Leveraging Hardware Transactional Memory for Cache Side-Channel Defenses

Sanchuan Chen
The Ohio State University
chen.4825@osu.edu

Fangfei Liu
Intel Corporation
fangfei.liu@intel.com

Zeyu Mi
Shanghai Jiao Tong University
yzmizeyu@gmail.com

Yinqian Zhang
The Ohio State University
yinqian@cse.ohio-state.edu

Ruby B. Lee
Princeton University
rblee@princeton.edu

Haibo Chen
Shanghai Jiao Tong University
haibochen@sjtu.edu.cn

XiaoFeng Wang
Indiana University at Bloomington
xw7@indiana.edu

ABSTRACT

A program's use of CPU caches may reveal its memory access pattern and thus leak sensitive information when the program performs secret-dependent memory accesses. In recent studies, it has been demonstrated that cache side-channel attacks that extract secrets by observing the victim program's cache uses can be conducted under a variety of scenarios, among which the most concerning are cross-VM attacks and those against SGX enclaves. In this paper, we propose a mechanism that leverages hardware transactional memory (HTM) to enable software programs to defend themselves against various cache side-channel attacks. We observe that when the HTM is implemented by retrofitting cache coherence protocols, as is the case of Intel's Transactional Synchronization Extensions, the cache interference that is necessary in cache side-channel attacks will inevitably terminate hardware transactions. We provide a systematic analysis of the security requirements that a software-only solution must meet to defeat cache attacks, propose a software design that leverages HTM to satisfy these requirements and devise several optimization techniques in our implementation to reduce performance impact caused by transaction aborts. The empirical evaluation suggests that the performance overhead caused by the HTM-based solution is low.

1 INTRODUCTION

Cache side-channel attacks are one type of security threats that break the confidentiality of a computer system or application which have several variants, *e.g.*, Prime-Probe [9, 12, 16, 17, 26], Flush-Reload [23, 24, 27], Evict-Time [21], and Cache-Collision [3] attacks. These attacks differ in ways they are conducted, their underlying assumptions about the attack scenarios, and exploitable levels of CPU caches (*e.g.*, L1, LLC), which makes the construction of effective defenses challenging.

ASIA CCS '18, June 4–8, 2018, Incheon, Republic of Korea

ACM ISBN 978-1-4503-5576-6/18/06…$15.00
https://doi.org/10.1145/3196494.3196501

Existing defenses against these attacks are generally classified into one of the three categories: hardware defenses, system-level defenses and software-level defenses. Hardware defenses propose new hardware designs to eliminate cache side-channel attacks from the root cause, *e.g.*, by randomizing cache accesses, partitioning cache regions, *etc.* Some of these approaches are particularly effective against certain types of attacks while preserving performance efficiency [22]. However, it usually takes a very long time for a novel cache design to be adopted in commercial products. In contrast, system-level defenses leverage the privileges of an operating system or hypervisor to enforce isolation between unprivileged software components by static cache partition [19] or dynamic cache partition [10, 11, 29], or to add random noise into side-channel observations [28]. System-level defenses can be effective solutions to many types of side-channel threats, but also come with the drawbacks of being very specific to the attacks they aim to defeat, usually at the cost of inefficient use of hardware resources, and more importantly not applicable in cases where the system software is not fully trusted. A third approach, software-level defenses, transforms the victim software itself to eliminate secret-dependent control flow and data flow [14], or to diversify the victim software to enforce probabilistic defenses [5]. Software-level defenses face the challenges of generalizing the protection to arbitrary software programs and at the same time to maintain low performance overhead.

In this paper, we present a mechanism to enable software applications to defend themselves against a wide range of cache side-channel attacks. Our solution leverages an existing hardware-based performance enhancement feature, hardware transactional memory (HTM), available on modern commercial processors to deterministically eliminate cache side-channel threats. The hardware transactional memory is usually implemented through cache, such as Intel's Transactional Synchronization Extensions (TSX) [4]. The hidden assumption of this mechanism is that whenever the cache line contained in the read set or write set is evicted out of the cache, it loses track of the cache line, therefore the transaction will abort, and all the modifications are rolled back.

In this work, we particularly explore Intel TSX as a case study, and show how such hardware features, when facilitated by a set of software mechanisms, can be leveraged to defeat known cache side-channel attacks that target various cache levels. To do so, we systematically analyzed four types of side-channel attacks, including Prime-Probe, Flush-Reload, Evict-Time, Cache-Collision,

and enumerated the security requirements that the software solution must meet to defeat all these types of cache side-channel attacks. Guided by the security requirements, we propose a system design that uses Intel TSX to protect security-critical regions from cache attacks and elaborated the implementation efforts to satisfy the requirements. We applied the solution to the vulnerable implementation of AES and ECDSA algorithm in `OpenSSL` and vulnerable modular exponentiation algorithm (used in RSA and ElGamal algorithms) in the `mbedTLS-SGX` library. Experimental evaluation suggests that the induced performance overhead is small.

In summary, we make the following contributions in this paper:

- An analysis of security requirements for software solutions to defeat all four types of side-channel attacks, including Prime-Probe, Flush-Reload, Evict-Time, Cache-Collision attacks.
- A software-level solution leveraging hardware transactional memory features in commodity processor to enable a program to protect itself from a wide range of cache side-channel attacks.
- Case studies of applying the solution to the AES, RSA and ECDSA implementations of popular open-source cryptographic libraries, such as `OpenSSL` and `mbedTLS-SGX`, demonstrating the efficiency of the protected algorithms.

2 BACKGROUND

Cache Side-Channel Attacks. Cache side-channel attacks exploit the timing difference between cache hits and cache misses to infer the victim's memory access patterns, hence indirectly deduce the secret information if it is dependent on such access patterns. Secret information can be modulated into the memory accesses through secret-dependent control flow or secret-dependent data flow.

Cache side-channel attacks are conventionally categorized into access-driven attacks and timing-driven attacks. In access-driven attacks, attacker can observe which cache lines have been accessed by victim by measuring his own memory access time, e.g., Prime-Probe and Flush-Reload attacks. In timing-driven attacks, attacker can measure execution time of some fragment of victim program, e.g., Evict-Time and Cache-Collision.

To systemetically model the threats we consider, we formally define a cache side-channel attack as 4-tuple: (C, M, L, G), which refers to type of caches exploited, attack method used, type of information leakage, and the granularity of information learned by attacker, respectively. We particularly consider two types of G: *synchronous* attacks (`SYNC`) and *asynchronous* attacks (`ASYNC`). In synchronous attacks, attacker is assumed to explicitly interact with victim by invoking victim's execution through regular service interface. In contrast, in asynchronous attacks, we do not assume any explicit interaction and assume an "asynchronous" attacker to perform multiple cache measurements during victim's execution. We summarize prior studies in cache side-channel attacks in Table 1.

Transactional Memory. Transaction memory is a parallel programming model for coordinating concurrent reads and writes of shared data, without the use of locks for synchronization. One of the simplest and most widely implemented Hardware Transaction memory (HTM) is the cache-based HTM. CPU cache serves as a natural place to track a transaction's read set and write set because memory accesses involve cache lookups. Furthermore, existing cache coherence protocol can be easily extended for conflict detection. In cache-based HTMs, a transaction must be terminated whenever an overflow occurs due to set associativity conflicts.

Table 1: Existing cache side channel attacks in literature.

Cache Side-Channel Attacks	Literature
(L1C, Prime-Probe, *, ASYNC)	[16–18, 26]
(L1C, Prime-Probe, data, SYNC)	[17, 21]
(LLC, Prime-Probe, inst, ASYNC)	[12]
(LLC, Prime-Probe, data, SYNC)	[9]
(L1C, Flush-Reload, data, ASYNC)	[8]
(LLC, Flush-Reload, inst, ASYNC)	[23, 24]
(LLC, Flush-Reload, *, SYNC)	[27]
(L1C, Evict-Time, data, SYNC)	[2, 17, 21]
(LLC, Evict-Time, data, SYNC)	[1]
(L1C, Cache-Collision, data, SYNC)	[3]

3 SECURITY REQUIREMENTS AND SYSTEM DESIGN

3.1 Security Requirements

To defeat various cache side channels we listed in Table 1, a software solution must satisfy the following design goals:

- **S1:** Cache lines loaded in the security-critical regions cannot be evicted or invalidated during the execution of the security-critical regions. If so it happens, the code must be able to detect such occurrences.
- **S2:** The execution time of the security-critical region is independent of the cache hits and misses.
- **S3:** The cache footprints after the execution of the security-critical region are independent of its sensitive code or data.
- **P1:** Performance overhead for the protected program is low without attacks.

We argue by satisfying the security goals (S1 - S3), we can prevent all types of cache side-channel attacks we consider. The security analysis is listed as follows.

- Asynchronous attacks: Asynchronous attacks can only be performed using Prime-Probe or Flush-Reload techniques. In either case, the attacker needs to evict (or invalidate) the victim's loaded cache lines out of the cache *during the execution* of the victim process. **S1** guarantees that such activities will not be conducted successfully without being detected.
- Synchronous timing-driven attacks: In these synchronous timing-driven attacks, the attacker does not perform measurements during the execution of the victim, hence achieving **S1** only cannot defeat the attacks. However, **S2** guarantees that the attacker cannot extract any information from the execution time.
- Synchronous access-driven attacks: In these attacks, the attacker tries to infer information from the cache footprints of the security-critical regions after the execution of the victim process. **S3** guarantees that the attacker cannot obtain information by observing the cache footprints.

3.2 System Design

Each of the security-critical regions is first enclosed into a single HTM-based transaction, as shown in Listing 1.

Satisfying S1 using TSX: Our key insight is that the root cause of many cache side channels is that victim's cache lines can be evicted by the adversary during the execution of the sensitive operations. How exactly transaction aborts can be associated with cache

Listing 1: Pseudo code for the prudent design.

```
while(1) {
   if (_xbegin() == _XBEGIN_STARTED)
      break;
   retry++;
   if (retry == THRESHOLD)
       goto fail;
}
  preload();
  //
  // security-critical region;
  //
  _xend();
fail:
  failure handler;
```

line evictions is clouded by the interaction with multiple layers of processor caches and undisclosed implementation details of TSX. We empirically study the implementation of Intel TSX by conducting a series of experiments, which can be found in Appendix A. Through these experiments, we have concluded with the following observations:

- **O1:** Eviction of cache lines written in a transaction out of L1 data cache will terminate the transaction, while eviction of cache lines read in the transaction will not.
- **O2:** Eviction of data read and instructions executed in a transaction out of LLC will abort the transaction.
- **O3:** Transactions will abort upon context switches.

We note that as observed in **O1**, when the cache lines that are read during the transaction are evicted out of the L1 cache by another thread sharing the same core, transaction will not abort. Therefore, Intel TSX cannot be used to protect against cache attacks when Intel HyperThreading is enabled.

Satisfying S2 and S3 by cache preloading: One promising approach to achieve **S2** and **S3** is to normalize cache usage inside the transactions, by either preloading all sensitive memory regions at the beginning of the transaction or touching all these regions at the very end of the transaction. Both approaches guarantees that after a successfully committed transaction, the cache footprint of the security-critical region is independent of the secret. However, cache preloading has two additional benefits: First, by preloading all memory regions that will be used in the transaction at the beginning, execution of the security-critical region will only have cache hits, e.g., in Evict-Time, preloading guarantees all the security-critical region is present in cache before execution and in Cache-Collision, there is no cold miss when executing security-critical region. Therefore, the total execution time of this region will be independent of the secrets; thus requirement **S2** is met. Moreover, cache preloading also guarantees that a prematurely terminated transaction will also have the same cache footprint regardless of the secrets. Therefore, requirement **S3** is met. As such, by preloading relevant memory regions into cache at the beginning of the transaction, both security requirements **S2** and **S3** are satisfied.

Minimizing transactions by breaking down security-critical regions: Large security-critical regions may cause self-conflicts in cache when executed inside a memory transaction, thus terminating the transaction during its execution. Moreover, longer execution time of a security-critical region will have higher probability to be interrupted. As such, it is important to select security-critical regions that are *small* enough to fit into a transaction without self-conflicts, and a *short* enough, in terms of execution time, that will finish without software or hardware interruption with high probability. To minimize the generated transactions, the developer must break down one entire security-critical region into several pieces. If it is impossible to do so, the code must be refactored.

Re-entering transactions to avoid false positives: Since a transaction may abort due to various reasons, even when it is not under active side-channel attacks, false detection of **S1** violation is possible. Following terms used in intrusion detection, we use false positives to refer to such false detection of **S1** violation when the program is not under cache side-channel attacks, and use false negatives to refer to cases where cache side-channel attacks is taking place but the program fails to detect such occurrences. In order to reduce false positives, instead of jumping directly to a failure handling path immediately after the transaction aborts, the transactional execution is re-entered a few times before failing indefinitely.

4 IMPLEMENTATION

In this section, we detail our implementation of the solution. We particularly discuss a set of techniques to reduce transaction aborts.

System calls. System calls will lead to privilege-level switch, which will result in transaction aborts. Therefore, the security-critical region that is encapsulated in the hardware transactions should not include any system calls. Any I/O operations are similarly disallowed. Our implementation specially avoids including system calls and I/O operations in transactions. Another complication is that dynamic memory allocation through `malloc()` also requires to issue system calls to request kernel service. To avoid transaction aborts due to memory allocation, we pre-allocate a large chunk of memory from the heap before the transaction starts, and implements a new user-level memory allocation interface to allocate and free memory buffers from this managed memory chunk.

Page faults. Page faults, if not handled properly, are one of the most common reasons to abort a transaction. All virtual memory pages that need to be accessed within the transaction has to be touched before the transaction starts, so that the page table entries corresponding to the work set of the transaction are properly set up. Particularly, to warm up the code pages, our implementation first obtains the start address and end address of the code segment, and then reads one word in each page of the code segment. To warm up the writable static data and heap accessed by the transaction, we write to each page before the transaction starts. To warm up the stack, our implementation calls a dummy function that writes to a local data array that is large enough. In this way, when returned from the dummy function, we are certain that the stack has enough mapped memory to be used during the execution of the transaction.

Code refactoring. When security-critical code region is too large to be placed into one transaction, we refactor the code region to fit into the transaction. One such technique is to avoid calling non-security-critical functions in security-critical code region. If additional function calls are inevitable, we move the function call site outside transaction. We will illustrate such techniques in case studies. It is worth emphasizing that after the refactoring, the security-critical regions become small. However, as long as the

region is larger than a cache line, the program's access pattern in these cache lines can leak information of the control flow inside the security-critical region which needs refactoring.

Abort reasoning. With the assistance of Linux `perf` tool, one can debug the transaction code to identify the reasons of transaction aborts. However, as `perf` only reports the abort code that corresponds to one of the six general categories, it is difficult to precisely identify the true reason of transaction aborts. To ensure low transaction abort rate, the developer can leverage hardware performance counters to sample TSX related events to report the instructions that cause the aborts and try to reduce the abort rate accordingly.

4.1 Examples

To demonstrate the use of hardware transactional memory for side-channel defenses, we applied our design to protect several known vulnerable cryptographic implementations.

AES in OpenSSL. The C implementation of AES in OpenSSL has been shown to be vulnerable to various side-channel attacks [3], because the AES table lookup indices are directly related to the round keys. In particular, the AES tables are security-critical data structures that needs to be protected. We enclose the whole block encryption/decryption function in one transaction. Only two minor changes were required: 1) Before entering the transaction, we need to "touch" the data pages to avoid page faults. The data pages include the AES tables, round keys, buffers for the plaintext and ciphertext. 2) At the beginning of the transaction, five 1-KB AES tables are preloaded into the cache. The AES block encryption/decryption is short enough with very low probability to be interrupted.

ECDSA in OpenSSL. The Elliptic Curve Digital Signature Algorithm (ECDSA) algorithm in OpenSSL v1.0.1e has been shown vulnerable to cache side-channel attacks [23]. Particularly as shown in Listing 2 (in Appendix B due to the space limitation), in the inner loop of `ec_GF2m_montgomery_point_multiply()` of the Montgomery ladder algorithm is security-critical, which is enclosed in one transaction. However, because both `Madd()` and `Mdouble()` (name shorten for convenience) calls a large number of functions internally, including them in transaction will result in transaction abort almost every time. Therefore, we refactored the code to keep security-critical region small. New code is shown in Listing 3 (in Appendix B). In addition to code refactoring, two minor changes are made to adapt Montgomery ladder algorithm to TSX: 1) Before entering the transaction, we touch first and last word of `ec_GF2m_montgomery_point_multiply()` to avoid page faults. 2) At the beginning of the transaction, we preload this function into cache for security purposes.

Modular exponentiation in mbedTLS-SGX. Intel SGX have been shown susceptible to cache side-channel attacks [20]. We show that vulnerable code run in SGX can also be protected using transactional memory. In mbedtls-SGX, modular exponentiation is implemented using the square-and-multiply algorithm which has been shown vulnerable to side-channel attacks. We refactored the piece of code as is shown in Listing 4 (in Appendix B). Particularly, it performs multiplication operation regardless of the bit value, and use transactional memory to protect the test of the bit value to prevent side-channel leakage.

5 EVALUATION

We evaluated performance overhead for cryptographic libraries due to our defense mechanisms using both micro benchmarks and macro benchmarks. All empirical evaluations were conducted on an Intel Core i5 6440HQ processor (single socket, 4 CPU cores, and 1 thread per core) with 32KB L1 instruction/data caches, 256KB L2 caches, and 3MB LLC. The maximum allowed consecutive transaction aborts (*i.e.*, the threshold) is 10000 in all experiments.

5.1 Micro Benchmarks

Micro benchmarks of the cryptographic code test performance of a particular cryptographic operation. Therefore, these results reflects the cost of protection on the algorithms themselves. We particularly tested the overhead of OpenSSL's AES and ECDSA implementation and the mbedTLS's RSA implementation. In all experiments we report below, no execution failure due to excessive transaction aborts (more than the threshold, *i.e.*, 10000) was observed.

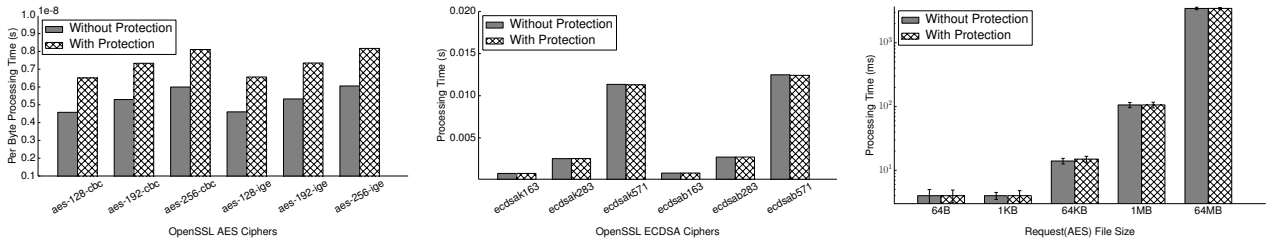
AES in OpenSSL. To test the overhead on AES encryption, we used OpenSSL's built-in `speed` command line tools to test OpenSSL v1.0.2f, with and without protection and evaluate throughput of the AES encryption. We tested 6 combinations of key size and mode of operations: `aes-128-cbc`, `aes-192-cbc`, `aes-256-cbc`, `aes-128-ige`, `aes-192-ige`, and `aes-256-ige`. We converted the output of the tool to reflect processing time for each byte, and illustrate the results in Fig. 1a. We can see that the performance overhead of the protection ranges from 34.1% to 42.7%. We also measured the abort rate of the hardware transactions using Linux's `perf`, a performance profiling tool using hardware performance counters. In the experiments shown in Fig. 1a, when the protected code is executed, the transaction abort rate on average was 0.0189%.

ECDSA in OpenSSL. We tested the ECDSA implementation in OpenSSL 1.0.1e using OpenSSL `speed` command line tool with the following six algorithms: `ecdsak163`, `ecdsak283`, `ecdsak571`, `ecdsab163`, `ecdsab283`, and `ecdsak571`. Measured average processing time of ECDSA signing with and without protection is shown in Fig. 1b. Performance overhead for these selected parameter settings is between -0.497% and 0.883%. The performance gain with the protected version in some cases is presumably due to code refactoring. The abort rate measured by `perf` was 0.001%.

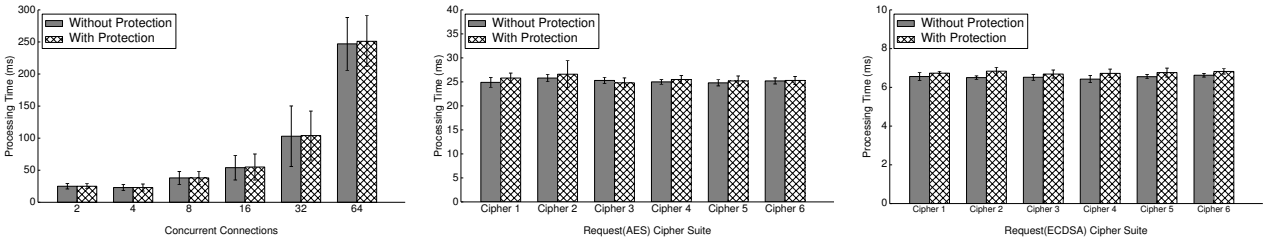
RSA in mbedtls-SGX. We tested the performance overhead of the RSA implementation in mbedtls-SGX due to our protection. The overhead is measured by the execution time of an RSA decryption (an average of 10 runs). The experiment result suggest that after applying the protection, the performance overhead is 1.107% without any transaction aborts (as measured by `perf`).

5.2 Macro Benchmarks

To test the performance impact of the defense to real-world applications, we set up an Apache HTTPS web server (version 2.4.25) which is dynamically linked to an OpenSSL library (*i.e.*, `libcrypto.so`) with the AES and ECDSA implementations protected using the hardware transactional memory. In the following tests, the HTTPS clients were run on a different machine that is connected with the server through a local area network. In all experiments we report below, no execution failure due to excessive transaction aborts (more than the threshold, *i.e.*, 10000) was observed.



(a) Per byte processing time of OpenSSL AES decryption. **(b) Processing time of OpenSSL ECDSA signing.** **(c) HTTPS latency with varying sizes of the requested files.**



(d) HTTPS latency with varying concurrent connections. **(e) HTTPS latency with varying ciphersuites.** **(f) SSL handshake latency with varying ciphersuites.**

Figure 1: Performance evaluation. Cipher 1 - 6 in are ECDHE-ECDSA-AES256-GCM-SHA384, ECDHE-ECDSA-AES256-SHA384, ECDHE-ECDSA-AES256-SHA, ECDHE-ECDSA-AES128-GCM-SHA256, ECDHE-ECDSA-AES128-SHA256, and ECDHE-ECDSA-AES128-SHA.

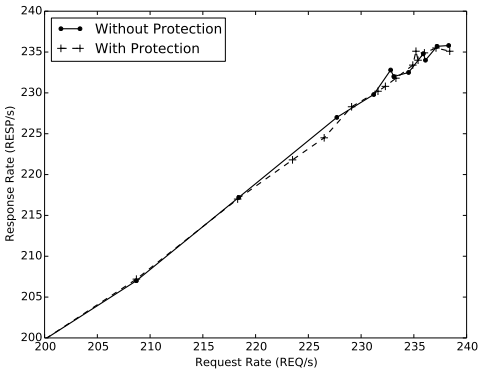


Figure 2: Throughput of Apache HTTPS server

Overhead of Apache web servers due to protected AES encryption/decryption. We conducted four sets of experiments to test the performance overhead to Apache web servers when the AES encryption was protected by the transactions and client runs an ApacheBench [6] to conduct the performance testing:

- We tested the latency of of the Apache web server when the fetched file size varies, with 64B, 1KB, 64KB, 1MB, and 64MB files and each file was requested 100 times. The ciphersuite used in these tests were `ECDHE-RSA-AES256-SHA` and the concurrent connection number was 1. The results are shown in Fig. 1c. We can see when the protection is applied to AES encryption/decryption, the latency of the HTTPS request only increases slightly (less than 7.1%, and 0 to 1% in most cases). Note the y-axis in the figure is in log scale.
- We generated HTTPS requests with 2, 4, 8, 16, 32, and 64 concurrent connections. Fetched file size was 1KB and ciphersuite used were `ECDHE-RSA-AES256-SHA`. The results are shown in Fig. 1d. We can see that performance overhead is between 0% and 1.85%. Therefore, we can conclude with concurrent HTTPS requests, the transaction-based protection still work well.
- We tested the response latency under 6 ciphersuites with 1 concurrent connection and the 1KB fetched file size. We show the results in Fig. 1e. From the figure we can see that the performance overhead is between -1.9% and 3.6%.
- We used `httperf` [15] to generate requests for 1KB files using HTTPS with the `ECDHE-RSA-AES256-SHA` ciphersuite. The `httperf` tool makes a specified and fixed number of connections with the HTTPS server (with 1 request per connection) and measures response rate. We altered the request rate until the Apache web server's response rate could not keep up with requests. The result is shown in Fig. 2. It can be seen that two curves for the server throughput with and without protection are almost the same. The saturate point without protection was 235.1 request per second, and server's saturation point with our protection was 234.9. Throughput drop was less than 0.1%.

SSL handshake performance with ECDSA signatures. We linked the Apache web server to an OpenSSL 1.0.1e library. We used the OpenSSL's `s_time` command line to test the performance of the handshake. The evaluation results are shown in Fig. 1f, which indicate the overhead is between 2.5% and 5.1%.

6 DISCUSSION

Our solution cannot be used to defeat side-channel attacks that are initiated from another thread sharing the same core and the protected code must avoid running on processors with HyperThreading enabled. Our current design requires the source code of the protected programs. However, by leveraging binary reassembly

techniques, it is possible to disassemble the binary code, apply our mechanism to protect critical regions, and recompile the source code into binary. We have discussed in Sec. 3 that the program must be allowed to re-enter the transaction if it only aborts a number of times that is lower than the threshold. A reasonably high threshold is preferred in practice, which will reduce false positives; but it should be kept low enough to prevent infinite loops.

7 RELATED WORK

Closest to our work are solutions that build defense mechanisms into the protected programs themselves. One such method is software transformation. Molnar *et al.* [14] proposed a program counter security model to eliminate secret-dependent control flows. Zhang *et al.* proposed methods to equip guest VMs with capabilities to detect existence of third-party VMs in public clouds [25] and to obfuscate cache contents to defeat L1 cache side-channel attacks [28]. Our solution can also help tenants of public clouds to protect their programs running in their own VMs. But our method also works in non-virtualized or SGX settings.

Although Intel TSX has been proposed as a mechanism for implementing concurrent programs, it has been leveraged to enhance system security in several previous studies, e.g., Liu *et al.* [13] proposed to use of Intel TSX in virtual machine introspection.

Concurrent to our work is a paper published recently by Gruss et al. [7], which also studied the use of TSX for cache side-channel defenses. Although these two papers achieves similar security properties by leveraging Intel TSX, they differ in the following aspects: (1) In contrast to Gruss et al., our paper discusses TSX-based defenses against not only Prime-Probe and Flush-Reload cache side-channel attacks but also Evict-Time and Cache-Collision attacks; (2) the security analysis in our paper arrives a similar conclusion from different angles; (3) our performance evaluation was conducted using Apache HTTPS web server linked to the OpenSSL library, validating the feasibility of this approach in practical uses, while Gruss et al. only applied the solution to cryptographic algorithms to demonstrate the defenses.

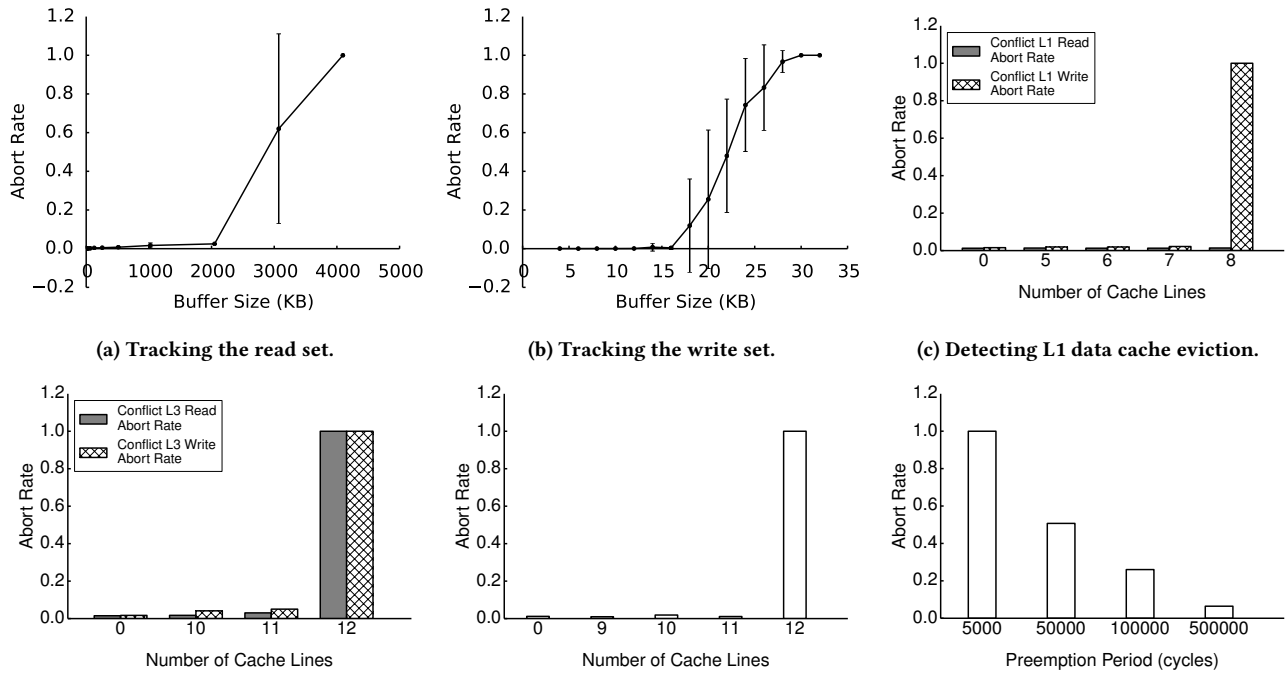
8 CONCLUSIONS

In conclusion, this paper presents a defense against cache side-channel attacks using hardware transactional memory that is already available in modern processors. The paper provides a systematic analysis of the security requirements that a software-only solution must meet to defeat cache attacks, and then propose a software design that leverages Intel TSX to satisfy these requirements. The defense mechanisms have been implemented on several cryptographic algorithms, including AES, ECDSA and RSA, and the evaluation suggests that the performance overhead due to the protection is very small to the Apache HTTPS servers.

Acknowledgments. This research was supported in part by NSF grants 1566444, 1750809, 1526493, 1618493, CNS-1527141 and ARO W911NF1610127.

REFERENCES

[1] Gorka Irazoqui Apecechea, Mehmet Sinan Inci, Thomas Eisenbarth, and Berk Sunar. 2014. Fine grain Cross-VM attacks on Xen and VMware are possible!. In *Cryptology ePrint Archive*.
[2] Daniel J. Bernstein. 2005. *Cache-timing attacks on AES.* Technical Report.
[3] Joseph Bonneau and Ilya Mironov. 2006. Cache-Collision timing attacks against AES. In *Proceedings of Cryptographic Hardware and Embedded Systems (CHES'06)*.
[4] Intel Corporation. 2014. Intel 64 and IA-32 Architectures Software Developer's Manual, Combined Volumes: 1, 2A, 2B, 2C, 3A, 3B and 3C. (2014).
[5] Stephen Crane, Andrei Homescu, Stefan Brunthaler, Per Larsen, and Michael Franz. 2015. Thwarting cache side-channel attacks through dynamic software diversity. In *ISOC Network and Distributed System Security Symposium*.
[6] The Apache Software Foundation. 2017. ApacheBench: Apache HTTP server benchmarking tool. (2017).
[7] Daniel Gruss, Julian Lettner, Felix Schuster, Olya Ohrimenko, Istvan Haller, and Manuel Costa. 2017. Strong and Efficient Cache Side-Channel Protection using Hardware Transactional Memory. In *26th USENIX Security Symposium*.
[8] David Gullasch, Endre Bangerter, and Stephan Krenn. 2011. Cache games – bringing access-based cache attacks on AES to practice. In *32nd IEEE Symposium on Security and Privacy*.
[9] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. 2015. S$A: A shared cache attack that works across cores and defies VM sandboxing—and its application to AES. In *36th IEEE Symposium on Security and Privacy*.
[10] Taesoo Kim, Marcus Peinado, and Gloria Mainar-Ruiz. 2012. STEALTHMEM: System-level protection against cache-based side channel attacks in the cloud. In *21st USENIX Security Symposium*.
[11] Fangfei Liu, Qian Ge, Yuval Yarom, Frank Mckeen, Carlos Rozas, Gernot Heiser, and Ruby B. Lee. 2016. CATalyst: Defeating last-level cache side channel attacks in cloud computing. In *22nd IEEE Symposium on High Performance Computer Architecture*.
[12] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. 2015. Last-level cache side-channel attacks are practical. In *36th IEEE Symposium on Security and Privacy*.
[13] Yutao Liu, Yubin Xia, Haibing Guan, Binyu Zang, and Haibo Chen. 2014. Concurrent and consistent virtual machine introspection with hardware transactional memory. In *20th International Symposium on High Performance Computer Architecture*.
[14] David Molnar, Matt Piotrowski, David Schultz, and David Wagner. 2005. The program counter security model: Automatic detection and removal of control-flow side channel attacks. In *8th International Conference on Information Security and Cryptology*.
[15] David Mosberger and Tai Jin. 1998. Httperf – A tool for measuring web server performance. *ACM SIGMETRICS Performance Evaluation Review* (1998).
[16] Michael Neve and Jean-Pierre Seifert. 2007. Advances on access-driven cache attacks on AES. In *13th International Conference on Selected Areas in Cryptography*.
[17] Dag Arne Osvik, Adi Shamir, and Eran Tromer. 2006. Cache attacks and countermeasures: the case of AES. In *6th Cryptographers' Track at the RSA Conference on Topics in Cryptology*.
[18] Colin Percival. 2005. Cache missing for fun and profit. In *2005 BSDCan*.
[19] Himanshu Raj, Ripal Nathuji, Abhishek Singh, and Paul England. 2009. Resource management for isolation enhanced cloud services. In *ACM Cloud Computing Security Workshop*.
[20] Michael Schwarz, Samuel Weiser, Daniel Gruss, ClÃĬmentine Maurice, and Stefan Mangard. 2017. Malware Guard Extension: Using SGX to conceal cache attacks. arXiv:1702.08719. (2017). https://arxiv.org/abs/1702.08719.
[21] Eran Tromer, Dag Arne Osvik, and Adi Shamir. 2010. Efficient cache attacks on AES, and countermeasures. *Journal of Cryptology* 23, 2 (Jan. 2010).
[22] Zhenghong Wang and Ruby B. Lee. 2008. A novel cache architecture with enhanced performance and security. In *41st Annual IEEE/ACM International Symposium on Microarchitecture*.
[23] Yuval Yarom and Naomi Benger. 2014. Recovering OpenSSL ECDSA nonces using the FLUSH+RELOAD cache side-channel attack. In *Cryptology ePrint Archive*.
[24] Yuval Yarom and Katrina Falkner. 2014. FLUSH+RELOAD: A high resolution, low noise, L3 cache side-channel attack. In *23rd USENIX Security Symposium*.
[25] Yinqian Zhang, Ari Juels, Alina Oprea, and Michael K. Reiter. 2011. HomeAlone: Co-residency detection in the cloud via side-channel analysis. In *32nd IEEE Symposium on Security and Privacy*.
[26] Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. 2012. Cross-VM side channels and their use to extract private keys. In *19th ACM Conference on Computer and Communications Security*.
[27] Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. 2014. Cross-tenant side-channel attacks in PaaS clouds. In *ACM Conference on Computer & Communications Security*.
[28] Yinqian Zhang and Michael K. Reiter. 2013. Düppel: Retrofitting commodity operating systems to mitigate cache side channels in the cloud. In *20th ACM Conference on Computer and Communications Security*.
[29] Ziqiao Zhou, Michael K. Reiter, and Yinqian Zhang. 2016. A software approach to defeating side channels in last-level caches. In *23rd ACM Conference on Computer and Communications Security*.



(a) Tracking the read set. **(b) Tracking the write set.** **(c) Detecting L1 data cache eviction.**

(d) Detecting data eviction in LLC. **(e) Detecting eviction of instruction in LLC.** **(f) Detecting context switch.**

Figure 3: Empirical evaluation of Intel TSX.

Appendices

A DETECTING CACHE EVICTIONS USING INTEL TSX

Our empirical study on the implementation of Intel TSX is conducted on a dual-core (4-threads with Intel HyperThreading) 2.3GHz Intel Haswell Core i5 5300U processor, which is equipped with separate 8-way 32KB L1 data and instruction caches and unified 256KB L2 caches private to each core, and a 12-way 3MB LLC shared by both cores. Our empirically assessment of TSX implementation aims to answer the following questions:

How are read set and write set tracked? We first study the size of the read set and write set in TSX, respectively. To do that, we sequentially read (or write) a buffer (of the size that is a parameter of the experiments) within a transaction using the RTM programming interface. This is repeated for 100,000 times and we count the number of cases that the transaction aborts. We expect that abort rate will increase drastically when the size of the buffer becomes larger than the read (or write) set. Fig. 3a and Fig. 3b shows the abort rate for reading and writing buffer of different sizes. We find that the size of the write set is slightly less than the size of the L1 data cache (32 KB), while the size of the read set is similar to the size of the LLC (3 MB). This means that the write set of Intel TSX is tracked in the L1 data cache and the read set is tracked in the LLC.

Can TSX detect L1 data cache evictions? In this experiment, we run two programs concurrently. The first program, P_1, runs in a loop that is encapsulated in a transaction and accesses a specific memory location, M, in each loop. The loop iterates 20 times before it terminates. The other program, P_2, tries to evict that cache line (where M is stored) of P_1, by repeatedly reading a tunable number of memory blocks, which map to the same cache set as M in the L1 cache. We measured the transaction abort rate encountered by P_1 while varying the number of memory blocks P_2 accessed.

We pin P_1 and P_2 on different hardware threads of the same core (with hyperthreading enabled), hence they share the same L1 data cache. Fig. 3c shows the results when P_1 access M by reading or writing, respectively. Since the associativity of the L1 data cache is 8, when the number of memory blocks accessed by P_2 is increased to 8, P_2 can evict the whole cache set by its own data. We find that the transaction will not abort if the data read by a transaction is evicted out of the L1 data cache, but it will cause the transaction to abort with very high probability if the data written by a transaction is evicted out of the L1 data cache.

Can TSX detect data eviction in the LLC? We conducted an experiment that is similar to the one above. The difference is that we pin P_1 and P_2 on different cores so that they only share the LLC and the memory blocks accessed by P_2 maps to the same cache set as M in the LLC. The results are shown in Fig. 3d. Since the associativity of the LLC is 12, P_2 can evict the whole cache set when it accesses 12 or more memory blocks. We find that the eviction of data accessed by a transaction out of the LLC, no matter read or write, always results in transaction aborts.

Can TSX detect eviction of instructions? Ideally, HTM only needs to handle memory accesses made by the load/store instructions within a transaction. In practice, many HTM designs adopt a paradigm called *implicit transactions*, which means that all memory accesses within the transaction boundaries are transactional, including instruction fetches. Therefore, we hypothesize that with implicit transaction, the read set also includes locations executed by the transaction, which means that we can also detect the evictions of instructions (from the LLC), similar to the data. To validate this, we conduct the following experiment: we run two programs P_1 and P_2 concurrently on different cores. P_1 repeatedly executes a transaction that calls a dummy function located at at address M for 20 times, with a small delay in between two invocations. The dummy function does nothing but returns to the caller immediately. P_2 repeatedly reads a tunable number of memory blocks, which map to the same LLC cache set as M, to evict the cache line of M. Fig. 3e shows that when instructions are evicted out of the LLC, the transaction will also abort.

Will transactions abort upon context switches? In this experiment, we developed two programs running on the same core, with HyperThreading turned off. Program P_1 run in the transaction and repeatedly reads from a memory location M. Program P_2 spawns 10 threads, each of which runs for a very short period of time to access 8 memory blocks that map to the same L1 data cache set as M and then sleeps. Each thread programs a software timer to wake up the next thread after certain time interval, t, which represents the period that P_1 is preempted [8]. In each of the experiments, we varied the preemption period, t. The result is shown in Fig. 3f. Although, as shown in the previous experiments, the eviction of data read by a transaction out of the L1 data cache does not abort the transaction, high-frequency preemption would yield high abort rate when P_2 is on the same core as P_1. It suggests that a transaction will abort upon context switches.

B SECURITY-CRITICAL REGION CODE REFACTORING EXAMPLES

Due to space constraints, we put Listing 2, Listing 3, Listing 4 here. In Listing 2, because both `Madd()` and `Mdouble()` calls a large number of functions internally, we refactored the code in Listing 3 to keep security-critical region small. In Listing 4, we refactored square-and-multiply algorithm so that it performs multiplication operation regardless of the bit value.

Listing 2: Pseudo code for Montgomery ladder.

```
for(; i >= 0; i--) {
  word = scalar->d[i];
  while(mask) {
    // security-critical region
    if(word & mask) {
      Madd(group, &point->X, x1, z1, x2, z2, ctx);
      Mdouble(group, x2, z2, ctx);
    }
    else {
      Madd(group, &point->X, x2, z2, x1, z1, ctx);
      Mdouble(group, x1, z1, ctx);
    }
    mask >>= 1;
  }
  mask = BN_TBIT;
}
```

Listing 3: Pseudo code for Montgomery ladder after refactoring.

```
for(; i >= 0; i--) {
  word = scalar->d[i];
  while(mask) {
    while(retry < THRESHOLD) {
      if(_xbegin() == _XBEGIN_STARTED) {
        // security-critical region
        if(word & mask) {a=x1;b=z1;c=x2;d=z2;}
        else {a=x2;b=z2;c=x1;d=z1;}
        _xend();
        break;
      }
      else retry++;
    }
    if(retry == THRESHOLD) failure_handler();s
    Madd(group, &point->X, a, b, c, d, ctx);
    Mdouble(group, c, d, ctx);
    mask >>= 1;
  }
  mask = BN_TBIT;
}
```

Listing 4: Pseudo code for Square-and-Multiply after refactoring.

```
TX = DUMMY;
while(retry < THRESHOLD) {
  if(_xbegin() == _XBEGIN_STARTED) {
    // security-critical region
    if( bit == 1 ) TX = X;
    _xend();
    break;
  }
  else retry++;
}
if(retry == THRESHOLD) failure_handler();
Mult(TX);
```