A Novel Covert Channel Attack Using Memory Encryption Engine Cache

Youngkwang Han and John Kim
KAIST, Daejeon, Korea
{sft_glory,jjk12}@kaist.ac.kr

Abstract

Microarchitectural covert channel attack is a threat when multiple tenants share hardware resources such as last-level cache. In this work, we propose a novel covert channel attack that exploits new microarchitecture that have been introduced to support memory encryption – in particular, the memory encryption engine (MEE) cache. The MEE cache is a shared resource but only utilized when accessing the integrity tree data and provides opportunity for a stealthy covert channel attack. However, there are challenges since MEE cache organization is not publicly known and the access behavior differs from a conventional cache. We demonstrate how the MEE cache can be exploited to establish a covert channel communication.

1 INTRODUCTION

As multiple tenants share hardware resources in the cloud, providing secure system environment has become a critical concern. In particular, shared hardware resources present opportunity for different type of attacks, including side-channel attacks and covert-channel attacks. Prior works have shown that cross-core microarchitectural attacks exploiting shared hardware can leak information to other tenants, including cache-based cross-core attacks [4, 7, 9, 17]. In particular, last-level cache attacks have been shown to be practical with low error rate. On the other hand, diverse hardware and software based defense mechanisms for the cache attack [1, 8, 15, 16] have also been proposed. Other microarchitectural attacks which utilize different shared hardware, such as DRAM row buffer, memory order buffer, and memory bus, have also been proposed [11, 13, 14]. All these microarchitectural attacks are susceptible to noise since the microarchitectural components are commonly used across all users. However, in this work, we propose a new type of covert channel that exploits dedicated hardware microarchitecture designed to support memory encryption.

To provide secure environment for user-level application, hardware support has been recently introduced. For example, Intel SGX (Software Guard eXtensions) allows user-level code to be allocated on a private region in memory, referred to as *enclaves*, that are protected from malicious privileged software. In this work, we exploit memory encryption hardware provided to support Intel SGX – in particular, the Memory Encryption Engine (MEE) cache to propose a new type of cross-core, microarchitectural covert channel attack.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
DAC '19, June 2–6, 2019, Las Vegas, NV, USA
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-6725-7/19/06…$15.00
https://doi.org/10.1145/3316781.3317750

MEE cache is similar to other hardware cache (e.g., last-level cache) as it is a shared resource across multiple cores and caches recently accessed data. MEE cache access characteristics are very different from other hardware cache and present new challenges to establish a covert channel. However, the differences also prevents previously proposed cache-based covert channel defenses from being directly applicable to our proposed MEE covert channel attack.

In this work, we discuss the challenges in implementing covert channel over the MEE cache. Since the hardware organization of the MEE cache is not publicly known, we first reverse engineer the MEE cache size and set-associativity. We then exploit the timing difference that exists between accessing the main memory from MEE cache hit to implement the covert channel attack. We demonstrate how previously proposed Prime+Probe [7] method can not be applied to MEE cache covert channel and describe how we implement covert channel attack over the MEE Cache. We evaluate our covert channel attack on native system and show that our covert channel attack achieves 35KBps bit rate with 1.7% error rate without any error handling. While other covert channel attacks [7, 9, 11] have demonstrated higher bit rate, this work is one of the first to exploit the MEE cache for covert channel attack.

2 BACKGROUND AND RELATED WORK

2.1 Microarchitectural covert channel

Covert channel is an unauthorized communication channel that bypasses computer security policy. It can transfer information by regulating some conditions of a medium, e.g., network, microarchitecture hardware, etc. Covert channel has two parties, trojan and spy. Trojan is implanted on victim's environment and leaks sensitive information such as encryption keys to the spy, and the spy resides on attacker's environment to retrieve the information. In particular, microarchitectural covert channel utilizes shared hardware microarchitecture resources and often exploits timing difference that comes from sharing the hardware.

Cache-based covert channel exploits shared cache in CPU to leak information by exploiting the fact that cache access latency is smaller than main memory access latency. Percival [10] utilized L1 cache to create a covert channel between two processes running on simultaneous multi-threading system, while Ristenpart et al. [12] demonstrated cache-based covert channel can work between different virtual machines in cloud environment. Shared last-level cache (LLC) enables covert channel communication across different cores as the LLC is often inclusive of L1 and L2 and guarantees that eviction of victim data from LLC leads to data eviction from private L1 and L2 cache. Different LLC covert channel attacks have been proposed, including Flush+Flush method [4], Prime+Probe method in virtualized environment [7], and error-free LLC covert channel with Prime+Probe method [9]. Memory-based covert channels includes using the memory bus for cross-VM covert channel [14] and using DRAM row buffers for cross-CPU covert channel [11].

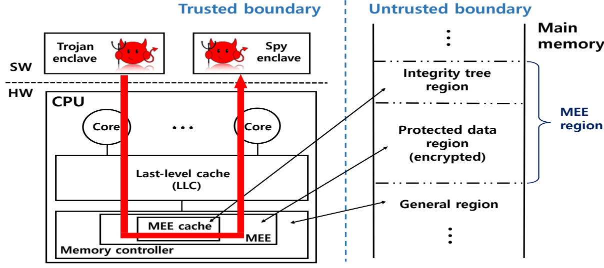

Figure 1: High-level block diagram of MEE (Memory Encryption Engine) and our covert channel attack.

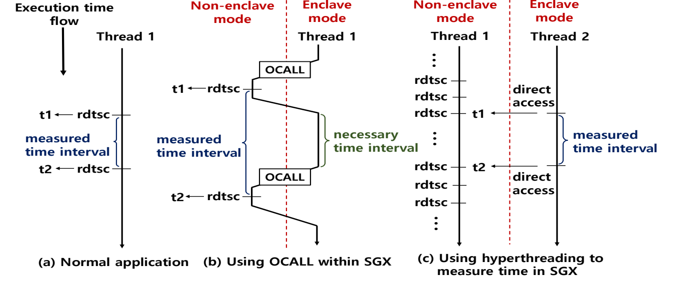

Figure 2: Different approaches to measuring time or latency on x86 system.

2.2 MEE cache

SGX [2] is an extension to Intel architecture that provides integrity and confidentiality to secure computation running in remote computer where privileged software can potentially be malicious. The SGX security model assumes the CPU package hardware is trusted, while the main memory (DRAM) is untrusted as shown in Figure 1. Users are provided a private region of the main memory, referred to as *enclave*, that can store both code and data. SGX guarantees DRAM confidentiality, integrity, and freshness of the protected data region by leveraging the Memory Encryption Engine (MEE) [5] – a hardware component in SGX located within the memory controller. MEE protects data in the enclave through the integrity tree [3, 5] that is commonly used to enable a small amount of internal storage to protect a large amount of memory and often consists of multiple levels. Integrity tree has to be tamper resistant, and the root level data of integrity tree are stored on internal SRAM within the CPU package.

Although the lower levels of the integrity tree are stored in main memory, they can be trusted by verifying up to root level. As a result, data for all levels within the integrity tree needs to be checked to determine integrity and freshness for a single memory access to the enclave (or the protected data region). In other words, a single memory access to the protected data region results in multiple integrity tree data access from the main memory and significantly increases overall memory access latency. To improve overall performance, MEE cache [5] is added within MEE to store recently accessed integrity tree data and reduces the number of main memory accesses to improve performance. If the integrity tree data is already loaded into the MEE cache, it signifies that the integrity tree data already went through integrity and freshness check and was verified. Integrity check starts from the "leaf" and moves up to the root of integrity tree. As soon as a MEE cache hit occurs, MEE stops integrity check from continuing moving up to the root of integrity tree. In this work, we exploit this MEE cache to establish a covert channel attack.

2.3 Threat model

We assume a multi-core system that is shared by multiple tenants, with both the trojan and the spy residing on different cores within the same CPU, and their goal is to establish a covert channel. We assume the CPU supports SGX and hyperthreading. The system software does not provide any additional features to both trojan and spy, such as shared memory and hugepage, and the system software does not necessarily have any vulnerabilities.

3 CHALLENGES IN EXPLOITING MEE CACHE

MEE cache differs from other hardware on-chip cache since it caches integrity tree data. In particular, the challenges of MEE cache-based attack include the following.

1. MEE cache is only utilized when main memory's protected data region is accessed. Unlike data cache in CPU memory hierarchy, the MEE cache is only accessed when data is loaded from the protected data region. Thus, enclave's data needs to be accessed for our covert channel attack. However, since enclave data can still be cached in the on-chip cache hierarchy, `clflush` instruction is used to ensure access goes to the MEE cache. In addition, `clflush` does not flush the integrity tree data in the MEE cache.

2. For each data access to the protected data region, the number of MEE cache accesses can vary. MEE cache loads the integrity tree data with multiple levels, and multiple MEE cache access can occur. If a cache hit occurs within the MEE cache, no further access is done to move up the integrity tree. However, in any data access to the protected data region, the lowest level data of the integrity tree (or the versions data) is *always* accessed and checked as integrity check starts from the lowest level of the integrity tree. If a MEE cache hit occurs for the versions data, no more MEE cache access is done. Thus, we exploit the versions data in our MEE cache covert channel attack since its access is guaranteed to occur for each data access to a protected data region.

3. Hugepage is not available in enclave mode. Hugepages (e.g., 2MB, 1GB) are commonly supported in modern systems today. Hugepages have been utilized in LLC Prime+Probe attack [7, 9] since hugepage size can be larger than the LLC and enables continuous mapping of the virtual address space to the physically addressed cache. However, SGX does not support hugepage and thus, the covert channel attack is implemented with 4kB default page size.

4. RDTSC instruction is not available in enclave mode. To implement timing-based side channel attack, time needs to be measured, and instructions such as `rdtsc` instruction can be used (Figure 2(a)). However, `rdtsc` instruction is not available in the enclave mode in SGX CPU. As a result, `OCALL` function[1] can be used to execute `rdtsc` instruction from the enclave mode (Figure 2(b)); however, our evaluation shows that `OCALL` can take between 8000 to 15000

[1] `OCALL` function enables codes within the enclave to call external functions.

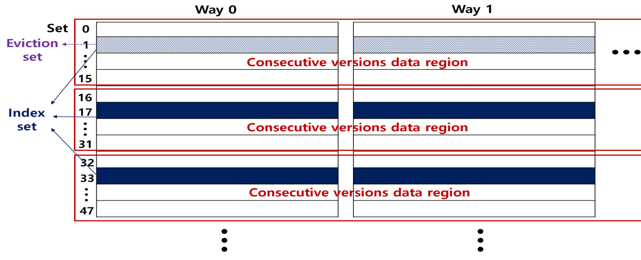

Figure 3: MEE cache organization for reverse-engineering the configuration.

clock cycles for the function call itself and results in significant overhead. To provide a low-overhead mechanism to estimate time, we exploit hyperthreading available in modern CPUs as well as the fact virtual address of non-enclave memory region can be directly accessed from within enclave mode (Figure 2(c)). A thread in the non-enclave mode continuously retrieves the time stamp counter value with `rdtsc` instruction and stores the values in the non-enclave memory space. A thread in the enclave mode retrieves the value directly from the non-enclave space with minimal cost (approximately 50 cycles).[2]

5. MEE cache structure is not publicly known. In the following section, we describe how we reverse-engineer the size and structure of the MEE cache.

4 REVERSE ENGINEERING THE MEE CACHE

In this section, we describe how we reverse-engineered the MEE cache organization. All analysis and evaluation are done on an Intel i7-6700K (Skylake) multi-core CPU, and our analysis suggests that the MEE cache is 64kB, 8-way set-associative cache with 128 sets.

4.1 MEE cache capacity

In conventional data cache, the working data set can be increased to determine the cache capacity by observing the change in data access latency. Hugepage can also be used to map a contiguous virtual address space to contiguous physical addresses. However, the same approach cannot be used for a MEE cache since hugepage is not available, and a single memory access to protected data region can cause multiple integrity tree data to be loaded into the MEE cache. In addition, data from each level of the integrity tree can be loaded into any location within the MEE cache and not necessarily loaded contiguously.

The cache line size of MEE cache is known to be 64B, and each data in the integrity tree is also 64B [5]. The integrity tree covers the versions data which are the counters used as part of compound nonce in SGX, and the versions data represents the lowest level of the integrity tree. The 64B of the versions data is used for integrity check of 512B data in the protected data region; thus, for a 4kB page size, 4kB/512= 8 versions data are guaranteed to be mapped contiguously within the MEE cache. In addition, the versions data are stored along with the corresponding Message Authentication Code (MAC) tag, referred to as PD_Tag, as the metadata – thus, the versions data is loaded into the odd index set of the MEE cache (while the PD_Tag is stored in the even index set). Thus, Figure 3 shows the consecutive versions data region within the MEE cache, and the *index set* describes the set of cache lines across different consecutive versions data region that have the same set index within that particular data region. The *eviction* set is subset of index set and is defined as the set of cache lines that have the same set index value (i.e., cache lines across the different cache ways).

[2] This approach is likely not needed in SGX2 [6] where support for `rdtsc` is provided. However, to the best of our knowledge, no current hardware supports SGX2.

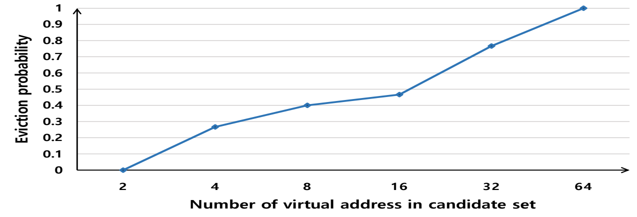

Figure 4: Result of eviction probability in 100 MEE cache size tests according to number of virtual address in candidate address set.

The *candidate address set* is defined as set of virtual addresses that can load the versions data into the same index set. To reverse engineer the MEE cache capacity, we used candidate address set that consists of virtual address with 4KB stride within the protected data region. If the size of versions data mapped to the candidate address set space is larger than the MEE cache capacity, then at least one of the versions data within the candidate address set needs to be evicted if all of the virtual addresses within the candidate address set are accessed. The 4KB stride access also minimizes versions data eviction caused by other levels' data of integrity tree. Figure 4 plots the eviction probability as the number of virtual addresses in the candidate address set is increased. Results are collected from running the experiment 100 times for a given number of virtual addresses. As the number of virtual addresses in the candidate address set increases, the eviction probability also increases and when there are 64 virtual addresses, the eviction probability reaches 100%. Therefore, MEE cache capacity can be calculated by multiplying the number of elements in the candidate address set (64) by the size of one cache way within consecutive versions data region ($16 \times$ 64B) and determine the overall MEE cache capacity as 64kB.

4.2 MEE cache associativity

In addition to the MEE cache size, the MEE cache associativity also needs to be known to determine the number of sets. Algorithm 1 summarizes how the MEE cache associativity can be determined. If the eviction address set can be determined, the number of elements in the eviction address set corresponds to the MEE cache associativity.

Lines 13-17 in the algorithm finds a set of virtual address from the candidate address set, whose versions data is loaded on the same index set. If the set's versions data evicts candidate's versions data, the candidate is not included in the set because the MEE cache set, where versions data of the candidate would be loaded, is already filled with other versions data of the set's addresses. The set includes at least one eviction address set if the number of element in candidate address set is equal or larger than 64 (the number of element in candidate address set when versions data eviction always occurs). Line 18-23 finds one test address from the rest of candidate address set to separate eviction address set from the set whose versions data is loaded on index set. A test address is chosen

Algorithm 1: Finding eviction address set

```
   input  : candidate address set – a set of all virtual address that
            can load the versions data into the same index set
   output : eviction address set – a set of virtual address whose
            versions data is loaded on the same eviction set
1  Function eviction test (set, victim) begin
2      access victim; flush victim ;
3      // load versions data on MEE cache but flush data from the LLC
4      mfence
5      foreach address ∈ set do
6          access address ; flush address ;
7      end
8      mfence
9      measure time to access victim ; flush victim ;
10     return time
11 end
12 candidate address set ← {A, A + 4KB, A + 8KB, ...}
13 index address set ← {}, eviction address set ← {}
14 foreach candidate ∈ candidate address set do
15     if eviction test (index address set, candidate) = main
         memory access latency with versions data hit then
16         insert candidate into index address set
17     end
18 end
19 foreach test ∈ (candidate address set − index address set)
     do
20     foreach address ∈ index address set do
21         access address ; flush address ;
22     end
23     mfence
24     if eviction test (index address set, test) = main memory
         access latency with level0 data hit then
25         foreach target ∈ index address set do
26             foreach address ∈ index address set do
27                 access address ; flush address ;
28             end
29             mfence
30             if eviction test (index address set − target, test) =
                 main memory access latency with versions data hit then
31                 insert target into eviction address set
32             end
33         end
34     end
35 end
```

if its versions data is evicted by the eviction address set. Finally, line 24-32 collects eviction address set by checking whether versions data of the test address is evicted or not by excluding address one by one from the index address set. If the versions data is not evicted, the excluded address is an element of the eviction address set. Based on the algorithm, we discover 8 virtual addresses in the eviction address set and determine the number of MEE cache ways to be 8.

5 MEE CACHE COVERT CHANNEL

Knowing the MEE cache configuration (i.e., 64kB 8-way set associative cache), in this section, we describe how covert channel can be established over the MEE cache. The covert channel exploits timing difference between MEE cache hit and miss when accessing the main memory's protected data region. MEE cache covert channel assumes the trojan and spy are running on different physical core. Trojan tries to evict spy's versions data in MEE cache to send bit signal. If spy's versions data is evicted by the trojan's versions data, spy's protected data region access latency will increase.

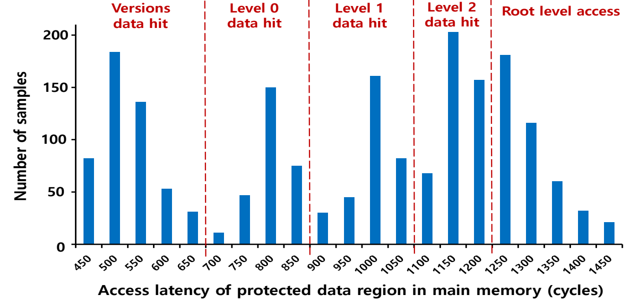

Figure 5: Histogram results of protected data region main memory access latency

5.1 MEE cache access latency

As described in the previous section, the versions data of the integrity tree is always checked in the MEE cache for a hit (or a miss) and thus, versions data will be used in our covert-channel attack. Since the covert-channel attack is based on the timing difference, we first analyze the MEE cache access latency and understand the timing difference between versions data hit or miss in the MEE cache when accessing the main memory's protected data region. In Figure 5, a latency distribution of main memory access is shown with different MEE cache hit (or miss) behavior. The result were obtained by accessing data in protected data region with different stride accesses, including 64B, 512B, 4KB, 32KB, and 256KB. With 64B and 512B stride access, 64B versions data loaded into the MEE cache (consisting of 8 counter values) covers 512B region of the memory – thus, results in effective spatial locality across multiple accesses and results in version data hit or in some cases, level 0 data with high probability. However, when accessing the main memory in 4KB, 32KB, and 256KB strides, there is a (cold) miss of the versions data but often results in level 1 or level 2 data hit within the MEE cache and results in higher latency. The results in Figure 5 show a clear trend of increasing latency as the level within the integrity tree that MEE cache hit occurs also moves up. While the difference between level 2 data hit or accessing the root level is relatively small, the different between the lowest level MEE cache hit (i.e., versions data hit) compared with a miss is rather significant. In this work, we exploit this timing difference between versions data hit and miss which have a difference of at least approximately 300 cycles.

5.2 Limitations of Prior covert channel attack

Prime+Probe [7] has been proposed for LLC covert channels and can also be applied to the MEE cache. The spy would have the eviction set for the versions data, and trojan has single virtual address whose versions data is conflicted with the eviction set. The trojan evicts one of the versions data in the spy's eviction set, and the spy decodes the signal by probing or measuring total latency.

Results (Figure 6(a)) from MEE cache covert-channel using Prime+Probe method shows that proper communication cannot be established with the spy. For LLC Prime+Probe [7], the timing difference

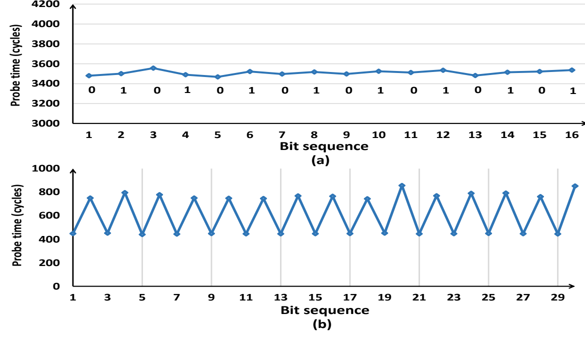

Figure 6: Results of MEE cache covert channel communication using (a) Prime+Probe and (b) this work. Trojan is sending '010101...' bit sequence in this example.

when probing between LLC hit and miss sufficiently large enough to determine if eviction occurred or not. However, the probing time of eviction set in the MEE cache is significantly high (compared to the timing difference between MEE cache hit or miss) because of the multiple main memory accesses required to access the data in the protected memory region – i.e., whether there is a MEE cache versions data hit or a miss, the main memory always needs to be accessed. As a result, each probe operation requires 8 main memory accesses (for the 8 cache ways), resulting in a probing latency that exceeds 3500 cycles. Thus, the difference of approximately 300 cycles for MEE cache hit or miss is insufficient for the covert channel given the inherent system noise and performance variations that can occur when accessing the main memory. Since the existing Prime+Probe is not suitable to MEE cache covert channel, we minimize the number of cache ways that need to be accessed in our implementation to ensure the difference in MEE cache access can be used for the covert channel.

5.3 MEE cache covert channel implementation

In Prime+Probe [7], the spy maintains the eviction set, and the trojan evicts one of the data. We also leverage a similar approach but in this work, we reverse the roles as the trojan maintains the eviction set, and the spy only needs to probe a single cache way. This enables the spy to leverage the MEE cache versions data hit or miss for the covert channel since the number of memory access is significantly reduced. The MEE cache covert channel consists of the following steps:

1. Prime : spy primes the MEE cache but only a single cache way.
2. Eviction : trojan evicts all cache ways if communicating a '1'.
3. Probe : spy probes the single cache way and primes a single cache way if MEE cache miss occurs.

Similar to other cache-based covert channel, the trojan and spy must first agree on same index in consecutive versions data region (Figure 3), and any arbitrary index can be used. The same index in consecutive versions data region can be selected by choosing the same 512B virtual address space unit within 4KB page. The trojan creates an eviction set by using Algorithm 1 and accesses the eviction set to communicate with the spy.

Before Prime step occurs, the spy utilizes the agreed index to determine the *monitor address* which is defined as the address that will be evicted by the trojan. After finding monitor address, MEE cache covert channel can be implemented based on Algorithm 2. One challenge with the trojan's eviction is the MEE cache replacement policy. While the details of the replacement policy are not

---

Algorithm 2: MEE cache covert channel protocol

$Data_{send}$[N], $Data_{recv}$[N]: N data bits to be sent and received
eviction set: a set of virtual address whose versions data is loaded on same MEE cache set
monitor address: virtual address whose versions data is conflicted with versions data of eviction set
T_{sync}: size of timing window shared between trojan and spy
T_{recv}: time for receiving data bits

Trojan's operation:

for $i \leftarrow 0$ *to* $N-1$ **do**
  if $Data_{send}[i] = 0$
    busy loop for time T_{sync}
  else if $Data_{send}[i] = 1$
    access *eviction set* in forward direction
    flush *eviction set* in forward direction
    mfence
    access *eviction set* in backward direction
    flush *eviction set* in backward direction
    busy loop for remaining time of T_{sync}
  endif
end

Spy's operation:

for *time* T_{recv} **do**
  measure $time$ to access *monitor address*
  flush *monitor address*
  if $time$ = *main memory latency with versions data hit*
    $Data_{recv} = 0$
  else if $time$ = *main memory latency with versions data miss*
    $Data_{recv} = 1$
  endif
  busy loop for remaining time of T_{sync}
end

---

publicly available, it can be assumed to an "approximate LRU" replacement policy, similar to other hardware cache. Thus, to ensure proper communication with the spy, the eviction process consists of two phases – a forward phase where the eviction set is accessed in the forward direction and a reverse phase where the opposite order is used. The two-phase approach comes at the cost of decrease in the communication bit rate but is necessary to reduce the error rate. While the eviction can take longer, the probe and prime stage for the next communication bit is overlapped since the probe of the MEE cache effectively primes the MEE cache for the next communication.

5.4 Evaluation

We tested and evaluated our covert channel attack on an Intel i7-6700K CPU (Skylake) with 4 physical cores and 32 GB of memory, with 128 MB for the MEE region. The system supports SGX and hyper-threading. We evaluated with Ubuntu 14.04 and SGX SDK v1.7. The trojan process and spy process were executed on different physical cores. Results from MEE cache-based covert-channel is shown in Figure 6(b) with a timing window of 15000 cycles. If the spy accesses protected data region and finds versions data miss (approximately 750 cycles), spy decodes it as a '1'. If the spy access results in versions data hit (approximately 480 cycles), the spy decodes it as a '0'.

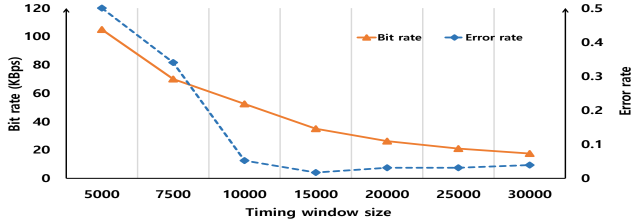

Figure 7: Trade-off between bit rate and error rate in MEE cache covert channel as timing window size is varied.

We evaluated our covert channel by varying the timing window size. For each timing window size, we measured bit rate and error rate of our covert channel. Figure 7 shows the trade-off between bit rate and error rate. Bit rate increases as the timing window size decreases. However, error rate significantly increases between 7500 cycles and 10000 cycles (5.2% → 34%). The trojan accesses eviction address set in forward and backward operation to send bit '1', and the latency of sending a single bit is approximately 9000 cycles. Therefore, error rate significantly increases if the timing window size is smaller than 9000 cycles. The MEE cache covert channel has the lowest error rate (1.7%) when the timing window size is 15000 cycles. We also evaluated the robustness in two noisy environments. First environment is when MEE cache is highly utilized by another physical core at the same time - i.e., another program frequently loads new integrity tree data from the main memory to the MEE cache. Two different access stride patterns (512B and 4KB) were used since it results in different MEE cache behavior. Second noisy environment is when general cache and main memory are intensively utilized. We utilized stress-ng tool to stress the main memory and cache [9, 11] and evaluated with 15000 cycles timing window size.

Figure 8 shows results under different environments when trojan sends '100100...' 128 bits sequence and red circles indicates error bit. Without any noise, there is only one error bit among 128 bits (Figure 8(a)). When additional noise is added with more accesses to the main memory through the on-chip cache hierarchy, the error rate has minimal impact since the MEE cache is not accessed (Figure 8(b)). However, if the MEE cache is highly utilized from additional noise (Figure 8(c),(d)), the error rate increases to 4 or 5 error bits during the 128 bits sequence.

5.5 Mitigations

There have been many prior work to defend against LLC covert channel attacks including detection works based on hardware performance counter [1, 4], replay confusion [15], cache partitioning [8], and new cache replacement policy [16]. However, as discussed earlier in Section 3, the MEE cache has very different characteristics and similar approaches used for LLC are not necessarily valid. In addition, recent LLC from some CPU vendors have proposed non-inclusive LLC and thus, carrying out covert channel attack based on shared LLC becomes more difficult. In comparison, this work proposes a new covert channel attack through the shared MEE cache. However, LLC defense mechanisms can be modified for the MEE cache by incorporating the characteristics of the MEE cache to prevent covert channel attack. For example, way-based partitioning [8] cannot be directly applied to MEE cache as simply partitioning the cache across different users will not work since the integrity tree is shared.

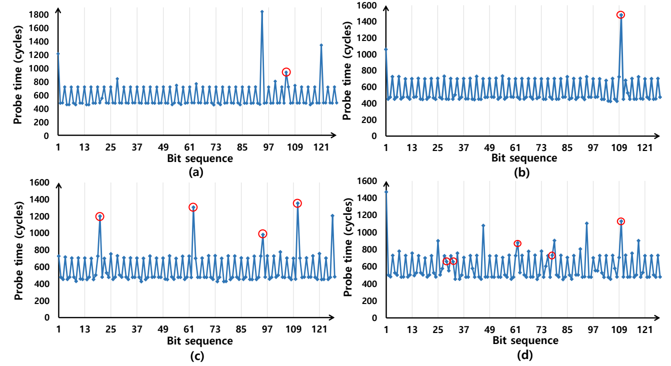

Figure 8: MEE cache covert channel results with (a) no noise, (b) noise from main memory access, and noise from additional integrity data access with (c) 512B and (d) 4kB stride.

6 SUMMARY

In this work, we proposed a novel cache covert channel attack by exploiting the MEE cache available in Intel SGX systems. We identified the challenges in leveraging the MEE cache and described how the MEE cache structure was reverse-engineered. We implemented our MEE cache-based covert channel to demonstrate the feasibility of using this new cache architecture for covert-channel attack.

ACKNOWLEDGMENTS

We thank Professor Brent B. Kang for helpful discussions. This research was supported in part by the National Research Foundation of Korea (NRF) by the Ministry of Science, ICT & Future Planning (MSIP)(NRF-2017R1A2B4011457), ONR (N00014-18-1-2661) grants, and MSIP, under the Human Resource Development Project for Brain Scouting Program(IITP-2017-0-01889) supervised by the IITP(Institute for information & communications Technology Promotion).

REFERENCES

[1] S. Briongos et al. Cacheshield: Protecting legacy processes against cache attacks. *CoRR*, abs/1709.01795, 2017.
[2] V. Costan et al. Intel SGX explained. *IACR Cryptology ePrint Archive*, 2016, 2016.
[3] B. Gassend et al. Caches and hash trees for efficient memory integrity verification. In *HPCA*, 2003.
[4] D. Gruss et al. Flush+flush: A fast and stealthy cache attack. In *DIMVA*, 2016.
[5] S. Gueron. A memory encryption engine suitable for general purpose processors. *IACR Cryptology ePrint Archive*, 2016.
[6] Intel. Intel 64 and IA-32 Architectures Software Developer's Manual Volume 3D: System Programming Guide, Part 4, 2016.
[7] F. Liu et al. Last-level cache side-channel attacks are practical. In *IEEE Symposium on Security and Privacy*, 2015.
[8] F. Liu et al. Catalyst: Defeating last-level cache side channel attacks in cloud computing. In *HPCA*, 2016.
[9] C. Maurice et al. Hello from the other side: SSH over robust cache covert channels in the cloud. In *NDSS*, 2017.
[10] C. Percival. Cache missing for fun and profit. In *Proceedings of BSDCan*, 2005.
[11] P. Pessl et al. DRAMA: exploiting DRAM addressing for cross-cpu attacks. In *USENIX Security*, 2016.
[12] T. Ristenpart et al. Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds. In *CCS*, 2009.
[13] D. Sullivan et al. Microarchitectural minefields: 4k-aliasing covert channel and multi-tenant detection in iaas clouds. In *NDSS*, 2018.
[14] Z. Wu et al. Whispers in the hyper-space: High-bandwidth and reliable covert channel attacks inside the cloud. *IEEE/ACM Trans. Netw.*, 2015.
[15] M. Yan et al. Replayconfusion: Detecting cache-based covert channel attacks using record and replay. In *MICRO*, 2016.
[16] M. Yan et al. Secure hierarchy-aware cache replacement policy (SHARP): defending against cache-based side channel atacks. In *ISCA*, 2017.
[17] Y. Yarom et al. FLUSH+RELOAD: A high resolution, low noise, L3 cache side-channel attack. In *USENIX Security*, 2014.