, Languages, and Programming, pp. 556–567. Springer (2014)
14. Stefanov, E., Shi, E., Song, D.: Towards Practical Oblivious RAM. CoRR (2011)
15. Stefanov, E., Van Dijk, M., Shi, E., Fletcher, C., Ren, L., Yu, X., Devadas, S.: Path oram: An extremely simple oblivious ram protocol. In: Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security. pp. 299–310. ACM (2013)
16. Wang, X.S., Chan, T.H., Shi, E.: Circuit oram: On tightness of the goldreich-ostrovsky lower bound (2014)
17. Wang, X.S., Nayak, K., Liu, C., Chan, T., Shi, E., Stefanov, E., Huang, Y.: Oblivious data structures. In: Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security. pp. 215–226. ACM (2014)
18. Zhuang, X., Zhang, T., Pande, S.: Hide: An infrastructure for efficiently protecting information leakage on the address bus. SIGOPS Oper. Syst. Rev. (2004)