# Cache-Miss Oblivious Data Shuffling on Hardware Enclaves

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

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

Hao Zhou
Syracuse University
Syracuse, NY 13244
hzhou09@syr.edu

## ABSTRACT

Mitigating memory-access attacks on the Intel SGX architecture is an important and open research problem. A natural notion of the mitigation is cache-miss obliviousness which requires the cache-misses emitted during an enclave execution are oblivious to sensitive data. This work realizes the cache-miss obliviousness for the computation of data shuffling. The proposed approach is to software-engineer the oblivious algorithm of Melbourne shuffle [22] on the Intel SGX/TSX architecture, where the Transaction Synchronization eXtension (TSX) is (ab)used to detect the occurrence of cache misses. In the system building, we propose software techniques to prefetch memory data prior to the TSX transaction to defend the physical bus-tapping attacks. Our evaluation based on real implementation shows that our system achieves superior performance and lower transaction abort rate than the related work in the existing literature.

## 1. INTRODUCTION

Today we witness the emergence of hardware enclaves, a trusted execution environment that protects trusted user program against the untrusted operating system. A notable example is the recently released Intel Software Guard eXtension (SGX [4]) with increasing adoption for secure public-cloud computing (e.g. in Microsoft Azure [7] and Google Cloud Platform [2]). Various side-channel attacks on hardware enclave exploiting memory access pattern [27, 12, 19] have been proposed and demonstrated feasible in practice. Defending side-channel attacks on SGX-alike enclave architecture becomes an important and open research problem.

A natural notion of the defense is cache-miss obliviousness: The hardware enclave features a trusted processor issuing cache misses to access the memory in the untrusted world. Security can be assured by making the boundary crossing of cache miss oblivious to the sensitive data. This is especially effective to defending the physical attack, e.g. by bus tapping [13] and software attacks, e.g. page-fault controlled side-channel attack [27]. In addition to the strong security, cache-

miss obliviousness helps induce better performance as cache-miss oblivious algorithms have lower time complexity than the classic word-oblivious algorithms [28, 21] (see § 4.1 for performance discussion).

This work realizes the cache-miss obliviousness for data shuffling computation. The data shuffling is a basic operation used in many analytical computations. Specifically, we consider the Melbourne shuffle algorithm [22] which divides the data accesses in a shuffle to 1) the oblivious ones to a large external storage and 2) the non-oblivious ones to a small internal storage. We map the internal storage to the cache inside trusted process and the external storage to untrusted world (e.g. memory and disk). By this means, the cache misses that only touch the untrusted memory are made oblivious and thus safe to be disclosed.

When engineering the above mapping paradigm on SGX, one challenge is how to conceal the internal storage in cache with assured isolation. We leverage Intel Transaction Synchronization eXtension [6], a Hardware Transaction Memory feature in the latest Intel Skylake processor. TSX was originally designed for efficient concurrency. In this work, we propose a technique to enable the TSX transaction to detect cache misses; the observation is that TSX provides the capability of detecting early cache write-back (before the transaction commit) and cache miss can be detected if one can *equate cache miss with cache write-back.* The proposed technique pre-fetches all memory referenced in a transaction and make them dirty cache lines so that their eviction triggering write-back can be detected.

In addition to cache-miss detection, we propose techniques to *avoid* unnecessary transaction abort by carefully aligning data in memory and ensuring no conflict during data prefetch. Our technique leverages the specific semantic of oblivious computation and is distinct from the compiler-based partitioning schemes such as T-SGX [24].

We conduct algorithmic analysis to demonstrate the better complexity of our hybrid external-oblivious algorithms. More importantly, we conduct performance study based on real implementation, with the hope of verifying the advantage of systems-level performance of the cache-miss oblivious computation. The systems-level performance study is necessary even in the presence of algorithmic analysis. Because the overhead induced by cache-miss obliviousness, mostly due to TSX transaction execution, is higher than that of word obliviousness. Our performance study in realistic settings show that the systems-level overhead of cache-miss obliviousness is negligible in the presence of better time complexity, and cache-miss obliviousness causes an overall better performance than the

word-oblivious computation.

## 2. PRELIMINARY

### 2.1 Intel Software Guard eXtension (SGX)

Intel SGX is a security-oriented x86-64 ISA extension on the Intel Skylake CPU, released in 2016. SGX provides a "security-isolated world" for trustworthy program execution on an otherwise untrusted hardware platform. At the hardware level, the SGX secure world includes a tamper-proof SGX CPU which automatically encrypts memory pages (in the so-called enclave region) upon cache-line write-back. Instructions executed outside the SGX secure world that attempt to read/write enclave pages only get to see the ciphertext and can not succeed. SGX's trusted software includes only unprivileged program and excludes any OS kernel code, by explicitly prohibiting system services (e.g. system calls) inside an enclave.

To use the technology, a client initializes an enclave by uploading the in-enclave program and uses SGX's seal and attestation mechanism [11] to verify the correct setup of the execution environment (e.g. by a digest of enclave memory content). During the program execution, the enclave is entered and exited proactively (by SGX instructions, e.g. `EENTER` and `EEXIT`) or passively (by interrupts or traps). These world-switch events trigger the context saving/loading in both hardware and software levels. Comparing prior TEE solutions [5, 8, 1, 3], SGX uniquely supports multi-core concurrent execution, dynamic paging, and interrupted execution.

### 2.2 Intel Transactional Synchronization eXtension (TSX)

The purpose of TSX is to enable atomic execution of a code block or transaction from other processors' view point. This goal entails two requirements: 1) During the transaction execution, it is fully contained inside a processor and its memory-access requests can all be resolved inside the data cache. In other words, the transaction writes are buffered by dirty cache lines without being reflected in the memory. 2) By the end of transaction execution, the cached writes are successfully written back and the transaction can be committed only when there is no data conflict with other processor. A conflicting data access occurs when there is a location being accessed by two processors and at least one access is a transactional write.

To realize the two requirements, the TSX hardware supports the capability of aborting the execution of a transaction under various causes. It aborts a transaction when data conflict is detected at the transaction commit time (Abort Cause AC1). In order to detect data conflict, the hardware needs to track both the readset and writeset of a transaction. The writeset needs to be kept inside the L1 data cache (L1D) and readset needs to be inside the L3 cache. Thus, it aborts the transaction when the dirty data-cache lines are evicted, triggering cache write-back, before the end of transaction (AC2). It also aborts upon the readset exceeding the L3 cache (AC3). In addition, it aborts upon various systems events such as page-fault, interrupts and other exceptions delivered to the processor (AC4).

While TSX is originally designed for the performance and programmability in multiprocessing, its capabilities can be (ab)used for security purposes: It can be used to realize the cache-based or register-based computation [15] and to protect
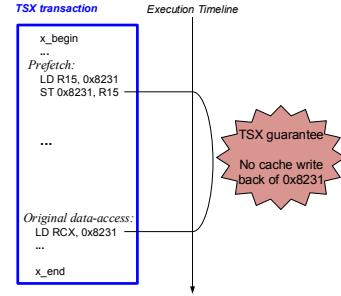


Figure 1: TSX (ab)used to detect cache misses: The hack here is that no cache write-back of a dirty line means the line stay present in the cache, implying a cache hit upon a memory request.

the private key from leaving a processor [16] by leveraging the TSX capability of detecting early cache write-back. T-SGX [24] defends the page-fault side-channel attacks by leveraging the TSX capability that page-fault events are intercepted by the TSX abort handlers before the untrusted OS. This work uses TSX for detecting cache-misses and for defending side-channel attacks. The use of TSX in this work is elaborated below.

### 2.3 Use of TSX for Detecting Cache-Miss

Our work requires to conceal the access-leaky internal storage in cache. Any access to the internal storage cannot be resolved by cache miss, which would otherwise leak the sensitive access. To conceal the internal storage in the last-level cache (LLC), it is equivalent to ensure no LLC cache miss during the time period when the internal storage is being accessed. Thus, the key requirement of our work is to ensure no LLC cache miss caused by internal-storage access.

TSX as is, however, does not provide this capability. For instance, a cache miss in a transaction does not necessarily cause the transaction to abort. A cache miss would abort a TSX transaction when it replaces a line that is read or written by the same transaction. The key insight of this work is to prefetch the entire read/write set of a transaction, such that the actual access must be served by cache hits (without leaky cache miss). More concretely, TSX guarantees any attempt to replace the prefetched cache line would abort the transaction. With unreplace-able prefetched lines, the actual access is guaranteed to be served by cache hits..

For example, consider the memory reference request of `LD RCX,0x8231` in the code sample in Figure 1. The memory reference can be resolved by a cache hit or a miss. To ensure no chance of cache miss, we prefetch the data to the LLC in the beginning of the transaction (i.e. "`LD R15,0x8231`"). After this instruction, the LLC cache-line buffering the content at `0x8231` is recorded into the readset of this transaction and is "pinned" there; TSX guarantees any attempt to replace the LLC line will abort the transaction, which is further captured by TSX abort handler. In other words, if the transaction does not abort when the execution reaches instruction "`LD RCX,0x8231`", the prefetched cache line is still present at least in the LLC and the memory reference must not cause LLC miss or cause any traffic on the system bus.

In general, the capability of prefetching transaction read-/write-set and pinning them to unreplaceable cache-lines can assist mitigate various memory-access attacks including phys-

ical bus tapping and cache-timing attacks. Bus tapping can be mitigated due to pinned cache-line guarantees cache hits. The cache-timing attacks are mitigated due to sharing cache-lines between transactions.

## 3. SYSTEM DESIGN AND IMPL.

### 3.1 Threat Model

We mainly consider a memory-access attacker who either directly sniffs the out-of-process memory-access traffic (e.g. by bus tapping or by page-fault channel [27]) or indirectly monitors the side-channel of cache timing [12].

The non-goals of this work includes the following attacks. 1) This work is complementary to rollback attacks load sealed but stale data across power cycles and restore the system to a stale state. 2) Given memory store both data and code, this work focuses on data-memory. The code-access attacks are orthogonal that exploit the access pattern to the memory region storing code, and that can be defended by existing techniques [20, 24] on a small memory. 3) We don't consider other side-channel attacks exploiting timing information or power usage [10]. 4) We don't consider denial-of-service attacks that the adversary declines to serve the requests from the enclave.

### 3.2 Security Definition

Intuitively, the memory-access obliviousness states that the memory access trace in a program execution is independent with any computation data (involving both input and intermediate data). Consider the execution of a program $P$ with data input $I$. The execution produces the memory-access trace $T$ that consists of all the last-level cache misses. The obliviousness requires that given two data values, $I_0$ and $I_1$, the an oblivious execution produces the same trace, that is, $T_P(I_0) = T_P(I_1)$. This definition assumes deterministic computation induced by $P$ and is about "perfect" obliviousness in the sense that it requires the traces under different input data stay exactly the same. Due to the systems nature of this work, we skip the more formal and generic definition of cache-miss obliviousness (e.g. based on indistinguishability formation [18]).

### 3.3 Step 1: Mapping Melbourne Shuffle to SGX

In this work, we focus on implementing cache-miss oblivious data shuffling. Data shuffle is a fundamental operator in oblivious data analysis computation. We describe the engineering of Melbourne shuffle [22] on Intel SGX. The idea is to map the program of Melbourne shuffle to TSX transactions and to isolate the leaky data access in cache by abusing TSX capability. Note this work only considers single-threaded execution.

**Preliminary of Melbourne shuffle [22]**: Melbourne shuffle is an oblivious, randomized algorithm for data shuffle. Given a data array and a permutation (of the same length), the computation of a data shuffle produces an array that reorders the data array based on the permutation. Internally, the Melbourne shuffle works in two data scans or passes, where the first pass, called distribution, scans the data array at the granularity of size $\sqrt{N}$ buckets and reorders individual buckets non-obliviously with the size-$p \log N \sqrt{N}$ internal memory. The second pass scans the array and sorts reorganized buckets internally. Figure 2a illustrates an example of Melbourne

shuffle. Overall, the trusted memory is $\log N \sqrt{N}$ with array length $N$. The details can be found in the original paper [22].

The Melbourne shuffle is mapped to TSX transactions such that the accesses to internal storage are kept inside transactions while external oblivious data accesses are kept outside transactions. In Melbourne shuffle, the mapping is illustrated in Figure 3 where the bucket-wise permutation multiplication in the "distribute" pass and the bucket-wise sort in the "cleanup" pass are mapped to individual transactions.

One implication of this mapping is that the internal storage of $\log N \sqrt{N}$ must be smaller than that of the size of L1 cache, which is one factor that constrains the scalability of CMOS on real SGX hardware (see § 4.1).

### 3.4 Step 2: Isolating Cached Data by TSX

Isolating cached data is realized by data prefetching which simply prefetch all data referenced inside a transaction. Given a prefetched line, the TSX capability guarantees that at least the line will not be replaced from LLC during the transaction. Our goal in this work is to avoid self-eviction, that is, the dataset prefetched does not conflict each other. Here, we consider both conflict and capacity cache misses during prefetching. Given a set-associative cache, we lay out memory properly such that the number of conflicts in each cache set do not exceed the capacity (i.e. the number of ways). This applies for both L1 and LL caches.

Figure 2b shows how isolation is realized with the distribution phase of Melbourne shuffle. First, data is prefetched from the enclave to the cache. Second, it runs non-oblivious computation on the cached data; this phase is wrapped in TSX transactions to ensure no cache miss. Third, the end of the transaction triggers the write-back of cached lines to the enclave memory. The second-phase transaction, in particular, takes two continuous memory regions as input and output data stored in another contiguous memory region. To avoid conflict in this layout, we partition the three regions at granularity of cache lines and precompute (at compilation time) that the number of conflicts in each cache set does not exceeds the number of ways.

### 3.5 Implementation Notes

**Abort Handling**: Transaction aborts are handled by re-executing the transactions. Before entering the transaction, the context (all the values in registers) is saved to a memory area pointed by a reserved register `R15`. Upon aborts, the handler reloads the context prior to the transaction from `R15`, before jumping back to the beginning of the transaction to re-execute it. An important property is that the value of `R15` must be preserved through the regular transactional path and abort path.

**CPUID**: Obtaining cache information (e.g. cache sizes) is realized by calling the functions provided in Intel SGX SDK [4] which switches out to the untrusted world and calls `CPUID` instructions. Here, the result of `CPUID` is not necessarily be trusted. The enclave can run test program to evaluate the cache sizes itself. Concretely, it can read a series of arrays with increasing lengths and the maximal length without aborting transaction is the size of LLC cache. Similarly, it can writes to a series of length-increasing arrays to obtain the maximal length as L1 cache size.

**Randomness generation**: Melbourne shuffle is a randomized algorithm and we generate true randomness using SDK-provided function `sgx_read_rand`.

(a) Melbourne shuffle: The example shows the shuffle of data array (a,b,c,...,i) based on permutation (3,1,6,5,7,2,0,8,4). The bucket used in the distribute pass is vertical and the bucket in the sort pass is horizontal

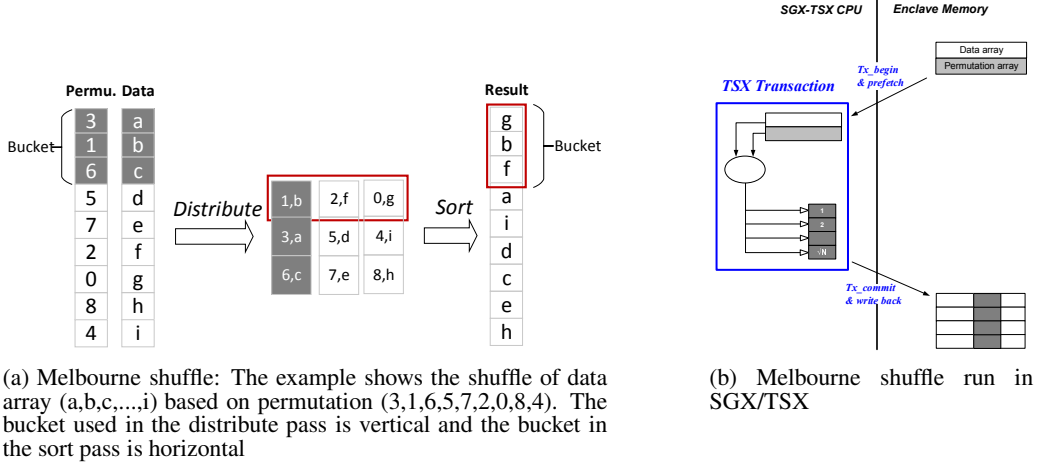(b) Melbourne shuffle run in SGX/TSX

Figure 2: Engineering Melbourne shuffle algorithms on SGX/TSX

```
1  int[] Melbourne_shuffle(int[] data, int[] perm){
2    perm_r = gen_perm();
3    data_r = shuffle_pass(data, perm_r);
4    perm_rr = shuffle_pass(perm, perm_r);
5    return shuffle_pass(data_r, perm_rr);
6  }
7  int[] shuffle_pass(int[] data, int[] perm){
8    int[][] inter = distribute(data, perm);
9    return cleanup(inter);
10 }
11 int[][] distribute(int[] data, int[] perm){
12   for(int i=0; i < sqrt(length(data));i++){
13     inter[i] = tx_bucket_permmul(data,perm,i);
14   }
15   return inter;
16 }
17 int[] cleanup(int[][] inter){
18   List res;
19   for(i < sqrt(length(data))){
20     res.add(tx_bucket_sort(inter,i));
21   }
22   return res.toarray();
23 }
```

Figure 3: Melbourne shuffle mapped to TSX transactions

## 4. EVALUATION

This section evaluates the performance and abort rate of CMOS system. Specifically, it aims at answering the following two questions.

- What is the performance of CMOS comparing the baseline of word-oblivious shuffle and pure transaction-based protection?

- What is the abort rate of CMOS comparing with the implementation without prefetching?

### 4.1 Performance

We consider two baselines for performance comparison against CMOS. The first baseline (BL1) is pack the naive, non-oblivious shuffle algorithm in TSX transactions whose time complexity at $O(N)$ is better than the $O(\sqrt{N} \log N)$ complexity of Melbourne shuffle. The second baseline is the word-oblivious shuffle that is realized by running a bubble sort on the permutation array. This baseline has worse time complexity than CMOS, but it does not need run transactions, adding performance uncertainty.
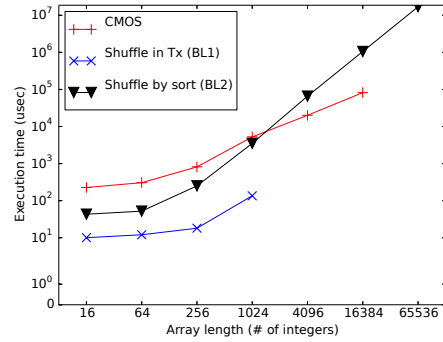


Figure 4: Execution time of CMOS and other shuffle baselines

**Experiment setup**: 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 use numeric datasets and generate them randomly.

In the evaluation, we measure the execution time and the maximal data size supported. In transaction-based approaches, the individual transaction is bounded by L1 cache size, which in our platform is 32 KB.

The performance result is illustrated in Figure 4. It can be seen that BL1 is the most efficient but with limited scalability. BL2 has the best data scalability but may not be efficient, especially when data size is large. CMOS starts to show superior performance when data size roughly grows beyond 4096. CMOS can scale to larger dataset than BL2 because of its smaller space complexity. It has smaller execution time than BL1 because of the better time complexity.

### 4.2 Transaction Abort Rate

One of the design goals of prefetch in CMOS is to reduce the rate of self-abort. To evaluate the design effectiveness, we run the program several runs and measure the average abort rate. We consider the first baseline that simply skips the external prefetching. The baseline would cause transactions to abort due to both interrupt (AC4) and cache write-back (AC2) (recall § 2.2). We also consider two additional baselines that do not cause write-back by placing nop instructions in a transaction. This way, the baselines are only caused by inter-
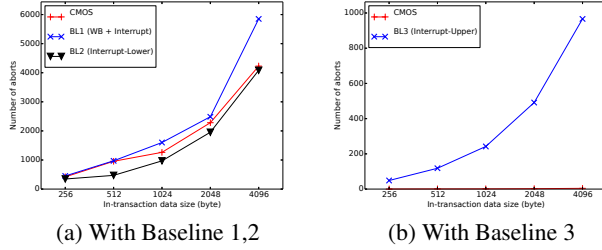
(a) With Baseline 1,2      (b) With Baseline 3

Figure 5: Number of transaction aborts

rupt. We implement two variants: the second baseline, named interrupt-lower, runs enough `nop` instructions such that the running time is equal to the case that all memory references are solved by cache hits. The third baseline, named interrupt-upper, runs enough `nop` instructions such that the running time is equal to the case that all memory references are solved by cache misses. Note that to determine the appropriate number of `nop` instructions, we profile our SGX hardware using some benchmark programs (e.g. calling cache-flush instructions to measure the cache-miss time).

We conduct experiments by running the four approaches above multiple times. We vary the size of data referenced inside the transaction and report the percentage of transaction aborts during the execution, in Figure 5. As can be seen, CMOS causes aborts that are fewer than BL1, close to BL2, and are much fewer than BL3. Given BL2 is ideal lower bound of aborts, the result shows that the CMOS design of double prefetch is highly effective in reducing cache write-back aborts. In addition, CMOS is approximately the same with the Ideal-lower bound, which shows the CMOS design *eliminates* the cache write-back aborts as the ideal can not be aborted by cache write-back.

## 5. RELATED WORK

In this section, we survey the related work on defending memory-access side-channel attacks on hardware enclaves.

Making memory access oblivious is a feasible defense strategy to side-channel attacks in general. Comparing the cumbersome Oblivious RAM protocols [14, 26], oblivious algorithms [21] are more lightweight and are promising towards practical attack defense. Various data-analytical systems are developed based on the computation-specific oblivious algorithms, such as Opaque [28] for relational data analytics and oblivious machine learning [23]. These systems instantiate the notion of word-obliviousness; recall that its goal is to make memory references at word granularity oblivious. ObliVM [21] is a source-to-source program transformation that translates the annotated program into efficient oblivious algorithms. This line of research does not consider external-oblivious algorithms and does not attain the cache-miss obliviousness on SGX.

T-SGX [24] takes a general-purpose approach to defend the page-fault side-channel attack [27]. T-SGX's approach is to assume the allocated memory is large enough to hold the data referenced by the application, such that during execution there is no page-fault. It leverages the TSX capability of detecting page-faults in user-space programs. It partitions the program and wraps the partitions into individual TSX transactions. Similarly, work [25] takes a compiler approach to de-

fend the page-fault based side-channel attacks. It is based on the notion of page-fault obliviousness.

This work is different from T-SGX in the following senses: 1) The goals are different: T-SGX is a defense of page-fault side-channel attacks, and this work is to defend all software/hardware memory-access pattern attacks on SGX. T-SGX makes page-fault in enclave unobservable, while this work is to make enclave execution cache-miss oblivious. 2) The approaches are different: T-SGX defends by detecting page faults inside TSX transactions, while this work defends by detecting cache misses inside TSX transactions. While both seem to rely on TSX, their *use* of TSX transactions is fundamentally different, which will be elaborated on in the next paragraph. 3) The applicability is different: T-SGX is a general-purpose, compiler-based solution, and this work is specific to data analytics and leverage the corresponding "semantics" for better efficiency. In addition, T-SGX mainly focus on the security of code-page execution, while this work focuses on data-obliviousness.

The use of TSX transactions in T-SGX and this work is different. In T-SGX, it assumes a memory large enough to store all data pages and ensure no page-fault during enclave execution. Given the limited "size" allowed by a TSX transaction, T-SGX packages enclave computation in as many TSX transactions as needed and ensure the security of page-access across transactions by placing all the code outside transaction on a single page (so-called Springboard page). It is important to note the Springboard code-page does not access memory on any other pages. In this work, the TSX transactions are used to isolate the enclave computation inside the processor and to detect any (unexpected) cache misses. Across transactions, the data security is ensured by running external oblivious algorithms. This work is applicable to a more realistic setting when handling a large volume of data, that is, we allow page-fault during program execution and the working-set memory can be much smaller than original dataset.

## 6. CONCLUSION

This work defends enclave side-channel attacks by cache-miss obliviousness. The proposed approach is software engineering the target oblivious computation on top of the SGX and TSX platform. It proposes cache-miss oblivious algorithms with small trusted space. It has several software-engineering strategies that package the computation into TSX transactions, achieving cache-miss obliviousness. Through initial evaluation, the performance overhead of cache-miss obliviousness is much smaller than that of the Strawman approach.

## Acknowledge

## 7. REFERENCES

[1] ARM TrustZone, http://www.arm.com/products/processors/technologies/trustzone/.
[2] Google cloud platform supports intel sgx.
[3] IBM SCPU, http://www-03.ibm.com/security/cryptocards/.
[4] Intel corp. software guard extensions programming reference.
[5] Intel TXT, http://www.intel.com/technology/security/downloads/trustedexec_overview.pdf.
[6] Intel's transactional synchronization extensions (intel's tsx) overview.
[7] Introducing azure confidential computing.

[8] TPM, http://www.trustedcomputinggroup.org/tpm-main-specification/.

[9] 2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015. IEEE Computer Society, 2015.

[10] J. B. Almeida, M. Barbosa, G. Barthe, F. Dupressoir, and M. Emmi. Verifying constant-time implementations. In Holz and Savage [17], pages 53–70.

[11] I. Anati, S. Gueron, S. P. Johnson, and V. R. Scarlata. Innovative technology for cpu based attestation and sealing.

[12] 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), Vancouver, BC, 2017. USENIX Association.

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

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

[15] L. Guan, J. Lin, B. Luo, and J. Jing. Copker: Computing with private keys without RAM. In 21st Annual Network and Distributed System Security Symposium, NDSS 2014, San Diego, California, USA, February 23-26, 2014. The Internet Society, 2014.

[16] L. Guan, J. Lin, B. Luo, J. Jing, and J. Wang. Protecting private keys against memory disclosure attacks using hardware transactional memory. In 2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015 [9], pages 3–19.

[17] T. Holz and S. Savage, editors. 25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016. USENIX Association, 2016.

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

[19] S. Lee, M. Shih, P. Gera, T. Kim, H. Kim, and M. Peinado. Inferring fine-grained control flow inside SGX enclaves with branch shadowing. CoRR, abs/1611.06952, 2016.

[20] C. Liu, A. Harris, M. Maas, M. W. Hicks, M. Tiwari, and E. Shi. Ghostrider: A hardware-software system for memory trace oblivious computation. In Ö. Özturk, K. Ebcioglu, and S. Dwarkadas, editors, Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '15, Istanbul, Turkey, March 14-18, 2015, pages 87–101. ACM, 2015.

[21] 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 [9], pages 359–376.

[22] 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.

[23] O. Ohrimenko, F. Schuster, C. Fournet, A. Mehta, S. Nowozin, K. Vaswani, and M. Costa. Oblivious multi-party machine learning on trusted processors. In Holz and Savage [17], pages 619–636.

[24] M.-W. Shih, S. Lee, T. Kim, and M. Peinado. T-sgx: Eradicating controlled-channel attacks against enclave programs. 2017.

[25] 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.

[26] 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.

[27] 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 [9], pages 640–656.

[28] 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.