TEE-MR: Developer-Friendly Data Oblivious Programming for Trusted Execution Environments

AKM Mubashwir Alam, Keke Chen

Abstract*—Trusted execution environments (TEEs) enable efficient protection of integrity and confidentiality for applications running on untrusted platforms. They have been deployed in cloud servers to attract users who have concerns on exporting data and computation. However, recent studies show that TEEs' side channels, including memory, cache, and micro-architectural features, are still vulnerable to adversarial exploitation. As many such attacks utilize program access patterns to infer secret information, data oblivious programs have been considered a practical defensive solution. However, they are often difficult to develop and optimize via either manual or automated approaches. We present the *oblivious TEE with MapReduce* (TEE-MR) approach that uses application frameworks, an approach between fully manual and fully automated, to hide the details of access-pattern protection to significantly minimize developers' efforts. We have implemented the approach with the MapReduce application framework for data-intensive applications. It can regulate application dataflows and hide application-agnostic access-pattern protection measures from developers. Compared to manual composition approaches, it demands much less effort for developers to identify access patterns and to write code. Our approach is also easy to implement, less complicated than fully automated approaches, for which we have not seen a working prototype yet. Our experimental results show that TEE-MR-based applications have good performance, comparable to those carefully developed with time-consuming manual composition approaches.

***Index Terms*—TEE, SGX, MapReduce, Data Analytics, Dataflow, Access Patterns, ORAM**

I. Introduction

With the development of resource-starving distributed applications in big data, artificial intelligence, and the Internet of Things (IoT), public clouds and edge devices have become common platforms for hosting data and compute-intensive tasks, which significantly expand the attack surface for potential adversaries. While encryption and secure data transmission protocols can ensure data security in transit and at rest, data in computing is still a big concern. For many years, researchers have been experimenting with novel crypto approaches based on secure software primitives, such as homomorphic encryption (HE) [10] and secure multi-party computation (SMC) [32], [47]. However, for complex data-intensive tasks like machine learning [58], these approaches are too expensive to be practical. More recently, the concept of hardware-assisted trusted execution environments (TEEs) has emerged as a more efficient and cost-effective approach, compared to the pure software-based cryptographic solutions [58].

A. M. Alam is with the Department of Computer Science, Marquette University, Milwaukee, WI, 53233. K. Chen is with the Department of Computer Science and Electrical Engineering, University of Maryland at Baltimore County, Baltimore, MD 21250
E-mail: mubashwir.alam@marquette.edu, kekechen@umbc.edu

TEEs depend on CPUs that provide hardware support to create an isolated environment within a potentially compromised computing environment, where a strong assumption holds: the entire system software stack, including the operating system and hypervisor, can be compromised. TEEs enable the concept of secure *enclaves*, which depends on hardware mechanisms to preserve the confidentiality and integrity of the enclave. Users can pass encrypted data into the enclave, decrypt it, compute with plaintext data, encrypt the result, and return it to the trusted client, avoiding expensive software primitives like HE and SMC. TEEs have been available on many commodity CPUs and supported by public clouds – for example, Microsoft Azure [44] and Alibaba [4] provide Intel Software Guard Extension (SGX) enabled servers, and Google has adopted AMD Secure Encrypted Virtualization (SEV) for confidential computing.

Although TEE enclaves cannot be directly breached, side channels are still there. The enclave interacts with untrusted memories and file systems, and the CPU cache is still shared among processes and virtual machines owned by different users. Recent studies have shown that side-channel attacks remain main threats to TEEs [27], [38]. Attackers can control and manipulate these channels via compromised operating systems or hypervisors, e.g., manipulating page faults and page-table entries and exploiting the flaws of modern CPU's micro-architecture execution optimization. Powerful attacks like Foreshadow [13] and Load Value Injection [68] can combine memory and cache access patterns and modern CPUs' speculative execution mechanisms to extract secrets during TEE execution.

While systematic approaches on side-channel attack defense are still missing, data oblivious programming appears an attractive and promising one. *Data oblivious programs expose invariant dataflow and execution path patterns regardless of the input values*. This unique feature can potentially address many side-channel attacks (details in Section II). Normal programs' data flow and execution paths vary according to different input data, and thus a specific input value may trigger unique steps to execute. Many attacks utilize the specific execution pattern to infer the input value [27], [38], to which data oblivious programs are immune.

Nevertheless, it is challenging for users to develop an oblivious solution for the following reasons. First, it's often time-consuming and error-prone to compose an oblivious data program manually. Some TEE-related studies [50], [59], [20] have utilized oblivious primitives, e.g., oblivious RAM [59] and oblivious branching [50], to compose an oblivious solution. However, manually composing an oblivious program for an arbitrary application is not a practical option for most developers. It requires careful analysis of application dataflow and skillful uses of the primitives to achieve performant solutions as naive transformation may have significant performance impact [9]. Developers can also use oblivious programming languages [41], [73]. However, it requires developers to learn a new language and completely re-write existing programs. Theoretically, the manual approach can also be automated via compiler techniques, e.g., static analysis and randomization [54]. However, it's difficult to automatically determine the best transformations [9], and we have not witnessed mature solutions yet.

A. *Scope of Our Research*

We hypothesize that, between the manual approach and the fully automated approaches, a semi-automated *framework*-based approach can significantly simplify developers' efforts and avoid the complexity of implementing fully automated and optimized conversion. The specific idea is to use an application framework to regulate application dataflow and implement the major access-pattern protection methods at the framework level, which will be transparent to the developers. By doing so, all algorithms that can be cast to the framework can benefit from the framework-level protection, and the developers only need to handle much smaller pieces of code with simpler access patterns. Following this idea, we developed the *oblivious TEE with MapReduce* (TEE-MR) approach. While the current approach has been implemented[1] on Intel SGX, it can be easily extended to other TEEs.

The framework approach hides the details of access pattern protection and their optimization within the framework. In the current implementation, we have taken the MapReduce (MR) framework and targeted data-intensive applications for several reasons. First, researchers and practitioners have accumulated extensive experience [19], [45], [43] on MapReduce solutions over the past decade, and thus it's also easy to learn and use TEE-MR. Second, the MR framework significantly reduces the developers' work in coding and access-pattern protections: only the user-defined mapper and reducer functions need to be handled. We have shown the significant workload reduction brought by this approach in experiments.

Specifically, our research has the following contributions.

- TEE-MR is the first framework approach for developing data-intensive data oblivious programs for TEEs. With a tiny framework core around a few megabytes, it can significantly reduce the developer's workload in coping with access-pattern protection.
- We have carefully studied the access patterns during each stage of the TEE-MR framework and designed robust and optimized framework-level access-pattern protection methods.
- We have conducted extensive experiments to understand the benefits of the TEE-MR framework. The result shows that TEE-MR can significantly reduce developers' efforts, and TEE-MR applications can achieve satisfactory performance comparable to manually composed oblivious programs.

[1]The preliminary version is SGX-MR [3], designed specifically for Intel SGX.

In the remaining sections, we will first present the background knowledge for our approach (Section II) and the threat model (Section III), and then describe the components in the proposed approach (Section IV), focus on the details of access-pattern analysis and protection (Section V), and discuss the evaluation result (Section VI). Finally, we examine the closely related work (Section VII) and conclude our work (Section VIII).

II. Preliminaries

We will present the necessary background knowledge before we dive into our approach. In the following, we will introduce the two main server-side TEE architectures, and the access-pattern based attacks in TEE applications.

A. *Mainstream TEE Architectures*

After the past ten years of development, so far the most well-known server-side TEE architectures are Intel SGX [21] and AMD SEV [5]. ARM TrustZone [67] is also available for most ARM processors. Since we focus on cloud-based applications, we will ignore TrustZone in the following discussion.

Intel Software Guard Extensions (SGX) promotes the idea of minimal trust computing base (TCB), i.e., the enclave. An enclave uses only the enclave page cache (EPC), a subset of processor reserved memory area (PRM) that cannot be accessed by other software, including operating systems and hypervisors. Enclave pages are encrypted in the EPC and only decrypted inside the processor with hardware AES support. This architecture requires the developer to carefully design the application program into two separated parts: the secret part of the program that needs to run inside the enclave, and the remaining parts running in untrusted areas out of the enclave.

Since Intel SGX is the earliest one available in commodity processors, researchers have conducted extensive studies on its security during the past few years. So far, a number of side channel attacks [27] have been discovered, most of which heavily depend on exploring access patterns of the victim program [27].

AMD Secure Encrypted Virtualization (SEV) takes an entirely different approach [5]. The memory encryption is done at the virtual machine (VM) level. It utilizes a CPU builtin secure co-processor to handle memory encryption operations. Each virtual machine will get a randomly generated fresh VM-specific AES key. All the VM pages are encrypted with this key in the main memory and only decrypted inside the CPU. SEV also includes the SEV-Encrypted State (SEV-ES) and the SEV-Secure Nested Page (SEV-SNP) techniques to address the security weaknesses associated with the early versions of SEV in virtual machine management and page table operations, respectively.

SEV does not require the application code to be rewritten, which was considered a huge benefit compared to SGX. For this reason, Intel also adopted the SEV approach and developed Intel Trust Domain Extensions (TDX) for virtual machine level protection. However, this convenience comes at a cost. The whole VM becomes a large trust computing base in this architecture, which potentially exposes even more weaknesses [38], including most of the side-channel attacks reported on SGX.

B. *Access Patterns Used in Side-channel Attacks*

The studies on side channel attacks have shown that program access patterns are critical to these attacks. Specifically, these attacks can be categorized into memory-based, cache-based, and micro-architecture-based attacks. We show how existing attacks utilize access patterns in the following.

Memory based attacks are further categorized into application-specific memory attacks and memory-page attacks, both of which heavily depend on access patterns. (1) Application-specific memory access patterns. Cash et al. [16] and Zhang et al. [75] show how an adversary can extract the content of encrypted documents by leveraging access pattern leakages. Ohrimenko et al. [49] and Zheng et al. [76] also demonstrate how sensitive information, such as age-group, birthplace, marital status, etc., can be extracted from data-intensive processing programs by only observing the network flow and memory skew. (2) Page-level access patterns. Intel SGX depends on OS provided virtual memory management, which exposes enclave memory's page access via page-fault interrupts [71], [62] or page table operations [14]. AMD SEV-Secure Nested Paging [5] addresses the integrity issues with the paging mechanism. However, the page level access patterns are out of its protection. It's widely believed most memory access-pattern attacks are also applicable to AMD SEV [38].

Cache-based attacks use cache hit or miss to detect victim's memory access patterns: if cache misses, the access time will be longer; otherwise, the access time is shorter. Cache-based side-channel attacks [48], such as *Prime+Probe* [42] and *Flush+Reload* [72], had been long exploited before TEEs became popular. The basic mechanism of cache attacks remains the same for systems with or without TEE. Since the cache is a shared resource between processes and virtual machines, an attacker can exploit fine-grained information at a specific stage of the program by probing the data access time in each cache line. Studies have shown a cache attack cannot distinguish the secret-dependent block IDs or block data from dummy ones if accessed through oblivious RAM [59], [2].

Micro-architectural attacks exploit the CPU's micro-architecture to retrieve secrets from TEE applications. Foreshadow [13] exploits meltdown-type [40] attacks on TEE applications. Load Value Injection (LVI) [68] is the most recent attack on Intel SGX that successfully retrieves the secrets from a victim's enclave. To make it work, the CPU's micro-architectural buffer must be prepared with some attacker-controlled secret value to perform the LVI attack. These attacks are powerful enough to extract plain text information from the TEE without physical access. Most such attacks depend on manufacturers' micro-architectural-level firmware patches to fix.

However, not all micro-architectural attacks can be prevented from firmware-level patches. Some micro-architectural-level attacks still utilize access patterns, e.g., in branch pre-fetching and pre-computation, a common technique used in

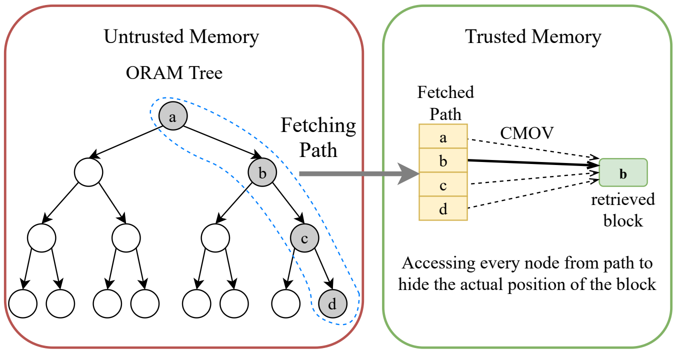


Fig. 1: Oblivious block retrieval from the ORAM Tree. The entire path containing the target block is loaded into the enclave to hide the access pattern. Furthermore, the CMOV instruction is used to hide the in-enclave access of the target block to address the page-fault attack.

CPU pipeline. For example, Bulck et al. [13] show that, by exploiting the timing of micro-architectural instructions, attackers can observe secret-dependent branches at the CPU instruction level. This type of micro-architectural-level attacks cannot succeed if the application developer hides the data-dependent branches with oblivious solutions. Therefore, oblivious programs can still help mitigate these attacks.

Other attacks include return oriented programming [37], which is difficult to defend and oblivious solutions may not help. Hardware and power analysis attacks assume physical accesses to the server, which does not match the TEE threat model assumptions, and they are not practical for cloud servers unless insider attacks are considered.

Overall, TEE manufacturers do not provide a systematic mechanism to protect against access pattern leakages. They have explicitly mentioned it's TEE application developer's responsibility to guard access patterns if the highest level of security guarantee is desired.

C. *Data Obliviousness*

Definition. If the execution path and data flow of a program does not do not change, or fully randomized, with different input data and parameter settings, we call this program is *data oblivious*. When all the steps of an algorithm or mechanism do not depend on input data, one cannot determine the nature of the data by observing the steps of that algorithm. Thus, oblivious solutions can effectively protect from attacks utilizing data-dependent access patterns.

Oblivious Primitives. Note that most algorithms are not designed with data obliviousness in mind. The goal of developing data oblivious programs is to eliminate any data-dependent operations. We list the primitives in the following categories that data oblivious programs heavily depend on.

- **Address-based Access**. This operation includes array element access or data block access. Exposing the position of accessed data is the fundamental access pattern. A naive solution is to iterate over the whole data structure to hide the actual accessed position. In contrast, Oblivious RAM (ORAM) [29] has been a well-accepted primitive for more efficiently hiding accessed addresses. It can effectively reduce the cost of oblivious access to $O(\log N)$ for a structure of N data items. By leveraging the ORAM construction, SGX applications can protect the access pattern of untrusted memory from the adversarial OS. The current SGX ORAM methods also partially address the in-enclave page-fault attack on ORAM-related operations. As shown by Figure 1, the popular SGX ORAM solutions, such as ZeroTrace [59] and Obliviate [2], utilize the most efficient Path ORAM [65] or Circuit ORAM [69], which uses a tree structure and paths from the root to the leaves to hide the actual accessed block. The tree and data blocks are maintained in the untrusted area. The drawback is the additional $O(\log N)$ cost per operation.

- **Data-dependent Branching.** Most programs contain data-dependent branching statements. Depending on the different inputs, a program execution may choose different paths, resulting in distinct access patterns. The following code snippet shows how an attacker can utilize the branching access pattern.

```
if (a >= b){
// swap a and b, and
// the page access can be observed.
}else{
// no page access.
}
```

The common method uses the CPU's conditional move (CMOV) instructions to eliminate the branching statements. A simplified example is shown as follows:

```
//if (a < b) x = a else x = b
//test a<b
CMOVL x, a
CMOVGE x, b
```

A few studies [50], [54], [3] have used CMOV instructions to provide code-level obliviousness for branching statements. Without specific conditional jumps, CMOV instructions move the source operand to the destination when a conditional flag is set. However, regardless the flag is set or not, it reads the source operand. Therefore, the access to the source operand cannot be used to infer whether the source is copied to the destination or not. Ohrimenko et al. [50] designed library functions *omove* and *ogreater* to wrap up the CMOV instructions for conveniently converting the branching statements. Notably, a completely oblivious execution needs to run both branches and select the desired result with the above method, which often leads to higher costs than non-oblivious version.

- **Circuit.** Circuits are considered a natural way to hide access patterns, as the circuit execution activates the gates in the predefined order regardless of the input values [31]. For example, the branching statement is readily implemented with a bitwise multiplexer with both branches executed and the result is selected by the multiplexer's input. Studies show that oblivious memory access is an expensive operation for the circuit approach. Many solutions depend on linear scan to hide memory access [63], [51], [15], which incurs high costs.
- **Oblivious algorithms.** Mechanically transforming memory accesses and branching statements to be oblivious often incurs significant performance penalties [9]. For example, MergeSort can be converted to an oblivious version by simply replacing every memory interaction of the merge phase with ORAM and unwinding data-dependent loops with fixed iteration loops. However, this direct conversion can be much more expensive than a specially designed oblivious sorting algorithm [9], such as BitonicSort [7]. Similarly, frequently used data-intensive operations, such as *join* and *group by*, can have more efficient dedicated oblivious versions [46].

D. MapReduce Processing

The basic idea of TEE-MR is to use an application framework to regulate dataflows, so that we can examine and protect the access patterns in an application-agnostic way. Currently, we have adopted the MapReduce framework to regulate application dataflows. MapReduce[22] is a popular programming model and also a processing framework, designed to handle large-scale data on a massively-parallel processing infrastructure. It has the major computation phases: map, optional combiner, shuffle and reduce. The input data is split into fixed-size blocks. Running in parallel, each Mapper takes the user-defined "map" function to process each data block and converts it to key-value pairs. An optional combiner can be used to pre-aggregate the map outputs, if hierarchical aggregation works for the reduce function. Then, all output key-value pairs are sorted, grouped, and partitioned individually for Reducers to fetch. Each Reducer then fetches the corresponding share of Map (or Combiner) output in the shuffling phase, sorts the shares, and applies the user-defined reduce function to process each group of key-value pairs to generate the final result. The MapReduce patterns are frequently seen in data-intensive applications, e.g., the filtering, group-by and aggregation. During the past 10 to 15 years, this processing pattern has been applied to numerous data-processing algorithms [45], which makes it easy to adopt to develop TEE-MR-based applications.

III. Motivation and Threat Model

Starting with an analysis of the typical threat model for data analytics algorithms, we then present the TEE-MR framework to address the access-pattern protection problem. Then, we analyze possible framework-level access pattern leakages and design methods to protect them. The design will be specific to the SGX environment. For AMD SEV, the design can be further simplified.

A. Threat Model

Users may run confidential computation tasks in an untrusted cloud server, where the server's OS or hypervisor can be compromised. The goal is to preserve data and program's integrity and confidentiality while availability is out of concern. A typical TEE, such as Intel SGX, provides a hardware-protected memory area, i.e., the *enclave* [21], and guarantees the integrity of the data and computation running inside the enclave. While adversaries cannot directly access the enclave, they can still glean information via side channels, such as memory access patterns and CPU caches. In contrast, the exposure of memory access patterns is inevitable as enclaves have to interact with the untrusted memory area. It's also reasonable to assume that attackers cannot access the cloud server physically, e.g., attaching a device to the server or touching the motherboard, which excludes all attacks based on physical accesses. Figure 2 illustrates the threat model.

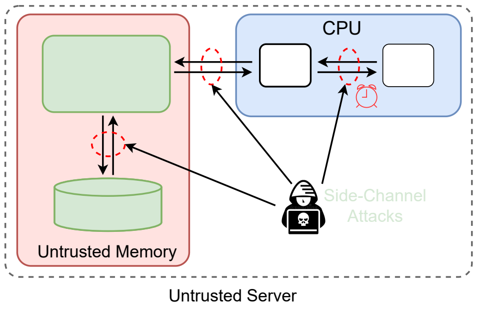


Fig. 2: TEE, side channels, and the threat model.

TEE Applications. Typical data analytics applications handle datasets much larger than the enclave memory and often they are sequentially scanned in multiple iterations. For simplicity, datasets are often organized in encrypted blocks and stored in a block-file structure. For security reasons, the enclave program that runs in the protected EPC area cannot access the file system APIs directly. When processed, they are first loaded into the main (untrusted) memory, and then passed to the enclave. Encrypted data blocks will be decrypted and processed inside the enclave. While adversaries cannot directly access the enclave, they can still glean information via side channels, such as memory access patterns and CPU caches. However, cache-based attacks target all CPUs (regardless of having TEEs or not) and thus need manufacturers' micro-architecture level fixes. In contrast, the exposure of memory access patterns is inevitable as enclaves have to interact with the untrusted memory area. It's also reasonable to assume that attackers cannot access the cloud server physically, e.g., attaching a device to the server or touching the motherboard, which excludes all attacks based on physical accesses. Figure 2 illustrates the threat model.

TEE Protection. TEE infrastructures provides the basic mechanisms for integrity and confidentiality protection for the data and programs in the enclave. We encrypt the code and data in the untrusted region of our framework with TEE provided crypto-library. Our framework also implements the block-level integrity checking mechanism, which will be described later.

Attacker Capability. Based on the analysis of the features of TEE-based data analytics algorithms, we make the following assumption for all adversaries: (1) They can compromise the operating system and any hypervisor of the host running the TEE applications. However, they cannot breach the TEE hardware and enclaves directly. As such, they can observe

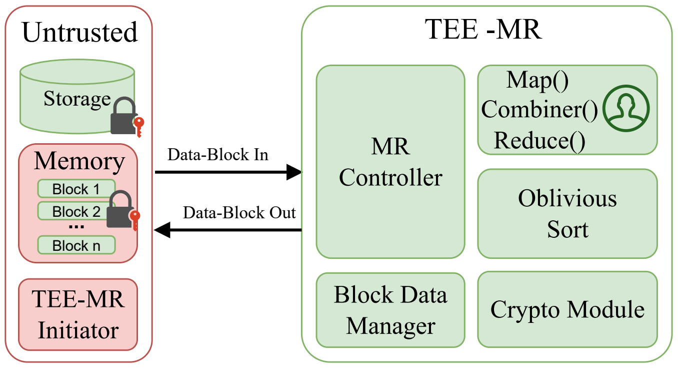


Fig. 3: High-level diagram of TEE-MR in SGX: Green shaded modules and memories either executed in the enclave or remain encrypted.

all of the data, out-of-enclave program execution, and access patterns between the enclave and out-of-enclave programs and inside the enclave at the page level. (2) Furthermore, adversaries are able to modify the data and program running in the untrusted memory, e.g., to compromise the integrity of data or force the generation of page faults for enclave pages.

Other side-channel attacks depending on direct hardware accesses, power analysis, or speculative execution [28], [68], [13], [30], [11] are out of the scope of this paper. It may depend on CPU manufacturers' firmware fixes to address the cache related attacks that affect all CPUs. We assume although a cloud infrastructure can be compromised the attacker cannot physically access the machine.

B. Existing Approaches to Constructing Oblivious Programs

Convert programs with data-oblivious operators. Developers can meticulously examine applications for access pattern problems. After analyzing the candidate solutions based on each context, the vulnerable code can be replaced with proper mitigation methods. There is an active line of work [20], [35], [50], [60], [46] available for manually modifying programs with data oblivious operators and oblivious algorithms for TEE applications. However, this approach requires developers to have extensive knowledge of sensitive access patterns and oblivious operators to re-design applications. It's time-consuming and error-prone, impractical for large-scale applications.

Rewrite programs with oblivious programming language. Some programming languages are designed to disguise program access patterns, such as ObliVM [41] and Obliv-C [73]. However, developers need to learn such languages and rewrite the programs, which is expensive.

Use special-purpose compilers. Another generic approach is to automatically convert a normal program to data-oblivious program with data oblivious compilers [54], [41], [74]. It depends on static analysis to detect vulnerable code blocks and replace them with data-oblivious alternatives such as ORAM, linear search, or other oblivious algorithms. While it is the most developer-friendly approach, static analysis cannot capture all problematic parts of the code and might be error-prone [64]. Naive direct transformations also result in low-efficiency code [9]. So far, the only available open-source tool is circuit-based converter, e.g., HyCC circuit generator [15].

IV. TEE-MR: Framework-Based Oblivious Programming for TEEs

This section includes two parts. First, we will present a prototype design of TEE-MR with SGX and describe how our approach utilizes the application framework to regulate application dataflows and make access-pattern protection easier. Then, we will analyze the framework-level access patterns and develop the protection methods in detail.

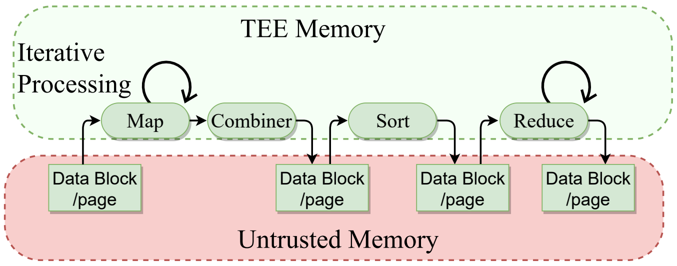


Fig. 4: Regulated dataflows between enclave and main memory.

A. *Prototype Design of TEE-MR with SGX*

The concept behind TEE-MR involves utilizing an application framework to ensure that input data conforms to a predetermined pattern for processing, regardless of the specific application. This approach enables us to systematically examine and safeguard framework-level access patterns. Meanwhile, developers can cover a wide range of applications by implementing straightforward user-defined functions. To grasp this concept, we will first outline the prototype system's structure and then detail the regulated flow of data. The prototype is based on the most popular TEE – Intel SGX. It's straightforward to export the implementation to virtual machine based TEEs, such as AMD SEV. We foresee that SEV-oriented implementation will be much simpler. However, all the analyses about the framework dataflow, sensitive access patterns, and mitigation methods will still be valid and applicable.

1) Components: According to the SGX working mechanism and features of data-intensive applications, we partition the entire framework into two partitions, i.e., the trusted in-enclave and untrusted out-enclave components. Figure 3 describes how they are designed.

- **Out-Enclave Component**. Since the data and the I/O library are in the untrusted area, our design needs to address both their confidentiality and integrity. We design a block file structure for encrypted data on disk and in the untrusted memory area. For simplicity, we assume that data records have a fixed size. Both block size and record size are tunable by the user based on the specific application. Each block also contains a message authentication code (MAC) to ensure data integrity. The whole block is encrypted systematically using the AES Counter (CTR) mode functions in the SGX SDK. To minimize the attack surface, we design the library running in the untrusted part to handle only block I/O, and a verification function inside the block data manager in the enclave to capture any adversarial modification on the loaded data blocks.
- **In-Enclave Component**. The TEE-MR controller handles MapReduce jobs and controls the dataflow of the application. Users only provide the map(), reduce(), and combine() functions to implement the application-specific processing. For simplicity, we will focus on the aggregation-style combine and reduce functions, such as COUNT, MAX, MIN, SUM, TOP-K, etc., which have been shown sufficient to handle many data analytics tasks [56]. Remarkably, with our careful design, the binary of the whole TEE-MR framework (without the application-specific map, combine, and reduce functions) takes only about 1.1MB physical memory. With the manually managed memory, we can also work with EPC memory as small as 3–4 blocks. The block size may depend on the specific application (we use block sizes varying from 2 KB to 2 MB for both WordCount and kMeans application in our experiment).

2) Regulated Dataflow: With a basic understanding of the components in TEE-MR, we describe how the application dataflow is regulated, which helps simplify access-pattern analysis and protection. While the original MapReduce is designed for parallel processing, Figure 4 sketches how the dataflow regulated by the MapReduce processing pipeline and processed sequentially in TEE-MR, and the interactions between the enclave and the untrusted memory. First of all, input files are processed by the data owner, encoded with the block format via a *file encoding* utility tool, and uploaded to the target machine running the TEE-MR application. Second, within a MapReduce job, all file access requests from the enclave (i.e., mapper reading and reducer writing) have to go through the TEE-MR block I/O module running in the untrusted memory area. Third, the intermediate outputs, e.g., of the Map, Combiner, and Sorting phases, can also be spilled out either by application buffer manager or system's virtual memory manager, in encrypted form.

Specifically, after the job starts, the Map module will read encrypted data blocks sequentially from the untrusted memory area, decrypt them, and apply the user-provided `map` function to process the records iteratively, which generates the output in key-value pairs. Note that the formats of both the input records and the generated key-value pairs are defined by users for the specific application (for readers who are not familiar with MapReduce programming, please refer to Section II and the original paper [22]). The controller accumulates the generated key-value records until they fill up a block, and then sort them by the key. With strictly managed memory, the filled data block will be written to the untrusted area temporarily. If we depend on the virtual memory management, the filled data blocks will stay in the enclave memory and swapped out by the system when needed.

We have restricted the reducer functions to a set of hierarchical aggregate functions (i.e., aggregates can be done locally and then globally), with which we can design their combiner functions for local aggregation. For example, for the COUNT function, the combiner will generate the local counts for a key, say k_i: $((k_i, c_1), (k_i, c_2), \ldots, (k_i, c_m))$ if there are m mappers. The reduce phase will get the final counts of $\sum_{j=1}^{m} c_j$. The inclusion of combiner has two benefits: (1) it can significantly reduce the cost of the most expensive phase: sorting, and (2) it can help address the group size leakage problem in the reduce phase that will be discussed later.

The Combiner outputs will go through the sorting phase, where the sorting algorithm will sort all key-value pairs into groups uniquely identified by the key. The Reduce phase then iteratively processed the groups of key-value pairs identified by the key.

Like the Map phase, key-value pairs stored in blocks will be handled sequentially in the Reduce phase. Specifically, the user-selected *reduce* function from the library takes a group of records with the same key and generates aggregated results. For all the aggregation functions we mentioned earlier, a sequential scan over the group will be sufficient to generate the aggregates. The aggregate of each group is also in the form of key-value pairs, which are accumulated in data blocks, encrypted, and written back to the untrusted area. The above described dataflow keeps the same for all applications that can be cast into the MapReduce processing framework. Now, by specifically addressing the potential access-pattern leakages in the MapReduce data flow, we can effectively protect a broad category of data-intensive SGX applications from access pattern related attacks.

3) Other Design Details: **Block Design.** We provide two types of block designs to meet different requirements. The first type is for fixed length records, mainly used for storing the intermediate key-value pair output, structured inputs, such as vectors, or approximately equal length inputs. The fixed-length record design can effectively protect from attacks that utilize the record length information, with a cost of padding for short records.

The second type is for variable length records. Some types of datasets, like graphs, may contain records with significantly varying lengths. Using fixed-length records will waste a large amount of space. However, storing exact-length records also reveal the record length information, which is not desirable. We will discuss design privacy-preserving variable-length methods in Section V-C1.

Integrity Guarantee. While SGX assures the integrity of enclave memory, both code and data that reside in untrusted memory remain vulnerable and can be modified. TEE-MR minimizes untrusted execution that was only used for storing and retrieving block data from untrusted memory. The integrity of the untrusted execution will be verified inside the enclave.

We consider three possible attacks to integrity: (1) modify a block, (2) shuffle a block with another block in the same file, and (3) insert a block from a different file (or a phase's output that is encrypted with the same key). To address all the attacks, we include the following attributes in the block: (i) Block ID, so that block shuffling can be identified, (ii) File Id, so that no block from different files can be inserted, and (iii) the block-level Message Authentication Code (MAC). At

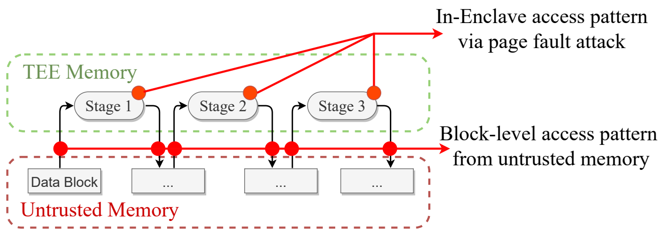

In-Enclave access pattern via page fault attack
TEE Memory
Stage 1
Stage 2
Stage 3
Block-level access pattern from untrusted memory
Data Block
...
...
...
Untrusted Memory

Fig. 5: Observable attack surface of TEE-MR (under SGX)

the end of each block, a MAC is attached to guarantee the integrity of records, before the whole block is encrypted. We also use the randomized encryption using AES-CTR encoding to make sure identical blocks will be re-encrypted to totally non-distinguishable ones so that adversaries cannot trace the generated results in the MapReduce workflow. A simple verification program runs inside the enclave that verifies the IDs and MAC, after reading and decrypting a block.

V. Framework-Level Access-Pattern Analysis and Protection

This section focuses on the framework-level sensitive access patterns. We will analyze the attack surface of the framework, discuss each sensitive access pattern, and design the mitigation methods to protect the sensitive access patterns.

A. Attack Surface of the Regulated Dataflow

In section IV-A, we have shown how the framework's basic design regulates the data flow and maintains the data confidentiality and integrity. The regulated data flow also makes the analysis of access patterns much easier. Figure 5 focuses on the access-pattern-based attack surface of the framework. Essentially, an attacker can monitor the block-level access pattern when TEE-MR reads or writes block data from TEE. Although the blocks are encrypted, the attacker can still record the statistics of block read/write operations. Moreover, when trusted modules execute within TEE, a skilled attacker with access to the host operating system can observe the page-level in-enclave access patterns.

Based on the TEE-MR dataflow analysis, we can identify several critical data access patterns pertaining to different phases: Map's input, intermediate processing, combining, and output; Shuffling/Sorting's input, sorting, and output; and Reduce's input, aggregation, and output. Among these access patterns, Map's input and Reduce's output involve only sequential block reads/writes. Thus, individually they do not leak additional information except for input and output file sizes.

The sensitive access patterns can be summarized into three groups:

- **Data in processing.** The most expensive step in this data-intensive pipeline is the sorting phase that may expose the relative order between input records. We adopt oblivious sorting for both block-level and in-enclave protection. Still, an adversary can observe the page-level access pattern of the framework operations via page-fault interrupts. We analyze the in-enclave processing code and design corresponding oblivious methods.

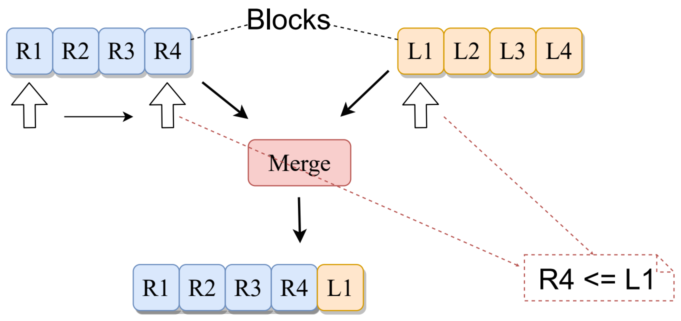


Fig. 6: Access-pattern leakage in MergeSort by observing the interactions between the enclave program and the untrusted memory. If R4 $\leq$ L1, then the Merge Phase reads R1 to R4 one by one to write sorted output and then reads L1 to L4. By observing this reading pattern (movement of the pointers), the adversary will learn that the entire right sub-list is smaller than the left sub-list.

- **Data in input/output.** With aggregation operations done in the reduce phase, the *group sizes* can be exposed, which may leak the application-specific private information. Furthermore, if records have variable lengths, an adversary may track unique records by their lengths in each phase that may result in leaking access patterns of specified records. Finally, iterative processing algorithms need to carefully handle the output from the previous iteration, e.g., removing dummy records inserted to protect group sizes.
- **User-defined functions.** While we focus on the framework-level protection to minimize developers' efforts in access-pattern protection, still a user may wish to have the oblivious UDFs.

In the next subsections, we will organize the content in these three aspects. For each aspect, we will analyze the access pattern and then give the protection method. These mitigation methods can be roughly categorized as follows: (1) replacing sensitive codes with oblivious branching, loops, or algorithms; (2) introducing randomness into the access patterns, e.g., with differential privacy; or (3) injecting padding data to align the sizes to make the sensitive data output difficult to distinguish.

B. Access Patterns in Data Processing

1) Sorting: We start with the most expensive part of the whole data flow – the sorting phase. Traditionally, the MapReduce framework adopts the MergeSort algorithm for its simplicity. As the shares of Map(or combiner)-output have been sorted individually, MergeSort only needs to merge the sorted shares. However, as we show next, it leaks vital information about the records under sorting.

If we look at the block-wise merge process carefully, we can identify some unusual merging behaviors, which reveals some sensitive attributes of the processed records. Starting from the records in each block, the main body of MergeSort is to merge two individually sorted lists of records recursively until all records are sorted. Assume the two merged lists contain the records $\{L_i\}$ and $\{R_j\}$, respectively, as shown in Figure 6. If one of the lists has all the values larger (or less) than the other, the corresponding access pattern will be continuously reading the blocks in one list first, followed by the whole list of the other. This block access pattern can be observed in-enclave if the enclave memory buffer is large enough to contain all sorted blocks; or outside of the enclave if the untrusted memory or hard drive is used to hold blocks. It may leak sensitive information for real applications. For example, in the WordCount program, adversaries might be able to guess the frequencies of words and derive the word distribution.

Protection Methods. Since this access pattern can be in enclave memory or outside of enclave, in the following we will discuss the protection method for outside-enclave case first, while the in-enclave page-level protection will be discussed later with other issues. A popular method for hiding the block-level access patterns is ORAM. However, we notice that applying the direct transformation with ORAM for block I/O is not the best option, as we show in experiments and also other researchers [9]. In contrast, a dedicated *oblivious sorting* algorithm [7] is more efficient. In the TEE-MR framework, we extend the well-known data-oblivious *BitonicSort* [7] algorithm as the default sorting to processing blocks with fixed number of records per block. Note that BitonicSort takes $O(N(\log N)^2)$ block accesses for N blocks. The ORAM-based MergeSort's cost has a similar complexity, but is constant times higher than BitonicSort as our experimental result shows. We also show in experiments that other options, such as ORAM+oblivious hashing, can also be an efficient option for a specific type of applications.

2) In-Enclave Page Access Patterns: In-enclave execution is vulnerable to the page-access attacks [27]. We have carefully analyzed the framework-level operations and categorize them: the branching statements and the sort operations. In particular, the sort operations include the in-enclave sorting part of the sorting phase and the map-output sorting before the combiner is applied. Note that the repetitive page accesses for the same data block reveal the ordering of records in a pair of blocks, a similar scenario to Figure 6, but happening at the page level.

Protection Methods. For the sorting operations, at first glance, we can just use the BitonicSort algorithm to hide the record-level access pattern as well inside the enclave. However, the core operation of this in-enclave BitonicSort, *compare-and-swap*, still shows the data-dependent access patterns. The following code snippet shows how possibly the access pattern is associated with the record order.

```
if (a >= b){
// swap a and b, and
// the page access can be observed.
}else{
// no page access.
}
```

Note that this is the common branching statement problem for all parts of framework-level in-enclave code. Thus, we adopt the oblivious-if [9] to handle all the if-else statements. The basic idea is to use the CMOV instructions to hide the page access patterns. However, the oblivious swap operation indeed occurs high costs, about $2 - 2.5\times$ cost increases for the sorting phase only.

C. Access Patterns in Data Input/Output

We have handled the access patterns in processing data. Next, we look at the problems when data is transitioned from or into the static form, including the record length, the group sizes in aggregation, and padding removal between iterations.

1) Record-Length Patterns : Variable record lengths leak important information, e.g., allowing attackers to trace records of specific lengths in sorting. A simple design to guard this length information is fixed-length records. The fixed-length design adopts the largest record length for all records, which disable the length-based record identification. However, it wastes space and incurs high processing costs. For some applications, especially, graph algorithms, record lengths can vary dramatically, e.g., the largest record may contain thousands of times more neighboring nodes than the smallest one. Thus, simply adopting the fixed-length design may not be ideal.

Protection Methods. We study two record designs. First, we keep the fixed-length design for some applications, e.g., data mining algorithms processing fixed-length vectors, which ideally preserves confidentiality without making any sacrifice. Second, we design and evaluate several efficient variable-length methods to balance space efficiency and information hiding.

Our first method is based on ***differential privacy (DP),*** [24], which is recognized as the de facto strongest privacy protection method. The basic method is to pad records with randomized sizes. Consider a record-length query function, e.g., the max length of record in any subset. The function sensitivity would be the maximum record length of the dataset, denoted as m. A straightforward solution for differentially private padding is to use a Laplacian noise distribution $Lap(0, m/\epsilon)$. There are two challenges in practice: (1) the padding length cannot be negative, and (2) the global m can be very large, resulting in high noise levels. The negative padding can be simply addressed by offsetting the noise distribution to $Lap(\delta, m/\epsilon)$, where δ is constant chosen to make sure about 99% of noise positive: $\delta \approx 3\sigma$, where σ is the standard deviation $\sqrt{2}m/\epsilon$ (the variance of $Lap(\delta, b)$ is $2b^2$). For the remaining very small number of cases generating negative noise, we set the padding length to 0. However, we find that for a large global m, the cost is close to the fixed-length design and thus impractical to use.

We also investigated a much more space-efficient (and thus weaker privacy protection method) with ***k-Anonymization.*** Note that the fixed-length method pads every record to the maximum length of the whole dataset. If we use a group-wise padding method instead, i.e., every record in the group padded to the maximum length of a group of k records, we can save space significantly. As Algorithm 1 shows, this k-anonymization method partitions the records sorted by record length into groups, where each group has at least k records and the records in the same group share the same padded record size. The parameter k implies a trade-off between the cost and confidentiality of record size. Our experimental result shows this strategy gives a lower overall storage and processing costs than the fixed-length solution, certainly with a weaker privacy guarantee than the global differential privacy method.

Algorithm 1 K-Anonymized Bucketing

```
1:  Function KA_Buckets($R, k, Q$)
2:  Note: R: original records, k: the minimum number of
    records in each group, $Q$: the length-anonymized records
3:  $N \leftarrow$ the number of records in $R$
4:  $R \leftarrow$ sort_by_record_length($R$)
5:  for $i \leftarrow 1$ to $N$ with step $k$ do
6:      $i_m \leftarrow$ min($i + k$, $N$ )
7:      $s_m \leftarrow$ record_length( $R[i_m]$ )
8:      for $j \leftarrow i$ to $i_m$ do
9:          $Q[j] \leftarrow$ pad_record( $R[j]$, $s_m$ )
10:     end for
11: end for
```

```
1:  Function pad_record($r, s_m$)
2:  $r$: a record; $s_m$: maximum length of the record
3:  $l \leftarrow$ length($r$)
4:  if $l == s_m$ then
5:      return $r$
6:  end if
7:  $b \leftarrow allocate(s_m)$
8:  for $i \leftarrow 1$ to $s_m$ do
9:      if $i \leq l$ then
10:         $b[i] \leftarrow r[i]$
11:     else
12:         $b[i] \leftarrow 0xFF$
13:     end if
14: end for
15: return $b$
```

2) Group Sizes: The Sorting phase will order key-value pairs by the key. For many applications, when the key type is defined as integer or string, the identical keys will be sorted into the same group. The controller will sequentially read the sorted groups and transfer the records containing the same key to the reduce function. Furthermore, in an aggregation-based reduce function, each group will be reduced to one key-value pair. By observing the input/output ratio, adversaries may estimate group sizes, as shown in Figure 7. In extreme cases, multiple blocks can be reduced into one record, which may lead to severe information leakage. For example, it can leak the word frequencies for the WordCount program, which can be used to guess the associated words.

Protection Methods. We consider a simple method to address the group-size leakage problem, which consists of two parts. Since the difference between Reducer's input and output sizes allows the inference of group size, the first method is to make the input number of records per group oblivious. We make the Combiner component mandatory in TEE-MR, which does not breach the utility of framework for most applications [56]. As such, each record in the reducer input may represent an aggregate of multiple records – by counting the observed number of records, one cannot infer proper group sizes.

A concern is whether the *frequency* of such combiner-output records still preserves the *ranking* of actual group sizes. The intuition is that the key of a large group may be presented in more map outputs than a key of a small group. We analyze the

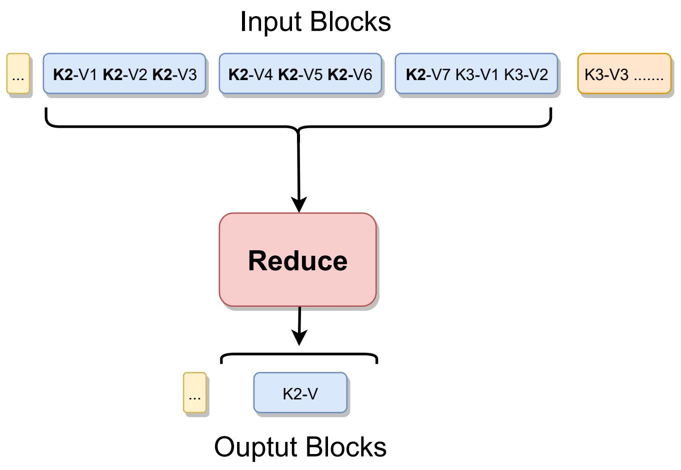


Fig. 7: Group-wise aggregation in Reduce phase may leak group sizes. K2 spread over three blocks. Thus, the access pattern will be reading three blocks sequentially and then possibly writing out one block.

observed group size after adding combiners to check whether the size ranking is still preserved. Figure 13(a) for WordCount and Figure 13(b) for kMeans both show that the ranking of group sizes is not preserved at all.

To completely address the possible ranking leakage, we can further take the following approach, by *randomly padding dummy records*, e.g., every time an input block is scanned, output an aggregated record, regardless of whether or not the actual aggregation happens. Note that since all the records are encrypted and non-distinguishable, it's impossible to distinguish dummy records from normal ones by observing the encrypted reduce output. However, this approach will incur more costs for sure, including the padding removal step if multiple iterations of the MapReduce job are needed.

3) Padding Removal: The discussion so far is based on single-round processing algorithms. When iterative processing is considered, a unique problem is to appropriately handle the injected dummy records in the reducing phase if the padding method is applied. Many data mining algorithms include iterative processing steps: between iterations, the output of the last iteration is pumped to the next iteration as the input. If we keep all dummy records and pump them into the next iteration, the processing cost will steadily increase after every iteration. However, a naive scan-and-filter step to remove the dummy records will make them identifiable, beating the purpose of padding.

Protection Methods. We design an approach to filter the dummy records obliviously. While creating dummy records, we assign a tag in each record to distinguish between actual and dummy records, e.g., 0 and 1, respectively. Meanwhile, we keep track of the total number of dummy records K among the N number of total records. To filter the dummy records, we first perform oblivious sorting by the filtering tag, and then take the first $N - K$ records from the sorted list. This step will significantly increase the overall cost as we will see in experiments.

D. Access Patterns in User Defined Functions

So far, we have addressed the major access pattern issues at the framework level. The framework now safeguards the sensitive access patterns, which reduces the attack surface to just the user-defined functions. These user-defined functions include map, combine, and reduce. The good news is that these function all handle within-block and in-enclave operations, leaving much simpler access patterns to users. Thus, a simple approach, e.g., directly transforming the data-dependent operations, might work just fine.

Protection Methods. As recent studies [9], [50] show, for within-block processing, we only need to take care of several data-dependent statements: if-else, loops, and logical operators. Note that a logical operator like "a && b" is implicitly converted to an if-else statement:

```
if (a == True) return b
else return False
```

We provide the corresponding oblivious APIs so that TEE-MR users can replace these statements. We may further deploy a static analysis mechanism like ObliCheck [64] to verify whether users have correctly and comprehensively covered all these sensitive operations.

VI. Experimental Evaluation

The TEE-MR approach is designed to reduce the manual efforts in developing oblivious programs with a tiny engine. The experimental evaluation has the following goals. (1) We show that the TEE-MR approach has achieved the design goal: it can significantly reduce developers' efforts, and the TEE-MR applications perform comparably to manually composed solutions. (2) We also examine the main design decisions to identify the best options for protecting critical access patterns.

A. Experiment Setup

TEE-MR is implemented with C++ and the Intel SGX SDK for the Linux environment. Our core framework consists of only about 2000 lines of code. The entire TEE-MR framework runs inside the enclave, except for a small component in the untrusted area that handles block-level read/write requests from the enclave. The compiled framework without the application code takes about 1.1MB. We use 128-bit AES-CTR encryption to encrypt the data blocks in the untrusted memory. We have implemented a customized bitonic sort that works with blocks. To protect record level access patterns in the enclave, we also apply the bitonic merge operation to obliviously merge in-enclave blocks. Furthermore, we have implemented all the oblivious operations for the core framework, and developed a combiner/reducer library for aggregation. The experiments were conducted on a Linux machine with an Intel(R) Core(TM) i7-8700K CPU of 3.70GHz processor and 16 GB of DRAM.

Benchmark Applications. We adopt three well-known data-intensive applications in our evaluation: WordCount, KMeans, and PageRank. WordCount takes a document collection and output the frequency for each word. It is an essential tool for natural language processing and has been included in various tutorials as an example of data-intensive processing. KMeans [33] is a fast, simple clustering algorithm. It takes the initial cluster centroids and iteratively conducts the two steps: (1) cluster membership assignment for each record and (2) the centroid re-computation, until the clustering converges, where the centroids (or all records' cluster membership) do not change anymore. In all KMeans related experiments, we only execute one iteration of the learning process. PageRank is a well-known link analysis algorithm [12]. It determines the importance score of every node in a directed graph based on the edge references. As the entries in a graph adjacency list have significantly varying lengths, it's perfect for our evaluation of variable record length and iterative processing.

Datasets. We have used several application-specific datasets in this experiments. Samples of the Bible dataset are used for evaluating WordCount. KMeans is evaluated with a simulated two-dimensional vector dataset that fills data blocks with randomly generated records. For PageRank-related evaluations, we have used three graph datasets: wikipedia (crocodile) [57], DBLP citation [55], and Human Protein-Protein Interaction (HPPI) [1] networks.

B. Compared Approaches

Compared to the manual composition approach and the automated approaches, TEE-MR is a semi-automated approach. We show that it can significantly reduce developers' efforts in the manual approach and generate comparably efficient oblivious programs. Furthermore, TEE-MR applications are much more efficient than the oblivious solutions mechanically transformed by the circuit-based automated converter, the only working automated approach we can find so far.

Manual Composition. To find the optimal composition, we have carefully examined implementations for the sample applications with ORAM and within-block oblivious operators as the building blocks. Specifically, we also noticed the most expensive operation is aggregating by groups, which can be implemented with either sorting or hashing. Correspondingly, we examine the following two candidate solutions.

- *Manual Composition with ORAM-Sort.* The ORAM-Sort approach utilizes ORAM as the block I/O, which was promoted by some approaches [59], [2]. Thus, the sorting operation will be MergeSort with ORAM for block accesses. Both the input and output will use sequential block I/O without ORAM protection. Note that we did not apply any additional access pattern protection methods, such as group-size disguising for in-enclave processing. We will show that this manual approach is not straightforward to construct, and, if not carefully designed, e.g., using ORAM directly for all basic block operations like sorting, we will get a solution more expensive than TEE-MR.
- *Manual Composition with ORAM-Hash.* The ORAM-Hash approach refines the ORAM-Sort approach by using hashing to find the items in aggregation. The basic workflow is as follows. The program sequentially reads input blocks without ORAM, processes the block, and generates the records for aggregation. Then, an incremental record-by-record aggregation step is applied with groups indexed and retrieved with an efficient I/O optimized hashing method, such as extendible hashing [26]. Specifically, when a new record is extracted from the input, the block containing its aggregate item (i.e., the group) is retrieved with ORAM, and then the corresponding aggregate is updated. To optimize the performance, we have also applied block-wise pre-aggregation and maintained a buffer of the same size as used for TEE-MR's sorting to reduce the number of ORAM accesses. Note that this approach actually benefits some applications, such as kMeans, when the number of groups in aggregation is so small that one or a few blocks can hold all the key-value pairs. The ORAM-Hash-based operations are thus more efficient than ORAM-Sort for this kind of applications.

The ORAM-Hash method will need to maintain a hash index – the mapping between hash function outputs and block ids, which we have used a HashMap to maintain for simplicity. However, researchers have found that processing such a hash table in enclave may incur unique page-level access patterns [34], [46]. An enhanced scheme [34] to protect these patterns will require additional $O(N)$ cost per key search if the table contains N keys. We did not implement these protections in our evaluation. With such suboptimal implementations, we show in general both ORAM schemes still cost significantly more than TEE-MR.

Automated Conversion Approach. Automated approaches include compiler and circuit-based methods. Unfortunately, we could not find a working open-source implementation for the few published compiler approaches [54]. Due to the sheer complexity of implementing such a compiler, we have not included the compiler method in the evaluation. For circuit-based, we use the HyCC circuit generator [15] that converts a plain application implementation to the corresponding circuit. HyCC was originally designed for converting programs to Garbled Circuits [8].

C. Result Analysis

Our evaluation is conducted on three aspects to show: (1) the amount developers' efforts can be reduced with the TEE-MR, (2) the performance of TEE-MR-based oblivious programs, and (3) design decisions made in TEE-MR protecting access patterns. We analyze the corresponding results as follows.

1) Reducing Developers' Efforts with TEE-MR: We are curious about how easily developers can achieve oblivious solutions in comparison to TEE-MR. This evaluation does not include the extra time learning the different approaches – apparently, developers need to take a significant amount of time to learn the manual approach and the framework approach. Instead, we look at the result of developing the evaluated applications to understand the difficulty levels of using different approaches. We also assume developers will use a library of oblivious primitives, e.g., ORAM, oblivious branching, and oblivious sorting. The use of library will also significantly reduce the line of code (LOC) for the manual and framework approaches.

Table I summarizes the additional effort a developer need to achieve data oblivious applications. (1) It shows the manual

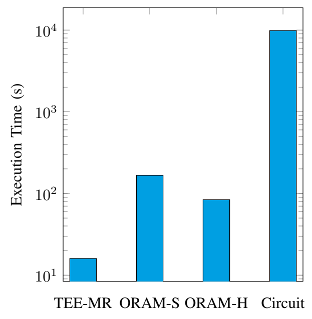
TABLE I: Developers' effort to implement the oblivious solutions. LOC: Total lines of code, AP: Acess-pattern sensitive code segments. LOC-overhead: lines used to hide APs.

Application	Manual			Circuit			TEE-MR		
	LOC	LOC-Overhead	AP	LOC	LOC-Overhead	AP	LOC	LOC-Overhead	AP
Word Count	277	21	6	155	0	-	22	0	0
KMeans	330	24	4	263	0	-	58	6	1
PageRank	305	22	4	160	0	-	45	3	1

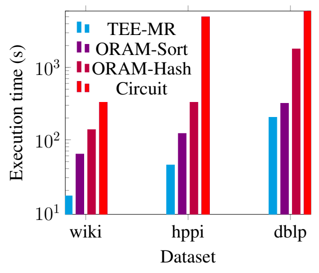
approach need to identify one to six sensitive code segments to hide access patterns with data-oblivious alternatives. This approach also requires the developer to write more lines of code than other approaches, even when the oblivious library is used. It indicates much more development efforts in analyzing the original code and conducting the conversion. (2) The circuit approach is fully automated, and the developer does not need to do any additional work. However, the generated programs have poor performance as we will show later. (3) With TEE-MR, the developer only focuses on small pieces of application-specific code, such as the map and reduce functions, dramatically reducing the developer's burden compared to the manual approach. Specifically, it does not require any LOC overhead for WordCount, and for KMeans it only requires six lines of code to solve one access pattern issue.

Cost of Learning MapReduce. Note that the reported developer's cost of TEE-MR does not include the cost of learning MapReduce programming, which is out of this paper's scope. However, previous studies have supported that junior to medium-level developers (e.g., Computer Science college students) should be able to learn the concepts and apply the skills to solve real problems within a reasonable time frame, e.g., 2-3 weeks. As early as around 2009-2012, universities started introducing MapReduce into big data and distributed systems courses, which include our effort [18] and large universities like the University of California at Berkeley (UC Berkeley) [53]. We have witnessed that junior and senior-level undergraduate Computer Science students started learning the main concepts and finished a MapReduce programming task within 2-3 weeks [18]. The evaluation was done in three semesters during 2014-2015, and most students reported they could finish the programming task within 20-25 hours in the two-week period. Similar results were reported by UC Berkeley in 2012, which integrated MapReduce programming tasks and the cloud-based deployment into lower-division CS courses that students in third or fourth semesters typically took. Students finished one two-hour lab and one two-week project assignment with satisfactory learning outcomes [53]. Thus, we conclude that the cost of learning MapReduce is reasonablly low, but it significantly benefits big data analytics and approaches like ours.

2) Performance on Benchmark Applications: We compare the implementations of TEE-MR and different oblivious solutions for the benchmark applications. We implement the manual approaches in two different approaches: ORAM-Hash and ORAM-Sort with ZeroTrace as the oblivious block I/O interface. We have ignored another trivial manual approach, i.e., following the same MapReduce dataflow, which gives similar results as TEE-MR. As we discussed earlier, the ORAM-Sort solution has used an o-swap protected MergeSort. However,



(a) WordCount cost comparison, block size = 2KB.



(b) PageRank cost comparison, block size = 2KB.

Fig. 8: Application-based comparisons for WordCount and PageRank.

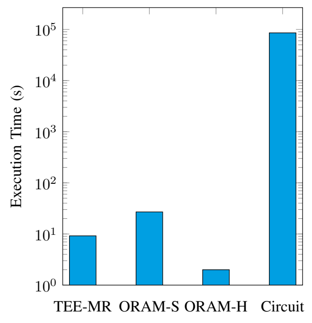
the in-enclave hashmap access has not used oblivious operations. Thus, the full version of ORAM-Hash will have costs higher than the reported. In all the implementations, we have used the same number of working memory blocks, e.g., only the number of working blocks for the sorting algorithms to maintain a minimum trust computing base.

The applications show almost similar patterns for the three approaches, while KMeans' ORAM-Hash shows a different pattern, which we will analyze in more details. Figure 8(a) and 8(b) for WordCount and PageRank show the TEE-MR-based implementations are significantly faster than all the compared approaches, and the circuit-converted programs take unbearable costs. In contrast, ORAM-Hash for KMeans has the best performance among all approaches (Figure 9). The reason is that the small number of centroids can be stored in one or a few in-enclave working blocks for incremental aggregation in the hash-based solution, which is much faster than sorting records and then aggregate. Figure 9(a) shows that this benefit does not hold when kMeans generates a large number of clusters, e.g., the k is large when kMeans is used for data summarization.

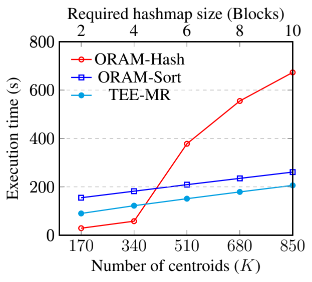
TABLE II: Application Level Comparisons of TEE-MR and ORAM for different applications. Block size = 2 KB

Application	Input size (records)	TEE-MR (s)	ORAM (s)
KMeans	68 $\times 10^4$ records	20.0	4.1
WordCount	20,000 2KB blocks	194.6	1593.2
PageRank (wiki)	170,918 edges	16.8	138.4

3) Major Design Decisions in TEE-MR: We conduct extensive experiments to show how different design decisions in the following three aspects may affect the overall performance. (1) We will study the impact of variable and fixed record



(a) kMeans cost comparison for an increasing number of records. Block size = 2KB.



(b) kMeans cost comparison for an increasing number of centroids (K). Number of records 17 $\times 10^4$, Block size = 2KB

Fig. 9: Application-based comparisons for kMeans.

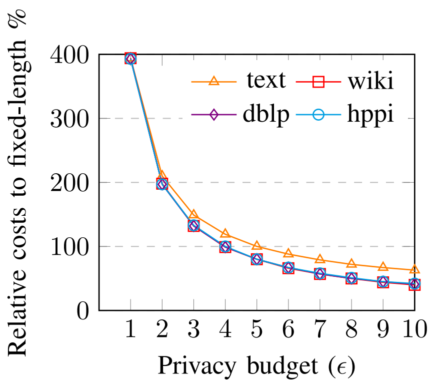


Fig. 10: Average storage cost of differentially private lengths for various graph datasets

length for different applications. (2) We show the benefits of the dedicated oblivious sorting, BitonicSort, compared to MergeSort with ORAM for block I/O. (3) Group sizes in the aggregation stage are sensitive. We experiment with different methods to show the corresponding costs.

Variable vs Fixed Record Length. As described in section V-C1, keeping variable record lengths as they appear will lead to a potential access pattern problem. However, using a fixed length for the whole dataset will waste space and increase processing costs. In this experiment, we first show the impact of the fixed-length solution on the memory cost, then explore two memory saving approaches using differential privacy and the k-anonymization accordingly.

TABLE III: Memory cost comparison: fixed-length records consume much more memory than variable-length records.

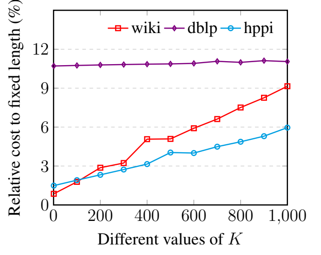
Dataset	fixed (MB)	fixed/variable (times)
wiki	157.6	115.0
hppi	143.2	66.0
dblp	49.0	8.4
text	10.9	3.5
vector	10	1.0

Table III shows the memory overhead of setting maximum record length to all the records. The memory overhead varies over different types of data, among which graph datasets incur the most overheads due to the large differences between record lengths. Text data can be partitioned almost arbitrarily, and thus fixed length records work well. Finally, vector data has fixed length and thus incurs no additional overhead.

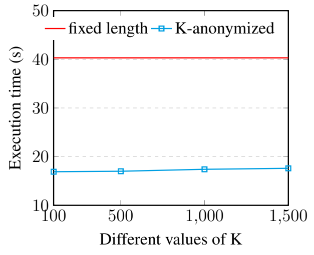
Figure 10 shows the memory overhead of differentially-private record lengths compared to fixed ones with the DP method described in section V-C1. Only when ϵ is larger than a threshold around 4, the cost can be smaller than the fixed-length method. Furthermore, for a commonly acceptable privacy budget, i.e., $\epsilon < 10$, the cost is still about 50% of the fixed-length method, which is not impressive for space saving.

Compared to the global DP method, the k-anonymization method gives much more space saving. In Figure 11(a) we estimate the percentage of the memory requirement of k-anonymized variable-length records over fixed-length data with different settings of k. The wiki and hppi datasets increase significant with increasing k values, due to the large variance of record lengths. However, the overall cost is still less than 10% of the fixed record-length. Because of the smaller variance in record-length, the dblp dataset showed negligible impact on large k values.

We also conducted experiments to estimate the processing cost difference between k-anonymized-length records and fixed-length records, using the PageRank algorithm. In figure 11(b), we see k-anonymized length performs significantly better than fixed length. The execution time increases linearly with larger k values due to the increased overall data size.



(a) Storage cost by using k-anonymized lengths.

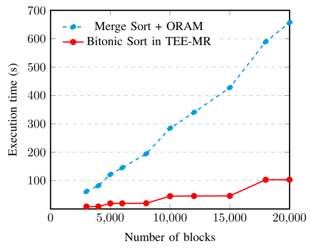


(b) Processing costs with different types of records in PageRank.

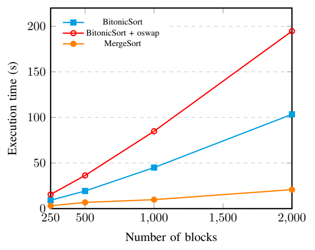
Fig. 11: K-anonymized variable-length records

Options for Oblivious Sorting. A key operation in the framework is key-value record and block sorting. We have shown that it is necessary to use oblivious sorting algorithms to protect the important relationship between keys. We compare the costs of BitonicSort and MergeSort with ORAM for block I/O. In both solutions, we also examined the impact of CMOV-based protection for disguising branch statements. Figure 12(a) shows that BitonicSort is much faster than MergeSort + ORAM. While the costs of both algorithms are asymptotically the same, i.e., $O(n \log^2 n)$, ORAMMergeSort+ORAM appears to have a much high constant factor – up to five times slower than BitonicSort. As described in Section V-B2, we also applied the oblivious swap technique for within-block operations. Figure 12(b) shows the additional cost brought by BitonicSort+o-swap is significant.

Protecting Group Sizes. We have mentioned that by using

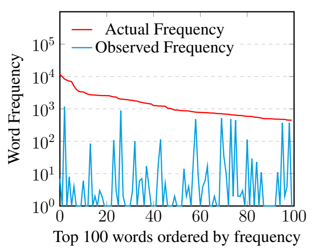


(a) Comparing oblivious sorting algorithms, block size = 1KB.

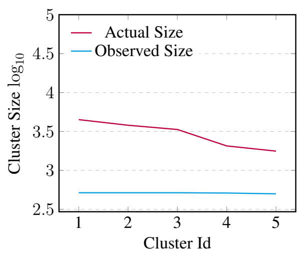


(b) Oblivious sorting with in-enclave CMOV protection, block size = 2KB.

Fig. 12: Costs of oblivious sorting solutions.



(a) Word frequencies protected by mandatory Combiner.



(b) KMeans cluster size protected by mandatory Combiner.

Fig. 13: Group size protection with mandatory Combiners.

Combiners mandatorily in our framework we can effectively protect from the group-size-estimation attack in reducing. Figure 13(a) and 13(b) show how well Combiners' outputs disguise the actual group sizes. It is well-known that combiners improve performance as they can significantly reduce the number of records going to the Reduce phase. We achieved about $\times 2$ and $\times 7$ speedup for WordCount and KMeans respectively. As our experiments showed, combiners are especially beneficial for applications with a small number of keys such as KMeans.

To further protect the possibly preserved group size ranking, we have also evaluated the dummy-record padding method, which impacts applications that require iterative processing. Table IV shows the oblivious post-processing cost to remove dummy records, which is necessary in iterative processing. The padding will significantly increase the execution time as we need an additional oblivious sorting step in the filtering step. However, with the post-processing step, the size goes back to the original one, which will not affect the performance of future processing.

TABLE IV: Additional costs for filtering padded records in iterative processing

Application	**Input size** records	**w/o filtering** (s)	**w/ filtering** (s)
KMeans	68 $\times 10^4$ vectors	20.0	39.2
WordCount	20,000 2KB blocks	194.6	381.8
PageRank (wiki)	170,918 edges	16.8	31.2

VII. Related Work

Side-channel attacks and mitigations. Side channel attacks on Intel SGX have been a hot research area during recent years. We will not intend to cover all the work in this area, as some surveys [27], [48] have done an excellent job. Instead, we focus on those studies that are most related to access-pattern protection for data-intensive SGX applications.

The most popular solutions are based on ORAM [65], [69], which has broader applications beyond SGX applications, e.g., including the setting of cloud-based client-server confidential computation. The most efficient ORAM schemes so far include tree-based Path ORAM [65] and Circuit ORAM [69]. ZeroTrace [59] experimented with both Path and Circuit ORAM to address the access-pattern problem for SGX applications. While using the most efficient implementation of ORAM, we show that ZeroTrace based block I/O still takes a significant overhead over the regular block I/O and much more expensive than TEE-MR. Obliviate [2] also uses the Path ORAM scheme to design the I/O interface for an SGX-based secure file system. Both schemes try to design a protection mechanism at the block I/O level, providing a high level of transparency to application developers. As we have discussed, an application-framework-level protection mechanism can be more efficient for data analytics applications.

TEE-enhanced big data systems. Researchers also try to extend big data processing platforms to take advantage of SGX. Although our method utilizes the famous big data processing framework, our purpose is entirely different from these studies. They follow the basic idea: to keep the codebase of current big-data systems: Hadoop and Spark unchanged as possible, while moving the data-processing parts to the SGX enclave. Access-pattern protection is their secondary goal. (1) VC3 [61] applied this strategy for modifying the Hadoop [70] system. It moves the execution of "map" and "reduce" functions to the SGX enclave, while the upper-level functions such as job scheduling and data management still stay outside the enclave. The most part of the Hadoop Java library is not changed at all. As a result, it achieves the goal of processing encrypted sensitive data in enclaves, but did not address the problems of access pattern protection and computation integrity (for the components running in the untrusted memory area). (2) M2R [23] targets the problem of access-pattern leakage in the shuffling phase of VC3 and proposes to use the oblivious schemes for shuffling. However, other security problems are still not addressed. (3) Opaque [76] tries to revise Spark for SGX. They focus on the data access patterns between computing nodes and illustrate how adversaries can utilize these access patterns to infer sensitive information in the encrypted data. To protect the distributed data access patterns, they provide four types of primitive oblivious operators to support the rich Spark processing functionalities: oblivious sort, filter, join, and aggregation. Noticing the problem of computation integrity, they try to move the job controller part, formerly in the master node, to the trusted client-side and design an integrity verification method to detect whether worker nodes process data honestly. However, to reuse the most parts of Spark codebase, it has the most of the system running in the untrusted area, especially for worker nodes. Thus, the local-level integrity guarantee and access-pattern protection might be insufficient.

These top-down approaches: VC3, M2R, and Opaque, have the fundamental problem – unless the whole framework is re-implemented and moved to the enclave, adversaries can easily attack the components running in the untrusted area or utilize the memory side-channels to extract confidential information. Our work is entirely different from these studies. Our goal is not to achieve massive parallel processing of data. Instead, we utilize the MapReduce framework to regulate application dataflow so that we can focus on the application-independent framework level protection mechanisms and thus protect a large number of data mining algorithms that can be cast to the processing framework.

Oblivious operators. Compared to ORAM at the block I/O level and TEE-MR at the framework level providing access pattern protection, Ohrimenko et al. [50] take an application-specific data-oblivious library approach. They analyzed several machine learning and data mining algorithms and extract a few primitive operations to make them data oblivious, including oblivious move, oblivious greater, and oblivious sorting. These primitives are then used to construct data-oblivious data analytics algorithms. Following the similar path, several works [60], [36], [35], [39] originate oblivious versions of primitive operators like move, swap, etc., by leveraging either CMOV instruction or boolean expressions. By using these oblivious operators as building blocks, they designed more complex algorithms. This approach can protect the application-specific access patterns, but it needs experienced developers to rewrite each algorithm carefully.

Access-patterns in enclave-enhanced databases. Other related work includes enclave-based database systems. Many of them have ignored the access pattern problem and focused more on efficiency and functionality of database systems, such as EnclaveDB [52], AlwaysEncrypted [6], and Enclage [66]. Several solutions have used improved ORAM as the basic building block to provide oblivious query processing, such as Oblix [46], ObliDB [25], and oblivious range and kNN query processing [17]. Oblivious join is the most expensive operation in oblivious database queries, which cannot be efficiently implemented with ORAM. Zheng et al. [76] proposed a non-ORAM primary-foreign key join algorithm and Krastnikov et al. [36] extended this method for general equi-joins. All of these studies are tailored to the access patterns in relational query processing, which are different from the batch-based data mining workloads that TEE-MR is designed for.

VIII. Conclusion

This paper proposes a framework-based approach to addressing the access-pattern protection problems in TEE applications. The proposed TEE-MR framework to regulate the application dataflow – once the framework-level access pattern problems are addressed, applications cast to the framework can all benefit from the systematic access-pattern protection mechanism. The regulated dataflow allows us to focus on the application-agnostic access pattern leakages at each stage and develop efficient solutions. As a result, the TEE-MR framework hides the details of access-pattern protection methods from developers and minimizes developers' work in developing TEE applications resilient to access-pattern attacks.

We have conducted extensive experiments to study the costs and performance advantages of the TEE-MR framework. The result shows that TEE-MR-based implementations of data-intensive applications work much more efficiently than most ORAM-based solutions. Although learning the MapReduce programming model may take a couple of weeks based on the reported studies, the advantage of using the framework in our approach is tremendous. They are also easier to develop and integrate into existing projects, as many algorithms have been implemented with MapReduce during the past years. Another unique benefit is that the framework-based implementation can be easily ported to different TEEs without significantly affecting the applications.

IX. Acknowledgment

This research was partially supported by National Institute of Health (Award# 1R43AI136357) and National Science Foundation (Award# 2232824). Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the funders.

References

[1] M. Agrawal, M. Zitnik, and J. Leskovec. Large-scale analysis of disease pathways in the human interactome. In *Biocomputing 2018*, pages 111–122, 2018.

[2] A. Ahmad, K. Kim, M. I. Sarfaraz, and B. Lee. OBLIVIATE: A data oblivious file system for intel SGX. In *the Network and Distributed System Security Symposium*, 2018.

[3] A. M. Alam, S. Sharma, and K. Chen. SGX-MR: Regulating dataflows for protecting access patterns of data-intensive sgx applications. *Proceedings on Privacy Enhancing Technologies*, 2021(1):5 – 20, 2021.

[4] Alibaba. Alibaba cloud's SGX encrypted computing environment. https://www.alibabacloud.com/help/en/elastic-compute-service/latest/build-an-sgx-encrypted-computing-environment/.

[5] AMD. AMD SEV-SNP: Strengthening vm isolation with integrity protection and more. https://www.amd.com/en/processors/amd-secure-encrypted-virtualization, 2020.

[6] P. Antonopoulos, A. Arasu, K. D. Singh, K. Eguro, N. Gupta, R. Jain, R. Kaushik, H. Kodavalla, D. Kossmann, N. Ogg, R. Ramamurthy, J. Szymaszek, J. Trimmer, K. Vaswani, R. Venkatesan, and M. Zwilling. Azure SQL database always encrypted. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, SIGMOD '20, page 1511–1525, New York, NY, USA, 2020. Association for Computing Machinery.

[7] K. E. Batcher. Sorting networks and their applications. In *Proceedings of the April 30–May 2, 1968, Spring Joint Computer Conference*, AFIPS '68 (Spring), pages 307–314, New York, NY, USA, 1968. ACM.

[8] M. Bellare, V. T. Hoang, and P. Rogaway. Foundations of garbled circuits. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, page 784–796, New York, NY, USA, 2012. Association for Computing Machinery.

[9] L. Biernacki, B. M. Tiruye, M. Z. Demissie, F. A. Andargie, B. Reagen, and T. Austin. Exploring the efficiency of data-oblivious programs. In *2023 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 189–200, 2023.

[10] Z. Brakerski and V. Vaikuntanathan. Fully homomorphic encryption from ring-lwe and security for key dependent messages. In *Proceedings of the 31st Annual Conference on Advances in Cryptology*, CRYPTO'11, pages 505–524, Berlin, Heidelberg, 2011. Springer-Verlag.

[11] F. Brasser, U. Müller, A. Dmitrienko, K. Kostiainen, S. Capkun, and A.-R. Sadeghi. Software grand exposure: SGX cache attacks are practical. In *11th USENIX Workshop on Offensive Technologies (WOOT 17)*, Vancouver, BC, 2017. USENIX Association.

[12] S. Brin and L. Page. The anatomy of a large-scale hypertextual web search engine. *Computer Networks*, 30:107–117, 1998.

[13] J. V. Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx. Foreshadow: Extracting the keys to the intel SGX kingdom with transient out-of-order execution. In *27th USENIX Security Symposium (USENIX Security 18)*, page 991–1008, Baltimore, MD, Aug. 2018. USENIX Association.

[14] J. V. Bulck, N. Weichbrodt, R. Kapitza, F. Piessens, and R. Strackx. Telling your secrets without page faults: Stealthy page table-based attacks on enclaved execution. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 1041–1056, Vancouver, BC, Aug. 2017. USENIX Association.

[15] N. Büscher, D. Demmler, S. Katzenbeisser, D. Kretzmer, and T. Schneider. HyCC: Compilation of hybrid protocols for practical secure computation. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 847–861, 2018.

[16] D. Cash, P. Grubbs, J. Perry, and T. Ristenpart. Leakage-abuse attacks against searchable encryption. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, page 668–679, New York, NY, USA, 2015. Association for Computing Machinery.

[17] Z. Chang, D. Xie, F. Li, J. M. Phillips, and R. Balasubramonian. Efficient oblivious query processing for range and kNN queries. *IEEE Transactions on Knowledge and Data Engineering*, pages 1–1, 2021.

[18] K. Chen, B. Wang, and P. Mateti. CUTE labs: Low-cost open-source instructional laboratories for cloud computing education. In *2016 ASEE Annual Conference and Exposition*, New Orleans, Louisiana, 2016. ASEE Conferences.

[19] C.-T. Chu, S. K. Kim, Y.-A. Lin, Y. Yu, G. Bradski, and A. Y. Ng. Map-reduce for machine learning on multicore. In *Proceedings Of Neural Information Processing Systems (NIPS)*, 2006.

[20] S. D. Constable and S. Chapin. libOblivious: A c++ library for oblivious data structures and algorithms. In *Electrical Engineering and Computer Science - Technical Reports. 184*, 2018.

[21] V. Costan and S. Devadas. Intel SGX explained. *IACR Cryptology ePrint Archive*, 2016:86, 2016.

[22] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI*, pages 137–150, 2004.

[23] T. T. A. Dinh, P. Saxena, E. Chang, B. C. Ooi, and C. Zhang. M2R: enabling stronger privacy in MapReduce computation. In *USENIX Security Symposium*, pages 447–462. USENIX Association, 2015.

[24] C. Dwork. Differential privacy. In *International Colloquium on Automata, Languages andProgramming*, pages 1–12. Springer, 2006.

[25] S. Eskandarian and M. Zaharia. ObliDB: Oblivious query processing for secure databases. *Proc. VLDB Endow.*, 13(2):169–183, oct 2019.

[26] R. Fagin, J. Nievergelt, N. Pippenger, and H. R. Strong. Extendible hashing – a fast access method for dynamic files. *ACM Transactions on Database Systems*, 4(3):315–344, 1979.

[27] S. Fei, Z. Yan, W. Ding, and H. Xie. Security vulnerabilities of SGX and countermeasures: A survey. *ACM Comput. Surv.*, 54(6), 2021.

[28] H. Gamaarachchi and H. Ganegoda. Power analysis based side channel attack. *CoRR*, abs/1801.00932, 2018.

[29] O. Goldreich and R. Ostrovsky. Software protection and simulation on oblivious ram. *Journal of the ACM*, 43:431–473, 1996.

[30] J. Götzfried, M. Eckert, S. Schinzel, and T. Müller. Cache attacks on intel SGX. In *Proceedings of the 10th European Workshop on Systems Security*, EuroSec'17, pages 2:1–2:6, New York, NY, USA, 2017. ACM.

[31] D. A. Heath. *New Directions in Garbled Circuits*. PhD thesis, Georgia Institute of Technology, 2022.

[32] Y. Huang, D. Evans, J. Katz, and L. Malka. Faster secure two-party computation using garbled circuits. In *USENIX Conference on Security*, pages 35–35, 2011.

[33] A. Jain, M. Murty, and P. Flynn. Data clustering: A review. *ACM Computing Surveys*, 31:264–323, 1999.

[34] T. Kim, J. Park, J. Woo, S. Jeon, and J. Huh. ShieldStore: Shielded in-memory key-value storage with SGX. In *Proceedings of the Fourteenth EuroSys Conference 2019*, EuroSys '19, New York, NY, USA, 2019. Association for Computing Machinery.

[35] S. Krastnikov, F. Kerschbaum, and D. Stebila. Efficient oblivious database joins. *arXiv preprint arXiv:2003.09481*, 2020.

[36] S. Krastnikov, F. Kerschbaum, and D. Stebila. Efficient oblivious database joins. *Proc. VLDB Endow.*, 13(12):2132–2145, jul 2020.

[37] J. Lee, J. Jang, Y. Jang, N. Kwak, Y. Choi, C. Choi, T. Kim, M. Peinado, and B. B. Kang. Hacking in darkness: Return-oriented programming against secure enclaves. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 523–539, Vancouver, BC, 2017.

[38] M. Li. *Understanding and Exploiting Design Flaws of AMD Secure Encrypted Virtualization*. PhD thesis, The Ohio State University, The Ohio State University, 2022.

[39] M. Li, Y. Zhang, Z. Lin, and Y. Solihin. Exploiting unprotected i/o operations in amd's secure encrypted virtualization. In *Proceedings of the 28th USENIX Conference on Security Symposium*, SEC'19, page 1257–1272, USA, 2019. USENIX Association.

[40] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg. Meltdown: Reading kernel memory from user space. In *27th USENIX Security Symposium (USENIX Security 18)*, 2018.

[41] C. Liu, X. S. Wang, K. Nayak, Y. Huang, and E. Shi. ObliVM: A programming framework for secure computation. In *2015 IEEE Symposium on Security and Privacy*, pages 359–376, 2015.

[42] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee. Last-level cache side-channel attacks are practical. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy*, SP '15, page 605–622, USA, 2015. IEEE Computer Society.

[43] D. Lyubimov and A. Palumbo. *Apache Mahout: Beyond MapReduce*. CreateSpace Independent Publishing Platform, 2016.

[44] Microsoft. Microsoft azure database. https://learn.microsoft.com/en-us/azure/azure-sql/database/always-encrypted-enclaves-enable-sgx?view=azuresql/.

[45] D. Miner and A. Shook. *MapReduce Design Patterns: Building Effective Algorithms and Analytics for Hadoop and Other Systems*. O'Reilly Media, 2012.

[46] P. Mishra, R. Poddar, J. Chen, A. Chiesa, and R. A. Popa. Oblix: An efficient oblivious search index. In *IEEE Symposium on Security and Privacy*, pages 279–296. IEEE Computer Society, 2018.

[47] P. Mohassel and Y. Zhang. SecureML: A system for scalable privacy-preserving machine learning. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 19–38, 2017.

[48] A. Nilsson, P. N. Bideh, and J. Brorsson. A survey of published attacks on intel SGX. *ArXiv*, abs/2006.13598, 2020.

[49] O. Ohrimenko, M. Costa, C. Fournet, C. Gkantsidis, M. Kohlweiss, and D. Sharma. Observing and preventing leakage in MapReduce. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, CCS'15, page 1570–1581, New York, NY, USA, 2015. Association for Computing Machinery.

[50] O. Ohrimenko, F. Schuster, C. Fournet, A. Mehta, S. Nowozin, K. Vaswani, and M. Costa. Oblivious multi-party machine learning on trusted processors. In *USENIX Security Symposium*, pages 619–636. USENIX Association, 2016.

[51] A. Ozdemir, F. Brown, and R. S. Wahby. Circ: Compiler infrastructure for proof systems, software verification, and more. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 2248–2266. IEEE, 2022.

[52] C. Priebe, K. Vaswani, and M. Costa. EnclaveDB - a secure database using SGX. In *IEEE Symposium on Security and Privacy*. IEEE, May 2018.

[53] A. S. Rabkin, C. Reiss, R. Katz, and D. Patterson. Experiences teaching mapreduce in the cloud. In *Proceedings of SIGCSE*, SIGCSE '12, page 601–606, New York, NY, USA, 2012. Association for Computing Machinery.

[54] A. Rane, C. Lin, and M. Tiwari. Raccoon: Closing digital side-channels through obfuscated execution. In *Proceedings of the 24th USENIX Conference on Security Symposium*, SEC'15, page 431–446, USA, 2015. USENIX Association.

[55] R. A. Rossi and N. K. Ahmed. The network data repository with interactive graph analytics and visualization. In *AAAI*, 2015.

[56] I. Roy, S. T. V. Setty, A. Kilzer, V. Shmatikov, and E. Witchel. Airavat: Security and privacy for mapreduce. In *USENIX Conference on Networked Systems Design and Implementation*, 2010.

[57] B. Rozemberczki, C. Allen, and R. Sarkar. Multi-scale attributed node embedding, 2019.

[58] S. Sagar and C. Keke. Confidential machine learning on untrusted platforms: a survey. *Cybersecurity*, 4(1):30, 2021.

[59] S. Sasy, S. Gorbunov, and C. W. Fletcher. Zerotrace : Oblivious memory primitives from intel SGX. In *Network and Distributed System Security Symposium*, 2018.

[60] S. Sasy, A. Johnson, and I. Goldberg. Fast fully oblivious compaction and shuffling. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, pages 2565–2579, 2022.

[61] F. Schuster, M. Costa, C. Fournet, C. Gkantsidis, M. Peinado, G. Mainar-Ruiz, and M. Russinovich. VC3: Trustworthy data analytics in the cloud using SGX. In *36th IEEE Symposium on Security and Privacy*, 2015.

[62] S. Shinde, Z. L. Chua, V. Narayanan, and P. Saxena. Preventing page faults from telling your secrets. In *Proceedings of ASIACCS*, page 317–328, New York, NY, USA, 2016. ACM.

[63] R. L. Simon. *Fair play: The ethics of sport.* Routledge, 2018.

[64] J. Son, G. Prechter, R. Poddar, R. A. Popa, and K. Sen. ObliCheck: Efficient verification of oblivious algorithms with unobservable state. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 2219–2236. USENIX Association, 2021.

[65] E. Stefanov, M. V. Dijk, E. Shi, T.-H. H. Chan, C. Fletcher, L. Ren, X. Yu, and S. Devadas. Path ORAM: An extremely simple oblivious ram protocol. *Journal of the ACM*, 65(4), Apr. 2018.

[66] Y. Sun, S. Wang, H. Li, and F. Li. Building enclave-native storage engines for practical encrypted databases. *Proc. VLDB Endow.*, 14(6):1019–1032, feb 2021.

[67] S. Thornton. Arm trustzone explained. http://alturl.com/icptx, 2017.

[68] J. Van Bulck, D. Moghimi, M. Schwarz, M. Lipp, M. Minkin, D. Genkin, Y. Yuval, B. Sunar, D. Gruss, and F. Piessens. LVI: Hijacking Transient Execution through Microarchitectural Load Value Injection. In *41th IEEE Symposium on Security and Privacy (S&P'20)*, 2020.

[69] X. Wang, H. Chan, and E. Shi. Circuit ORAM: On tightness of the goldreich-ostrovsky lower bound. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, page 850–861, New York, NY, USA, 2015. Association for Computing Machinery.

[70] T. White. *Hadoop: The Definitive Guide*. O'Reilly Media, 2009.

[71] Y. Xu, W. Cui, and M. Peinado. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy*, SP '15, pages 640–656, Washington, DC, USA, 2015. IEEE Computer Society.

[72] Y. Yarom and K. Falkner. Flush+reload: A high resolution, low noise, l3 cache side-channel attack. In *Proceedings of the 23rd USENIX Conference on Security Symposium*, SEC'14, page 719–732, USA, 2014. USENIX Association.

[73] S. Zahur and D. Evans. Obliv-c: A language for extensible data-oblivious computation. Cryptology ePrint Archive, 2015.

[74] K. Zhang, Z. Li, R. Wang, X. Wang, and S. Chen. Sidebuster: automated detection and quantification of side-channel leaks in web application development. In *Proceedings of the 17th ACM conference on Computer and communications security*, pages 595–606, 2010.

[75] Y. Zhang, J. Katz, and C. Papamanthou. All your queries are belong to us: The power of file-injection attacks on searchable encryption. In *Proceedings of the 25th USENIX Conference on Security Symposium*, SEC'16, page 707–720, USA, 2016. USENIX Association.

[76] W. Zheng, A. Dave, J. G. Beekman, R. A. Popa, J. E. Gonzalez, and I. Stoica. Opaque: An oblivious and encrypted distributed analytics platform. In *USENIX Symposium on Networked Systems Design and Implementation*, 2017.

,

Appendix A
Sample ORAM-Hash algorithms

Algorithm 2 Buffer Management for ORAM-Hash algorithms

```
1: Buffer contains a working block $B$ for new records, and
   a cache of $m$ blocks $C$.
2: **Function** GetBlock($block\_id$)
3: **if** $block\_id$ **not** in the cache $C$ **then**
4:     decide a block to overwrite with an algorithm like LRU;
       the victim block is written back to the output file.
5:     $new\_block \leftarrow request\_oram\_block(block\_id)$
6:     add $new\_block$ to the cache
7: **end if**
8: $block\_reference \leftarrow$ find the $block\_id$ in the cache
9: **return** $block\_reference$

1: **Function** AddRecordToBlock($record$)
2: **if** working block $B$ is **full then**
3:     evict LRU and write out the victim block
4:     copy $B$ to the cache
5:     clear the working block
6: **end if**
7: add $record$ to the working block
8: **return** $working\_block\_id$
```

Algorithm 3 HashMap-based WordCount in Enclave

```
1: **Function** WordCount($input\_file, output\_file$)
2: $h \leftarrow$ a HashMap to maintain (output-key, block-id)
3: initialize $ORAM$ and a buffer
4: **for** all $block$ in $input\_file$ **do**
5:     $words \leftarrow$ parse_words($block$)
6:     $word\_freq \leftarrow$ local_frequencies($words$)
7:     **for** all ($word$, $freq$) in $word\_freq$ **do**
8:         **if** $word$ not **not** in the HashMap $h$ **then**
9:             $record \leftarrow$ ($word$, $freq$)
10:            $id \leftarrow$ add_record_to_block($word$)
11:            $h[word] \leftarrow id$
12:        **else**
13:            $id \leftarrow h[word]$
14:            $block \leftarrow$ get_block_from_buffer($id$)
15:            update $block$ with ($word$, $freq$)
16:        **end if**
17:    **end for**
18: **end for**
19: flush all blocks in the buffer via $ORAM$ to $output\_file$
```

Algorithm 4 HashMap-based KMeans in Enclave

```
1: Function KMeans($centroid\_file, coordinates\_file$)
2: initialize $ORAM$ and $Cache$ block
3: load initial $centroids$ from $centroid\_file$
4: for all $block$ in $coordinates\_file$ do
5:   $points$ ← ParseCoordinates($block$)
6:   for each $pt$ in $points$ do
7:     $centroid\_index$ ← FindNearestCentroid($pt$, $centroids$)
8:     LocalMap[ $centroid\_index$ ] ← $pt$
9:   end for
10:  $combined\_points$ ← aggregate points under same centroid
11:  for each ($centroid\_index$, $point$) in $combined\_points$ do
12:    if $centroid\_index$ not not in $HashMap$ then
13:      $record$ ← ($centroid\_index$, $point$)
14:      $id$ ← AddRecordToBlock($centroid\_index$)
15:      $HashMap$[centroid_index] ← $id$
16:    else
17:      $id$ ← $HashMap$[centroid_index]
18:      $block\_reference$ ← GetBlock($id$)
19:      update block in $block\_reference$ with ($centroid\_index$, $point$)
20:      Write($block$)
21:    end if
22:  end for
23: end for
24: write all blocks from $ORAM$ to $centroid\_file$
```