Scaling Memory Access-Pattern Secure Computations via Dynamic Program Partitioning

Ju Chen
Syracuse University
Syracuse, NY 13244
jchen133@syr.edu

Cheng Xu [*]
Hong Kong Baptist University
Kowloon Tong, Hong Kong
chengxu@comp.hkbu.edu.hk

Kai Li
Syracuse University
Syracuse, NY 13244
kli111@syr.edu

Yuzhe (Richard) Tang
Syracuse University
Syracuse, NY 13244
ytang100@syr.edu

etc.
University
Place
username@email.edu

ABSTRACT

With the advent of commercial TEEs, defending memory-access pattern attacks becomes a pressing issue and is crucial to the wide adoption of these security platforms. The key challenges lie on two aspects: strong security against memory-access pattern attacks of various and emerging forms (e.g., cache-timing attacks, page-fault attacks, page-access attacks, etc.) and low performance overhead.

In this work, we propose a security notion, cache-miss obliviousness (CMO), which requires the trace of cache misses in a program execution to be irrelevant (or "oblivious") to secret data. For strong security, CMO intends to defend the bus-tapping attack, the strongest form of all memory-access pattern attacks. For efficiency, CMO allows secret-dependent computation inside CPU and allows for a broad class of external-oblivious computations with asymptotic efficiency. To implement CMO, we use Intel TSX to enforce the condition that no cache-miss occurs.

Technically, we study the dynamic program partitioning problem, that is, how to leverage runtime information to automatically partition the program execution to TSX transactions. This problem is different from static TSX-transaction partitioning (e.g., T-SGX) and manual partitioning (e.g., Cloak). We propose a security-oriented partitioning strategy that splits only the group of non-leaky data accesses and keep leaky accesses in their entirety.

We implement a library for programming support of CMO: First, our library exposes API for developers to specify the scope of leaky code section and data types accessed in TSX transactions. Second, we implement the proposed dynamic-partitioning strategy by closely monitoring program executions as triggers, and by inserting transactional boundaries with double data preloading for high success rate. Third, we optimize the performance by amortizing transactional cost and enlarging transaction size with effective data-cache utilization.

We evaluate our CMO-based library system in security, computation expressiveness, and performance. Using our library, we implement various computations in the field of data analytics and cryptographic operations. Our performance study shows that comparing existing work, our CMO-based library achieves speedup of multi-order of magnitudes and better data scalability.

1. INTRODUCTION

The modern architecture of trusted execution environment (TEE) has been commercially available, notably Intel Software Guard eXtension (SGX [3]) and ARM TrustZone [1], and has recently been adopted in the cloud services, such as Google Cloud Platform [2] and Microsoft Azure [4]. These TEEs supports the architecture of a "hardware enclave" for secure and trustworthy execution of programs on an untrusted third-party host, such as Cloud. Despite the security-oriented design (e.g., encrypted memory, CPU access control), existing TEEs are vulnerable to one major type of side-channel attacks – memory-access pattern attacks [38, 13, 26, 36].

To mitigate memory-access pattern attacks on SGX, existing work either closes the leakage channel by detecting the presence of a memory-access sniffer [33, 22, 13] or makes the disclosed access trace irrelevant to sensitive information (i.e., data obliviousness [39, 29, 32]). The mitigation by attack detection relies on some hardware capabilities, such as Intel TSX. This approach limits resource sharing and prohibits data movement across trust boundary (G1). The mitigation by data-obliviousness [39, 31, 18] incurs unacceptable high performance overhead (G2). Therefore, existing work is limited to processing a small volume of data in practice, as acknowledged in [22, 39]. While this limitation may be irrelevant for conventional applications of cryptographic key management [24] where the security-sensitive data is small keys, it presents a major barrier when one wants to support access-secure big-data applications on the cloud.

In this work, we tackle the open challenge of supporting secure data-scalable computations on Intel SGX. For security, our goal is to ensure strong security against all memory access-pattern attacks. We consider the strongest form of these attacks, that is, bus-tapping attacks, and our security goal is set to eliminate any memory-access pattern attacks, either software attacks or hardware attacks, as long as the CPU is trusted (a typical setting in TEE). For data scalability, our goal is to scale the computation with arbitrary data size and with efficiency (see § **??** for impossibility results and the specific goal of this work). Supporting both goals is lacking in existing research work as described.

In this work, we propose a security-performance notion, cache-miss obliviousness (CMO). Informally, it requires that the sequence

[*] The work is done when the author visits Syracuse Univeristy.

Conference name Year

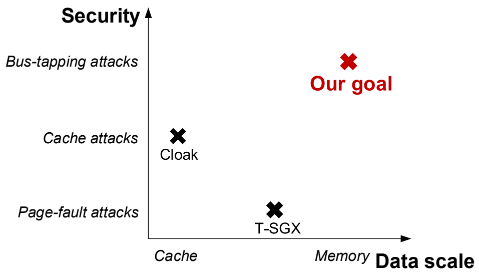


Figure 1: System-desing goals in security and data scale

of cache misses in executing a program is oblivious to sensitive data, thus safe to be disclosed (to a bus-tapping adversary). The rest of memory accesses are resolved in the trusted CPU (namely cache hits and register accesses) and can afford to be secret-dependent. With cache-miss obliviousness, our observation is that the security to bus-tapping attacks can be attained, and performance efficiency can be retained due to that it allows for secret-dependent computations and algorithm design with asymptotic efficiency (See § 4.2.1 for details).

Engineering CMO in software presents a non-trivial task. To start with, we focus on the class of external-oblivious computations [**?**]. The common computing paradigm is that the data is stored 1) in a small internal memory with leaky access and 2) in a much larger external memory with non-leaky access (see § 6 for a list of examples). Externally oblivious computations have better asymptotic performance other than oblivious mechanisms.

First, for expressing CM-oblivious computations, we propose a programming interface for expert developers to annotate the program and to distinguish the code section with leaky data access and that without. It annotates the begin and end of leaky code sections and data types accessed in TSX transactions.

Second, we propose a program-partitioning framework that splits the program execution and wraps them in TSX transactions. The use of TSX transactions is to ensure that there is no occurence of cache misses during the program execution (which is similar to prior works [22, 14, 15]). However, we uniquely study the dynamic partitioning problem that partitions the program based on various runtime information. For security, our scheme only partitions the non-leaky data accesses across transactions, while keeping the leaky data accesses atomic. To implement the dynamic-partitioning scheme, we build a library that monitors program execution at runtime, looking for triggering conditions of partitioning. When conditions are met, it partitions the computation by inserting TSX instructions at transaction boundaries.

Third, we optimize the performance by enlarging the transaction size. The observation is that the larger a transaction is, the more instructions the transactional setup cost can be amortized to. To ensure the successful execution of large transactions, we propose to arrange the cache data usage based on the data-access type; the intuition is that different data types in a transaction are bounded by different cache conditions. For instance, read-only transaction data can reside in both last-level cache and level-one cache, whereas read-write transactional data must reside in level-one cache.

To evaluate the expressiveness of our scheme, we build programs for a variety of common computations, including sorting, shuffle, binary search, k-means, etc. To evaluate the performance and data scalability, we measure the execution time of the computation.

2. RELATED WORK

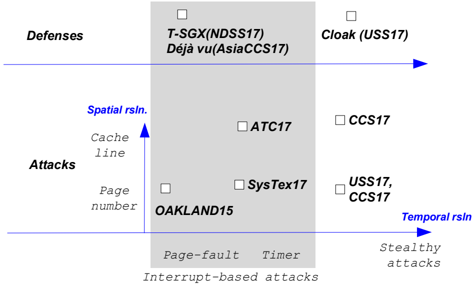


Figure 2: Taxonomy of memory-access attacks and defenses in SGX: Existing attacks shown in the diagram are code-named by the publication venue and year, including USS17 (USENIX Security) [13], CCS17 [36], SysTex17 [12], ATC17 [23], OAKLAND15 [38]. Existing defenses shown in the diagram are code-named similarly and with their name, including T-SGX [33], Deja Vu [16], and Cloak [22].

2.1 Preliminary

TEE and memory access-pattern attacks: The architecture of SGX-alike TEE features a trusted world (a.k.a. enclave) and untrusted world sharing a processor, the boundary between which is protected by some isolation mechanisms. The main memory usually resides in the untrusted world where the content is protected by hardware encryption. In this TEE architecture, a memory reference has to access a series of micro-architectural "buffers" that are shared, such as last-level cache (LLC), translation look-aside buffer (TLB), memory row buffer, etc., creating leakage side channels.

A passive memory-access attacker monitors the access trace of an enclave execution and infers sensitive information. To monitor, the attacker *breaks* enclave execution and *observes* some micro-architectural side-channel. Existing memory-access attacks can be characterized by the spatial resolution of the leakage channel being observed and the temporal resolution of breaking the enclave control flow. For instance, in the seminal work [38], a controlled page-fault attacker both breaks and observes the enclave execution through page-fault interrupts, where the the trace spatial granularity is at page numbers and the temporal granularity is individual page accesses. More advanced attacks exploit other leakage channels (e.g. page-table access/dirty bits [13, 36], shared last-level caches [23]) to improve spatial and temporal resolution. To make the defense more difficult, recent *stealthy* attacks break the enclave execution invisibly through hyper-threading that eliminate interrupts [22, 36].

Data-oblivious algorithms: The conventional oblivious algorithms ensure oblivious data access at the "word" granularity. These *word-oblivious algorithms (A1)* are constructed based on primitives such as compare-exchanges and tend to be not very practical (e.g., AKS sorting [6] and sub-optimal column sort [8]). In the external-memory model, **external oblivious algorithms** improve the time/IO complexity by assuming a small amount of internal storage that is accessed in a leaky fashion. Due to a smaller constant and optimal complexity, this family of algorithms are of great practical interest. For instance, Melbourne shuffle [30] causes $O(\sqrt{N})$ IO with $O(N \log N)$ time at the expense of $\sqrt{N}$ internal space and was recently used in constructing practical systems (e.g., MapReduce [29]). Research to improve the algorithmic "locality" and to further reduce the internal-space complexity is underway [28]. The *oblivious RAM (A3)* [35, 20] is an external data structure supporting oblivious external data access by translating virtual random-access to oblivious physical data access with poly-logarithmic blowup.

Comparing ORAM (A3) and word-oblivious algorithms (A1), the external oblivious algorithms (A2) cause lower overhead in complexity. Comparing generic ORAM, external oblivious algorithms are computation specific; existing algorithms support sort, compaction, and relational queries and other batched-oriented computations [30, 37, 21, 7].

2.2 Existing Work on Access-Pattern Secure Systems

Memory-access attack mitigation: A successful memory-access attack entails two conditions: 1) an observable channel of memory-access leakage, and 2) the capability of relating the leaked memory-access to sensitive information. A memory-access attack can be mitigated by breaking either condition: M1) Detecting that a leakage channel is being observed by *enclave fortification*, or M2) Eliminating the source of leakage by making it difficult to correlate the access trace with sensitive information, via *obliviousness*.

For the former, existing enclave-fortification work detects the attack on the events of page-fault [34, 33], cache-miss [22, 11], etc.

For the latter, existing work on **data-oblivious systems** is based on oblivious algorithms [39, 29] or less-efficient oblivious RAM [32, 19]. Concretely, data obliviousness can eliminate the source of leakage in the presence of memory-access attacks on TEE. When building data-oblivious systems, a design choice is which oblivious-computing mechanisms to use: Opaque [39] supports database queries by engineering word-oblivious algorithms (Column Sort [8]). Oblivious machine-learning [31] combines word-oblivious algorithms and the trivial mechanism of converting random-access to a full-array scan. Scan-based transformation is similarly adopted in building ZeroTrace [32], an ORAM system in SGX. ObliVM [27] is a compiler that supports the engineering of both word-oblivious algorithms and ORAM [35, 20]. Oblivious Map-Reduce [29] and Prochlo [10] use external oblivious shuffle [30] to protect distributed systems under traffic analysis, but insecure under micro-architectural memory-access attacks.

3. RESEARCH FORMULATION

3.1 System Model

System model: In this work, we consider the modern computing paradigm that a security-sensitive computation is outsourced and run on a third-party host. This paradigm becomes increasingly popular due to the advent of cloud computing which has the advantage in cost effectiveness, availability, etc. Concretely, the data owner possessing security-sensitive personal data uploads the data and program to a third-part host in the cloud where the program is executed over the data to evaluate the computation on the owner's behalf.

The host in our system has an SGX-alike trusted execution environment, or enclave. The enclave is trusted by the data owner, which includes only the processor at the hardware layer and the owner-provided program at the software layer. Hardware and software outside this perimeter of the enclave is untrusted by the data owner.

The target computation.

of this work is what we call by external-oblivious computation. The working set of this computation consists of a large volume of data accessed in a non-leaky fashion (namely, non-leaky data) and a small volume of data accessed in a leaky fashion (namely, access-leaky data). The small volume of access-leaky data is defined to be cache resident.

In practice, we deem there are a large number of real-world applications that meet the external-oblivious computations, especially in data-analytical applications. For instance, streaming applications feature a big-data stream that is accessed in a non-oblivious fashion, while keeping a small state that is accessed randomly (or in a leaky fashion). In other data-analytical computations,[1] a common paradigm is to iterate through the dataset multiple iterations, each of which runs a stateful computation with a data pass through the dataset. Even for computations that are not natively external-oblivious, such as sorting (quick sort or merge sort are not oblivious), there are external oblivious algorithms, such as Melbourne shuffle [30], oblivious merge sorts [37]. We conduct a study of a series of related cases and support sample external-oblivious computations in § 6.

Out of the scope is the computation that randomly accesses a large external dataset, such as database search (with index); these computations can be supported by existing work such as ZeroTrace [32] that engineers an ORAM in external storage.

3.2 Threat Model

This work considers memory-access pattern attacks in the emerging hardware enclaves. In the target system, a program runs inside an enclave of trusted processor and encrypted memory, and the adversary monitors certain micro-architectural side-channel during the enclave execution. Based on the observed access trace, she then infers other sensitive information. The exploited leakage channels include page-fault [38], page-table access/dirty bits [26, 13], cache-timing [36, 23]. Access-driven attacks against data-intensive computations can lead to significant information disclosure [29, 39, 9]. For clarity, the non-goals of this work include other side-channel attacks (e.g., timing attacks, power analysis), denial-of-service attacks, rollback attacks, etc.

3.3 Security Definition: Cache-Miss Obliviousness

Intuitively, the cache-miss obliviousness or CMO requires that the trace of cache-misses (as exposed in bus-tapping attacks) in a program execution should be independent with any data of the execution (both input and intermediate data). Consider the execution of a program P with data I. The execution produces trace T consisting of last-level cache misses. CMO requires that given two data values, I_0 and I_1, a CM-oblivious execution produces two identical traces, that is, $T_P(I_0) = T_P(I_1)$. This definition assumes that the computation induced by P is deterministic. And the definition of obliviousness is "perfect" as it requires the two traces of different data are exactly the same. We skip the more formal and generic definition of cache-miss obliviousness (e.g. based on indistinguishability formation [24]).

3.4 Research Problem and Distinction

The focus of this work is the support of efficient data-intensive computations (in § 3.1) with cache-miss obliviousness (in § 3.3) on the TEE platforms (in § 3.1). Particularly, the cache-miss obliviousness ensures the memory-access security under cache-timing attacks while retaining the efficiency of external-oblivious algorithms.

This goal presents an open research problem: Existing research on cache-timing attack detection (e.g. by HTM) is limited in data scalability as the working set must be cache-resident. Existing research on data-obliviousness engineering largely focuses on word-oblivious algorithms (e.g., Opaque [39], ObliviousML [31]). The limited work considering the more efficient external-oblivious algorithms (e.g., ZeroTrace [32], ObliviousMR [29], Prochlo [10])

[1] Data analytical algorithms usually have super-linear time complexity and need to access data multiple iterations.

has either sub-par security or sub-optimal performance efficiency.

We survey existing research from the angles of strong cache-attack security, data scalability, and algorithmic complexity. The research distinction of our work is presented in Table 1.

3.5 Proposed Method

Our method for engineering external obliviousness is that given the target computation expressed by an external-oblivious algorithm, we place and only place the part of program accessing internal memory in external-oblivious algorithms to the HTM transactions. By this means, the cache-misses which only occur outside the internal program are oblivious, while cache-hits and other internal accesses can be made secure by HTM protection.

The motivation to combine HTM and external oblivious algorithms can be better illustrated by an example: Consider implementing a sort on Intel SGX with cache-attack security. The first baseline (BL1) entails implementing the word-oblivious sorting network at the expense of a logarithmic multiplicative factor (i.e., $O(N \log^2 N)$). The second baseline is to use HTM to protect the leaky accesses in a classic sorting algorithm of $O(N \log^2 N)$ complexity, such as quick sort or merge sort. This approach limits the data size by CPU cache size. In a big-data setting (with large N), either approach is feasible as the former is inefficient and the latter is unscalable. By contrast, our proposed method combining HTM and external oblivious algorithms could lead to an optimal $O(N \log N)$ sorting algorithm (based on the scramble-then-shuffle paradigm [17]) implemented with cache-attack security.

Note that our approach shares some general idea with Cloak and T-SGX in the sense that it leverages Intel TSX to detect the memory-access attacks. However, Cloak does not automatically partition the computation and is limited to small-data case. T-SGX is a static program-partition scheme, which limits its applicability in handling loops and thus big-data computations. Our work uniquely addresses the problem of dynamically partitioning the programs to support big-data computations with access-pattern security.

4. SYSTEM OVERVIEW AND API

4.1 System Overview

Our goal is to build a data-intensive runtime system with access-pattern security on hardware enclaves. Concretely, given an external oblivious algorithm, we provide an API (S1) for developers to annotate the leaky section of the program. Given the annotated program, we provide an execution engine (S2) runs the program on architecture of two execution environments: the one with access-pattern protection (e.g., prohibiting cache-misses) and the other without. We call the former by fortified enclave, and the latter by unprotected enclave. Briefly, the execution engine partitions the runtime instance dynamically in a way that assures no cache-miss yet can scale to relatively large data. Figure 3 illustrates the overview of our system. In the following, we present the detailed design and implementation of our system in each component, S1 in § 4.2.2 and S2 in § 5.

4.2 External-Obliviousness API

4.2.1 Preliminary: External-Obliviousness Algorithms

External oblivious algorithms are a class of oblivious mechanisms that has advantageous time/round complexity at the expenses of assuming a trusted internal memory. External oblivious algorithms are proposed for a wide variety of computations, including shuffle (Melbourne shuffle [30]), sort (oblivious merge sort [37]), most aggregation computations, etc. Comparing word-oblivious algorithms (e.g., sorting networks causing a multiplicative factor of $O(\log N)$), an external oblivious algorithm has better time complexity (e.g., Melbourne shuffle based sorting [29, 17] with the O(1) multiplicative factor). Comparing ORAM (with a multiplicative factor of $O(\log^2 N)$), an external oblivious algorithm is specific to target computation and is more efficient. While external oblivious algorithms achieve better complexity, it may present limitation on data scalability. Concretely, most of existing external oblivious algorithms are designed for a client-server setting, and assume a large internal memory at a client side. For instance, a Melbourne shuffle assumes an internal memory of $O(\sqrt{N})$. The large internal memory may be suited for a client machine (in the big-data setting), but presents challenges when treating cache as internal memory, whose size is very limited; for instance, a L1 cache is 32 KB and L3 cache is 8 MB in an Intel SGX CPU. Very recently, space-efficient external oblivious algorithms are proposed, for instance, stash shuffle with small $O(\log N)$ internal space.

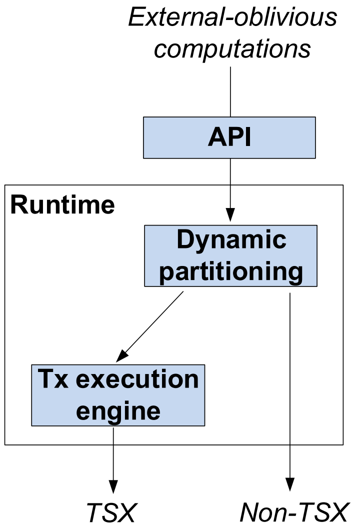


Figure 3: System overview: A target computation is annotated by our programming API that specifies data-access pattern. Our runtime system first partitions the program dynamically to units (i.e., dynamic partitioning), and then runs each unit in an TSX transaction (i.e., tx-execution engine).

4.2.2 API

Our system provides a programming interface for developers to annotate the leaky section of the program and define the type of data referenced inside the leaky section. Specifically, 1) we provide two library functions, `begin_leaky()` and `end_leaky()`, for developers to declare the begin and end of a program snippet where leaky memory access occurs. 2) In addition, all the memory data accessed inside a leaky section needs to be defined with our data type. We provide four data-container types as defined below. The explicit data type allows our execution engine to be know ahead of time the possible data-access pattern and take action in partitioning more precisely.

The data-container types are classified based on whether the access is leaky and whether the data is read-only. The four data types are `NobRO`, `ObRO`, `NobRW`, `ObRW`, where `ObRW` stands for oblivious read-write data container and NobRO is non-oblivious read-only data container. The API of the data classes are provided below. We also provide an example code that declares the merge passes in

Table 1: Research distinction

	Method	Cache access-pattern security	Data scalability	Algorithmic complexity
T-SGX [33], Cloak [22]	Attack detection by HTM	+	-	+
Opaque [39], ObliviousML [31]	Word-oblivious algo.	+	+	-
ObliviousMR [29],Prochlo [10]	External oblivious algo.	-	+	+
ZeroTrace [32]	External ORAM	+	+	-
This work	External oblivious algo./HTM	+	+	+

a merge sort algorithm to be leaky sections.

```
class ObRO {
  int32_t read_next();
  void reset();
}
class ObRW {
  void write_next(int32_t data);
  void reset();
}
class NobRW {
  int32_t read_at(int32_t addr);
  void write_at(int32_t addr, int32_t data);
}
class NobRO {
  int32_t nob_read_at(int32_t addr);
}
```

```
void MergeSort(int[] array, int start, int end){
  if (start == end - 1) return;
  MergeSort(array, start, (end + start)/2);
  MergeSort(array, (end + start)/2, end);
  Merge(array, 0, (end + start)/2, (end + start)/2, end);
}
void Merge(int[] array, int start1, int end1, int start2,
     int end2){
  NobRW parray = new NobRW(array);
  begin_leaky();
  originalMerge(parray, start1, end1, start2, end2);
  end_leaky();
}
```

Figure 4: title

5. DYNAMIC PROGRAM PARTITIONING

5.1 Motivation

To secure a computation against memory-access pattern attacks, existing work leverages HTM features to detect any unintended cache misses. Notably, Cloak [22] detects cache attacks using Intel TSX. Briefly, it executes the computation in HTM transactions, such that the intra-transaction memory accesses are resolved by cache hits and thus are concealed from the adversary. Note that the inter-transaction memory accesses are disclosed. The security of these Cloak alike schemes requires that access-leaky memory region must reside in data cache; this security requirement significantly limits the data scale. In addition, Cloak's programming interface is rather low-level and requires programmer to manually specify the scope of individual HTM transactions, which further limit its applicability in big-data computations.

To systematically improve the data scalability, we study two technical problems: (O1) dynamic program partitioning and (O2) enlarging transaction. First, given a target computation, we dynamically partition the program to smaller execution units, each run in an individual transaction. Comparing existing work, such as Cloak [22] and T-SGX [33], our unique perspective about program partitioning is to take into account various runtime information (see § 3.5 for detailed comparison).

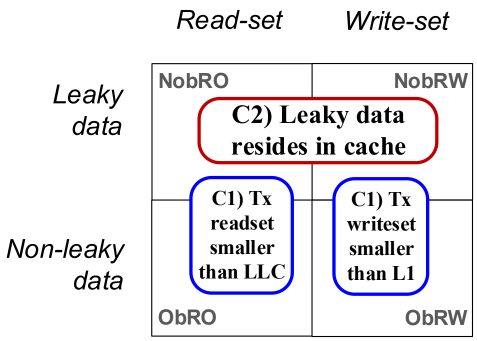

Read-set
Write-set
Leaky data
Non-leaky data
NobRO
NobRW
C2) Leaky data resides in cache
C1) Tx readset smaller than LLC
C1) Tx writeset smaller than L1
ObRO
ObRW

Figure 5: Transactional data constraints imposed by hardware (C1) and security (C2) requirements. The data is classified to four types based on read/write and leaky/non-leaky.

Second, for each execution unit, we enlarge the transaction as much as possible. Here, the larger a transaction is, the performance efficiency it can be, as the transaction setup cost (e.g., running xbegin/xend instructions) can be amortized among more instructions. Note that the size of any TSX transaction has a theoretic limit in that its working set cannot exceed a CPU data cache. Our goal is to reach this theoretic bound in transaction size.

5.2 Problem of Dynamic Program Partitioning

As revealed in our API design in § 4.2.2, a starting point of our approach is to expose to the runtime system some high-level semantic information. This information includes different types of data accesses; concretely, the data-container types in our API lets the runtime be aware of four data-access classes: 1) non-oblivious reads (or access-leaky reads as in `NobRO`), 2) non-oblivious writes (or access-leaky writes as in `NobRW`), 3) oblivious reads (as in `ObRO`) and 4) oblivious writes (as in `ObRW`).

With awareness of data access type, we can specify the constraints of program partitioning problem. In general, there are two constraints, the constraint imposed by the TSX hardware (C1) and that imposed by the security requirement (C2). Figure 5 illustrates the two constraints in the presence of four types of data accesses. For C1, it requires that a transaction's write-set should not cause L1 cache replacement, and a transaction's read-set should not cause LLC cache replacement. For C2, the access-leaky memory region must reside in data cache without being split. Here, an access-leaky memory region is defined by memory data accessed in a data-dependent way; for instance, `NobRO` and `NobRW` instances are access-leaky regions.[2]. The constraint of C2 is due to the strong security concern. Because the HTM based protection only conceals the intra-transaction memory access pattern and the inter-transaction memory access pattern is disclosed. Placing the leaky memory accesses across multiple transactions would inevitably disclose the data secrets.

Because of the constraints (particularly C2), a key design choice we made is that we focus on partitioning the non-leaky part of the

[2] We may use terms `NobXX` to represent access-leaky regions

program. That is, we consider the problem of partitioning the program into smaller units, each unit accessing a partition of non-leaky data (`ObXX`) but accessing leaky data in their entirety. Note that the leaky data is not partitioned and has to be loaded in total to the data cache. The implication of this design choice is that we only support computation with small cache-bound leaky data (`NobXX`).[3], which is consistent with our target computations (as described in § 3.1).

Our goal is two-fold: (O1) Support computations of (non-leaky) data as large as possible by dynamically partitioning the computations, and (O2) Run these computations using transactions of as large size as possible.

The high-level approach is illustrated in Figure 6. As in the figure, consider a computation whose application memory consists of (cache-bound) access-leaky data (i.e., A1 and A2) and large non-leaky data (i.e., A3 and A4). We split the non-leaky data to a series of smaller data partitions (i.e., {P3} and {P4}), each of which is small enough to fit in a CPU data cache. To support large transactions (O2), we enable L1 cache replacement on read-only data. That is, the footprint of A1 and P3 in L1 cache can be reduced to just one set, as will be explained next.

Formally, our work can be formulated as an optimization problem: Given a program G on data $\{A1, A2, A3, A4\}$, it partitions G to a series of $\{g\}$ with each program partition g operate on data $A1, A2, P3, P4$ such that it maximizes min $P3, P4$ and it is subject to the following constraints:

$$\begin{aligned} \cup\{P3\} &= A3 \\ \cup\{P4\} &= A4 \\ A1 + P3 + A2 + P4 &< LLC \\ A1/L + P3/L + A2 + P4 &< L1 \end{aligned}$$

Here, L is the maximal number of LLC cachelines that can be mapped to one L1 cache set.

5.3 Supporting Large Transactions by Shadow Memory

A1 can be replaced As will be described next, read-only data (A1 and P3) in a transaction can reuse L1 data cache without aborting the transaction.

cache full utilization is essential.[4]

Efficient use of the cache is of particular importance to ensuring large transactions. The transaction will not abort when the cache is underutilized; for instance, the transaction may end when only one cache set (out of 64) becomes full while the other sets are empty.

We propose a technique called shadow memory that arranges the in- transaction memory layout such that memory data is mapped to the CPU cache hierarchy without unnecessary conflict. Briefly, we enforce the rules that different types of data (A1, ...A4) should not overlap in data caches. Specifically, consider a cache hierarchy with a 8-way 32KB L1 cache and 16-way 8MB L2 cache. We allocate 60 L1C sets to be A2 (NobRW), 2 L1C sets to be P4, 1 L1C set to be P3 and L1C set to be A1. Note that A1 and P3 can reuse the L1 cache and their sizes are bounded by LLC. Figure 6 illustrates an example of the shadow memory.

5.4 Implementing Dynamic Partitioning

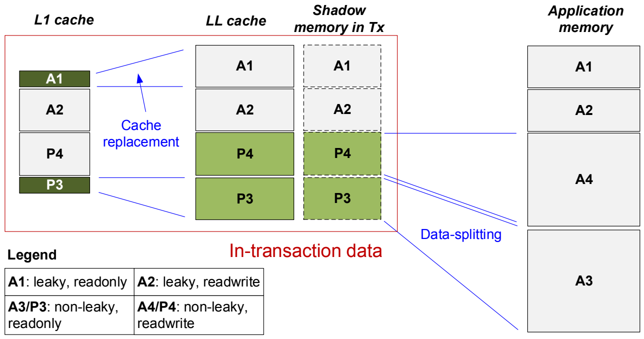


Figure 6: Splitting data to transaction-wise partitions and mapping data in memory/cache hierarchy: Shadow-memory is an indirection layer that arranges data in cache based on access pattern. From LL (Last-Level) cache to L1 (Level-one) cache, read-only data replacement is enabled.

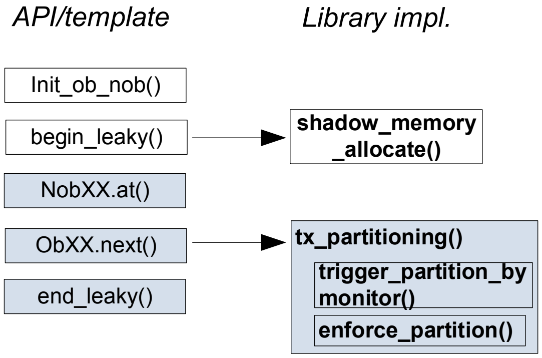


Figure 7: Library implementation: Each box represents a function in our library. White boxes run outside TSX transactions and blue boxes run inside TSX transactions.

To realize the dynamic partitioning on real hardware, there are two problems to be tackled: 1) How to realize the partitioning, in other words, how and when to insert the xbegin and xend instructions such that it can successfully execute the transaction (without abort) yet does not prematurely end a transaction? 2) How to realize the shadow memory in its full life cycle? 3) How to decide the condition that the computation cannot be executed using transactions at an early time (e.g., when the A1 is larger than LLC).

We build a library that realizes the partitioning life cycle. Recall that our API supports declaring the scope of leak section and wrapping each in-transaction memory access by custom data types. Upon leaky section declaration, we realize the allocation of shadow memory. Upon the in-transaction memory access, we hook the transaction partitioning schemes. The overall mechanism is illustrated in Figure 7.

Concretely, for shadow memory allocation, we allocate the shadow memory according to the layout described in § 5.3. Note that the shadow-memory size is fixed (as the total size of a cache is fixed). We perform checks on whether the data size, specifically the size of access-leaky memory data, is small enough to fit into the data cache. In other words, we enforce the constraint C1 that A1 does not exceed L1 and A2 does not exceed LLC.

For transaction partitioning, we implement two components: 1) monitoring cache usage, and 2) realizing transaction partitioning. For 1), we simply monitor the condition that if each non-leaky data

[3] Note that the assumption presents a fundamental limitation of our approach in supporting computation of large access-leaky data.

[4] For instance, a poorly utilized cache may abort a transaction when just one set (out of say 64 cache sets) is full. A well utilized cache may abort a transaction when the entire cache is full, which results in a much larger transaction.

overflows, that is, the iterator call to ObXX.next()[5] reaches its capability. Note that we enforce a joint capability limit, that is, the total size of ObXX objects is bounded instead of individual ObXX.

For 2), we insert the TSX instruction xend and xbegin. Between them, we make data copy between shadow memory and application memory of the non-leaky data. Because the computation is partitioned based on non-leaky data, different transactions in the same leaky-section computation will use different non-leaky data, thus making it necessary to reload non-leaky data across transactions. In addition, we perform double preloading across the transactional boundary.

```
begin_leaky(){
 shadow_memory_alloc();
}

ObXX::next(){
  //dynamically partition computations in next();
  //1. monitoring the overflow
  if(shadow_mem.ob.size + 1 < shadow_mem.ob.capability())
      return;
  //2. start to execute the partitioning
  xend();
  //reload non-leaky data (ob) in shadow memory from
     application memory
  shadow_mem.reload_ob();

  double_preload();
  xbegin();
}
```

Figure 8: Programming API: in-transaction data types and leaky section

6. CASE STUDY

Our system supports external-oblivious computing. In our perspective, there are some computations that are natively external-oblivious, such as binary search, iterative machine learning, etc.. There are some computations whose classic algorithms are not oblivious, but there exist external-oblivious algorithms. For instance, for sorting, the most common algorithms, such as quick sort, are not oblivious. There are external-oblivious sorting algorithms, such as oblivious merge sort and Melbourne shuffle based sort, that have the same $O(N \log N)$ complexity with quick sort.

In this section, we present our experience building various external-oblivious computations using our library.

6.1 K-Means Computation

A K-means computation takes as input a data array and randomly initialized K centroids. It produces the output of K centroids in K clusters that are the most representative of the data array. Here, each element in the data array represents a data point and there are pair-wise distances defined between them. The key-means computation runs iteratively, where each iteration consists of two data passes, one (KM1) is a nested loop that (re)-assigns every array element to the closest centroid, and the other (KM2) is an array scan that randomly accesses the centroid array to update the centroid position.

K-means is an external oblivious computation in the sense that it keeps the centroid as internal memory accessed in a leaky way, and the data array as an external memory, which is accessed obliviously.

To implement K-means using our library, we declare the data pass (KM2) to be the leaky section where the internal memory is accessed. In the leaky section, the centroid array is defined as a NobRW, data array as an ObRO, the data-centroid mapping as an ObRW. The related sample code is in Listing **??**.

6.2 Melbourne Shuffle

A Melbourne shuffle is a randomized algorithm for data shuffling. A data-shuffle operation takes as input two arrays of equal length, one storing data and the other storing a permutation. It produces the output of permuted data array. For instance, $\texttt{mshuffle}(x, y, z, 1, 0, 2) = y, x, z$. Internally, Melbourne shuffle is a randomized algorithm that bucketizes each of the two array to $\sqrt{N}$ buckets, each of $\sqrt{N}$ size. A Melbourne shuffle runs in two rounds, each round using $\log N\sqrt{N}$ internal memory accessed in a leaky fashion. The original input arrays are in external memory.

To implement Melbourne shuffle using our library, we declare both rounds to be leaky sections. In particular, for scalability, we declare the first round, which is to “distribute data bucket based on permutation bucket”, to be two leaky sections. Briefly, the first leaky section in the distribution round produces the unordered, linked list as output, without dummy elements. The second leaky section adds dummy elements. In this way, we can improve the data scalability significantly and reduce the access-leaky internal memory from $O(\log N\sqrt{N})$ to $O(\sqrt{N})$.

6.3 Oblivious Merge Sort

In oblivious merge sort, the operation of merging two sorted lists is made oblivious by a randomized algorithm [37]. Given two sorted lists, an oblivious merge keeps a $O(\sqrt{N})$ internal-memory buffer and finishes the merge computation with the same $O(N)$ complexity. It is a randomized algorithm and the probability of internal buffer overflowing is made negligible.

We implement oblivious merge sort by declaring the merge operation to be leaky section and the internal buffer to be a NobRW and the two merging arrays to be two ObRO.

6.4 Streaming Binary Search

Given a query and a sorted data array, a binary search locates the matching element in the array to the query. Given a stream of queries, the binary search can be treated as an external-oblivious process in the sense that the internal memory maintains the data array and external memory maintains the query stream.

To implement the streaming binary search using our library, we declare the binary search to be the leaky section, the data array to be a NobRO, the query stream (array) to be an ObRO.

7. PERFORMANCE EVALUATION

In this section, we evaluate our system in terms of data scalability and performance. We first conduct a micro-benchmark study on the transaction success rate. We then conduct a macro-benchmark study that extensively evaluates the system performance under different data scales.

7.1 Micro-Benchmark: Transaction Success Rate

7.2 Macro Benchmark: Execution Time

7.2.1 *Comparison with Scan Baseline*

This set of experiments evaluates the performance of CMO, with comparison to scan-based baselines. The scan-based baseline translates each leaky random access (on NobXX) to a full array scan. Note that while ORAM [35, 20] presents a generic solution with better (asymptotic) efficiency, the scan-based data-obliviousness approach is the state-of-the-art; it is implemented

[5] ObXX refers to non-leaky data type, either ObRO or ObRW.

more widely for in-memory data processing due to its simplicity, such as in oblivious machine learning [31] and ZeroTrace [32][6]. In addition, at a medium data scale, the better asymptotic efficiency of ORAM may not translate to better concrete performance. We implement the scan baseline in our library by converting each access to NobXX to a full-array scan.

We consider the various computations in our case study. In each experiment, we measure the performance by execution time. Specifically, the execution time only includes the time spent on executing the computation, and excludes the time spent on data loading and system initialization. We use numeric datasets and generate them randomly.

We did all the experiments on a laptop with an Intel 8-core i7-6820HK CPU of 2.70GHz, 32KB L1 and 8MB LL cache, 32 GB RAM and 1 TB Disk. This is one of the Skylake CPUs equipped with both SGX and TSX features.

We vary the data size in each computation and report the results in Figures 9 and 10.

Figure 10a and 10b evaluate the execution time of Melbourne shuffle and oblivious merge sort. Comparing the baseline of scan approach, the Melbourne shuffle by CMO achieves a speedup of more than 100X at the largest data scale of 1 million records. For oblivious merge sort, the speedup is much less significant, about 10

Figures 9b and 9a present the performance result of streaming binary search with varying data size and the number of queries. With varying data size, the CMO outperforms the scan baseline by a speedup ranging from 100X to 1000X. With varying query number, CMO first keeps the execution time constant, until the cache cannot accommodate all the queries. CMO's execution time then linearly scale with the query number. With a large query number, CMO outperforms the scan baseline by 1000X times. Similarly, the performance result of K-means in Figures 9c and 9d also show multi-magnitude speedup of CMO.

7.2.2 *Comparing Different Algorithms*

In this set of experiments, we evaluate the importance of dynamic partitioning by measuring the performance of CMO against two baselines without partitioning: 1) pure data-obviousness approach and 2) TSX based attack-detection (as in Cloak [22]). We allow one to choose the best and most-efficient algorithm in each baseline. For TSX attack detection, we consider quick sort. For data-oblivious computing, we consider bubble sort. For CMO, we consider three sorting algorithms, Melbourne shuffle with quick sort (following the scramble-then-compute paradigm [17]), oblivious merge sort, and cache shuffle with quick sort.

The performance result is in Figure **??**. It can be seen clearly the TSX attack detection is limited in data scalability, the baseline of bubble sort causes high performance overhead, especially as data size grows. The CMO paradigm, with different algorithms, achieves much better performance than the baselines with up to two orders of magnitudes speedup.

8. REFERENCES

[1] ARM TrustZone, https://www.arm.com/products/security-on-arm/trustzone.

[2] Google cloud platform supports intel sgx.

[3] Intel corp. software guard extensions programming reference.

[4] Introducing azure confidential computing.

[5] 2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015. IEEE Computer Society, 2015.

[6] M. Ajtai, J. Komlós, and E. Szemerédi. An o(n log n) sorting network. In D. S. Johnson, R. Fagin, M. L. Fredman, D. Harel, R. M. Karp, N. A. Lynch, C. H. Papadimitriou, R. L. Rivest, W. L. Ruzzo, and J. I. Seiferas, editors, Proceedings of the 15th Annual ACM Symposium on Theory of Computing, 25-27 April, 1983, Boston, Massachusetts, USA, pages 1–9. ACM, 1983.

[7] A. Arasu and R. Kaushik. Oblivious query processing. In Proc. 17th International Conference on Database Theory (ICDT), Athens, Greece, March 24-28, 2014., pages 26–37, 2014.

[8] Y. Azar and U. Vishkin. Tight comparison bounds on the complexity of parallel sorting. SIAM J. Comput., 16(3):458–464, 1987.

[9] J. Bater, G. Elliott, C. Eggen, S. Goel, A. N. Kho, and J. Duggan. SMCQL: secure query processing for private data networks. CoRR, abs/1606.06808, 2016.

[10] A. Bittau, Ú. Erlingsson, P. Maniatis, I. Mironov, A. Raghunathan, D. Lie, M. Rudominer, U. Kode, J. Tinnés, and B. Seefeld. Prochlo: Strong privacy for analytics in the crowd. In Proceedings of the 26th Symposium on Operating Systems Principles, Shanghai, China, October 28-31, 2017, pages 441–459. ACM, 2017.

[11] F. Brasser, U. Müller, A. Dmitrienko, K. Kostiainen, S. Capkun, and A. Sadeghi. Software grand exposure: SGX cache attacks are practical. In W. Enck and C. Mulliner, editors, 11th USENIX Workshop on Offensive Technologies, WOOT 2017, Vancouver, BC, Canada, August 14-15, 2017. USENIX Association, 2017.

[12] J. V. Bulck, F. Piessens, and R. Strackx. Sgx-step: A practical attack framework for precise enclave execution control. ACM SOSP Workshop (SysTex), 2017.

[13] 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 Kirda and Ristenpart [25], pages 1041–1056.

[14] J. Chen, Y. Tang, and H. Zhou. Strongly secure and efficient data shuffle on hardware enclaves. CoRR, abs/1711.04243, 2017.

[15] J. Chen, Y. R. Tang, and H. Zhou. Strongly secure and efficient data shuffle on hardware enclaves. ACM SOSP Workshop (SysTex), 2017.

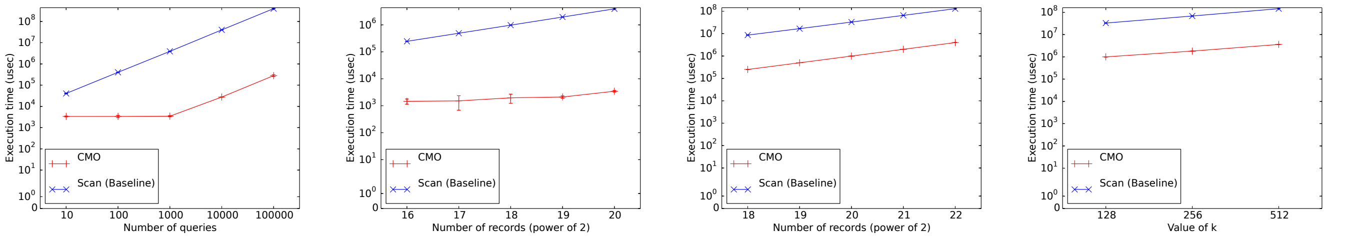
[16] S. Chen, X. Zhang, M. K. Reiter, and Y. Zhang. Detecting privileged side-channel attacks in shielded execution with déjà vu. In R. Karri, O. Sinanoglu, A. Sadeghi, and X. Yi, editors, Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security, AsiaCCS 2017, Abu Dhabi, United Arab Emirates, April 2-6, 2017, pages 7–18. ACM, 2017.

[17] H. Dang, T. T. A. Dinh, E. Chang, and B. C. Ooi. Privacy-preserving computation with trusted computing via scramble-then-compute. PoPETs, 2017(3):21, 2017.

[18] T. T. A. Dinh, P. Saxena, E. Chang, B. C. Ooi, and C. Zhang. M2R: enabling stronger privacy in mapreduce computation. In J. Jung and T. Holz, editors, 24th USENIX Security Symposium, USENIX Security 15, Washington, D.C., USA, August 12-14, 2015., pages 447–462. USENIX Association, 2015.

[19] S. Eskandarian and M. Zaharia. An Oblivious

[6] ZeroTrace uses ORAM for disk data access and scan for its internal-memory data access.



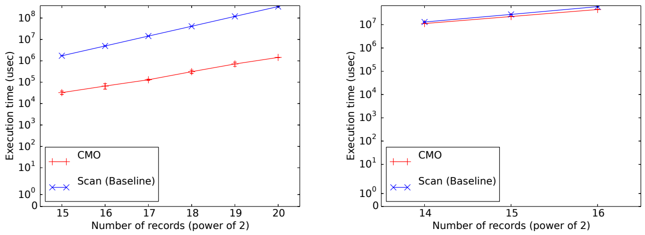
(a) Binary search: Number of queries

(b) Binary search: Number of records

(c) K Means: Number of records

(d) K Means: Number of centroids (K)

Figure 9: Comparing CMO with scan: Binary searches and k-means



(a) Melbourne shuffle

(b) Oblivious merge sort

Figure 10: Comparing CMO with scan: External-oblivious sorts

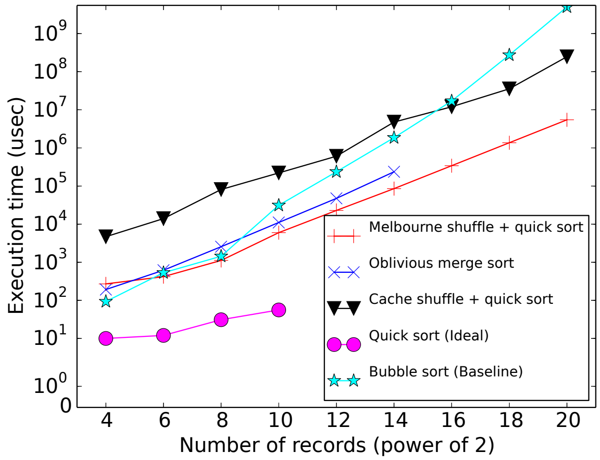


Figure 11: Comparing CMO with word-obliviousness and Cloak: the case of sorting

General-Purpose SQL Database for the Cloud. ArXiv e-prints, Oct. 2017.

[20] O. Goldreich and R. Ostrovsky. Software protection and simulation on oblivious rams. J. ACM, 43(3):431–473, 1996.

[21] M. T. Goodrich. Data-oblivious external-memory algorithms for the compaction, selection, and sorting of outsourced data. In R. Rajaraman and F. M. auf der Heide, editors, SPAA 2011: Proceedings of the 23rd Annual ACM Symposium on Parallelism in Algorithms and Architectures, San Jose, CA, USA, June 4-6, 2011 (Co-located with FCRC 2011), pages 379–388. ACM, 2011.

[22] D. Gruss, J. Lettner, F. Schuster, O. Ohrimenko, I. Haller, and M. Costa. Strong and efficient cache side-channel protection using hardware transactional memory. In Kirda and Ristenpart [25], pages 217–233.

[23] M. Hähnel, W. Cui, and M. Peinado. High-resolution side channels for untrusted operating systems. In 2017 USENIX Annual Technical Conference, USENIX ATC 2017, Santa Clara, CA, USA, July 12-14, 2017., pages 299–312. USENIX Association, 2017.

[24] J. Katz and Y. Lindell. Introduction to Modern Cryptography. Chapman and Hall/CRC Press, 2007.

[25] E. Kirda and T. Ristenpart, editors. 26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017. USENIX Association, 2017.

[26] S. Lee, M. Shih, P. Gera, T. Kim, H. Kim, and M. Peinado. Inferring fine-grained control flow inside SGX enclaves with branch shadowing. In Kirda and Ristenpart [25], pages 557–574.

[27] 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, SP 2015, San Jose, CA, USA, May 17-21, 2015 [5], pages 359–376.

[28] P. Maniatis, I. Mironov, and K. Talwar. Oblivious stash shuffle. CoRR, abs/1709.07553, 2017.

[29] O. Ohrimenko, M. Costa, C. Fournet, C. Gkantsidis, M. Kohlweiss, and D. Sharma. Observing and preventing leakage in mapreduce. In I. Ray, N. Li, and C. Kruegel, editors, Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-6, 2015, pages 1570–1581. ACM, 2015.

[30] O. Ohrimenko, M. T. Goodrich, R. Tamassia, and E. Upfal. The melbourne shuffle: Improving oblivious storage in the cloud. In Automata, Languages, and Programming - 41st International Colloquium, ICALP 2014, Copenhagen, Denmark, July 8-11, 2014, Proceedings, Part II, pages 556–567, 2014.

[31] O. Ohrimenko, F. Schuster, C. Fournet, A. Mehta,

S. Nowozin, K. Vaswani, and M. Costa. Oblivious multi-party machine learning on trusted processors. In T. Holz and S. Savage, editors, 25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016., pages 619–636. USENIX Association, 2016.

[32] S. Sasy, S. Gorbunov, and C. W. Fletcher. Zerotrace : Oblivious memory primitives from intel SGX. IACR Cryptology ePrint Archive, 2017:549, 2017.

[33] M.-W. Shih, S. Lee, T. Kim, and M. Peinado. T-sgx: Eradicating controlled-channel attacks against enclave programs. In NDSS Symposium 2017 in San Diego, California.

[34] S. Shinde, Z. L. Chua, V. Narayanan, and P. Saxena. Preventing page faults from telling your secrets. In X. Chen, X. Wang, and X. Huang, editors, Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security, AsiaCCS 2016, Xi'an, China, May 30 - June 3, 2016, pages 317–328. ACM, 2016.

[35] E. Stefanov, M. van Dijk, E. Shi, C. W. Fletcher, L. Ren, X. Yu, and S. Devadas. Path ORAM: an extremely simple oblivious RAM protocol. In 2013 ACM SIGSAC Conference on Computer and Communications Security, CCS'13, Berlin, Germany, November 4-8, 2013, pages 299–310, 2013.

[36] W. Wang, G. Chen, X. Pan, Y. Zhang, X. Wang, V. Bindschaedler, H. Tang, and C. A. Gunter. Leaky cauldron on the dark land: Understanding memory side-channel hazards in SGX. In B. M. Thuraisingham, D. Evans, T. Malkin, and D. Xu, editors, Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017, pages 2421–2434. ACM, 2017.

[37] P. Williams and R. Sion. Usable PIR. In Proceedings of the Network and Distributed System Security Symposium, NDSS 2008, San Diego, California, USA, 10th February - 13th February 2008. The Internet Society, 2008.

[38] Y. Xu, W. Cui, and M. Peinado. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In 2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015 [5], pages 640–656.

[39] 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 14th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2017, Boston, MA, USA, March 27-29, 2017, pages 283–298, 2017.