An Off-Chip Attack on Hardware Enclaves via the Memory Bus

Dayeol Lee
UC Berkeley

Dongha Jung
SK Hynix

Ian T. Fang
UC Berkeley

Chia-Che Tsai
Texas A&M University

Raluca Ada Popa
UC Berkeley

Abstract

This paper shows how an attacker can break the confidentiality of a hardware enclave with MEMBUSTER, an *off-chip* attack based on snooping the memory bus. An attacker with physical access can observe an unencrypted address bus and extract fine-grained memory access patterns of the victim. MEMBUSTER is qualitatively different from prior on-chip attacks to enclaves and is more difficult to thwart.

We highlight several challenges for MEMBUSTER. First, DRAM requests are only visible on the memory bus at last-level cache misses. Second, the attack needs to incur minimal interference or overhead to the victim to prevent the detection of the attack. Lastly, the attacker needs to reverse-engineer the translation between virtual, physical, and DRAM addresses to perform a robust attack. We introduce three techniques, *critical page whitelisting*, *cache squeezing*, and *oracle-based fuzzy matching algorithm* to increase cache misses for memory accesses that are useful for the attack, with no detectable interference to the victim, and to convert memory accesses to sensitive data. We demonstrate MEMBUSTER on an Intel SGX CPU to leak confidential data from two applications: Hunspell and Memcached. We show that a single uninterrupted run of the victim can leak most of the sensitive data with high accuracy.

1 Introduction

Hardware enclaves [1–5] provide secure execution environments to protect sensitive code and data. A hardware enclave has a small trusted computing base (TCB) including the trusted hardware and program and assumes a strong threat model where even a privileged attacker (e.g., hypervisor, OS) cannot break the confidentiality and integrity of the execution. In such a threat model, the attacker cannot physically attack the internals of the processor package, but can attempt to tamper with or observe the externals of the processor (e.g., Cold-Boot attacks [6]). As a result, hardware enclaves are attractive for protecting privacy-sensitive workloads such as database [7], big data [8–10], blockchains [11–15], and machine learning [16, 17].

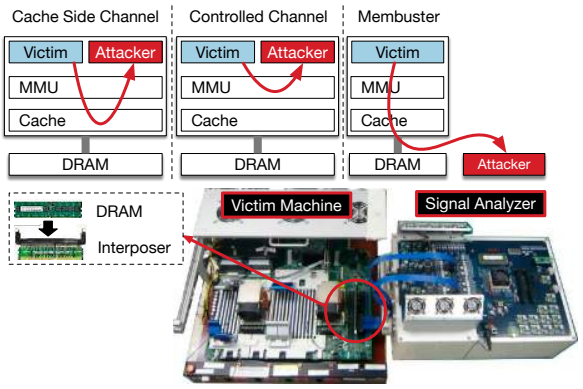

Figure 1: On-chip side channels compared to MEMBUSTER. The cache side-channel attack leaks addresses through a shared cache, whereas the controlled-channel attack uses adversarial memory management. MEMBUSTER leaks addresses directly through the off-chip memory bus. The photo shows an example hardware setup for the attack.

Along with the proliferation of hardware enclaves, many side-channel attacks against them have been discovered [18–23]. Side-channel attacks leak sensitive information from enclaves via architectural or microarchitectural states. For instance, controlled-channel attacks [24] use the OS privilege to trigger page faults for memory access on different pages, to reconstruct secrets from page-granularity access patterns inside the victim program. We categorize these attacks as *on-chip side-channel attacks*, where the attacker uses adversarial or shared on-chip components to reveal memory addresses accessed by the victim (Figure 1).

An attacker who can *physically* access the machine can perform an *off-chip side-channel attack* that directly observes the memory addresses on the *memory bus*. The memory bus, which consists of a *data bus* and an *address bus*, delivers memory requests from a CPU to an off-chip DRAM. Although the CPU encrypts the data of an enclave, all the addresses still leave the CPU unencrypted, allowing the attacker to infer program secrets from the access patterns. Since off-the-shelf DRAM interfaces do not support address bus encryption, no

existing hardware enclave can prevent physical attackers from observing the memory address bus.

Several studies have hinted at the possibility of attacks based on the memory address bus [25–27]. Costan *et al.* [26] suggest the possibility of tapping the address bus, but acknowledge that they are not aware of any successful example of the attack. Maas *et al.* [25] suggest that an attacker who can collect physical memory traces of a database server can distinguish two different SQL queries operating on the same dataset. However, to the best of our knowledge, no work has shown how such a side channel can be exploited to break the confidentiality of an enclave.

In this paper, we present MEMBUSTER, an off-chip side-channel attack on the memory address bus. We show that MEMBUSTER can be a substantial threat to hardware enclaves because of its unique traits compared to the existing on-chip attacks (§2.2). The need for off-chip access, despite being a disadvantage, advantages the attacker as it makes MEMBUSTER much harder to mitigate with *protected-access* solutions (Table 1). Recently, a wide range of tools [28–32] have been developed for mitigating on-chip side-channel attacks for enclaves with a reasonable overhead. These tools either partition the resources (e.g., cache) to prevent an attacker from learning information via shared resources or intercept actions (e.g., page faults) to prevent an attacker from observing the side channels. At their core, these solutions attempt to protect the memory accesses from an attacker's sight.

However, these protected-access solutions do not prevent MEMBUSTER, which observes the memory addresses off-chip and thus can bypass the protection of any on-chip solutions. To prevent MEMBUSTER on the current hardware enclave design, one must *hide* the accessed memory addresses, by making the enclave execution *oblivious* to the secret data. This requires either using oblivious algorithms [33] inside the enclave or running the enclave atop an ORAM [34, 35]. Both mechanisms bring significant performance overhead to the enclave. An alternative is to change the CPU and DRAM design to encrypt the address bus, but implementing a decryption module in DRAM can be expensive [36, 37].

We describe the challenges to perform a robust off-chip attack as follows: **(1) Address Translation.** The attacker needs to translate the DRAM requests into the physical addresses by reverse-engineering the mapping and to further translate them into virtual addresses of the victim enclave; **(2) Lossy Channel.** The attacker only sees DRAM requests when cache misses or write-back occurs. Since most modern CPUs have a large last-level cache (LLC), a significant portion of memory accesses do not issue any DRAM requests. We show why simple methods such as *priming* the cache does not incur sufficient cache misses needed for the attack; **(3) Unusual Behaviors in SGX.** SGX has unique memory behaviors which increase the difficulty of the attack. For example, we show that common architectural features such as disabling the cache do not work in SGX. We also find that *paging* in SGX hides most of the memory accesses.

We first show how an attacker can translate the DRAM requests, and can filter out irrelevant addresses to leave only the *critical* addresses that are useful for the attack. Then, we introduce two techniques, *critical page whitelisting* (§5.2) and *cache squeezing* (§5.4.1), to increase useful cache misses by thwarting page swaps and shrinking the effective cache for the critical addresses. With more cache misses, the attacker can observe more DRAM requests. These techniques do not cause detectible interference to the victim, and can be combined with cache priming to make more memory accesses visible to the attacker. Our *oracle-based fuzzy matching algorithm* (§6) can create an "oracle" of the secret-to-access-pattern mapping, to identify the sensitive accesses from a sizable memory bus trace. We then extract the sensitive data from the noisy memory accesses by fuzzy-matching the accesses against the oracle. We further show that *hardware prefetching* can increase the efficiency of this algorithm in MEMBUSTER.

We demonstrate the attack by attaching Dual In-line Memory Module (DIMM) interposer to a production system with an SGX-enabled Intel processor and a commodity DDR4 DRAM. We capture the memory bus signals to perform an off-line analysis. We use two applications, Hunspell and Memcached, to demonstrate the attack. Finally, we show the scalability of our techniques by simulating the attack in modified QEMU [38].

To summarize, the contributions of this paper are as follows:

- The setup of an off-chip side-channel attack on hardware enclaves and identification of the challenges for launching the attack robustly.
- Effective techniques for maximizing the side-channel information with no detectible interference nor order-of-magnitude performance overhead to the victim program.
- A fuzzy comparison algorithm for converting the address trace collected on the memory bus to program secrets.
- Demonstration and experimentation of the attack on an actual Intel SGX CPU. To our best knowledge, it is the first work that shows the practicality of the attack.

The security implications of the off-chip side-channel attacks can be pervasive because such a channel exists on almost every secure processor with untrusted memory. We hope to motivate further research by alarming the community about the practicality and severity of such attacks.

2 Background and Related Work

In this section, we discuss the background, including hardware enclaves, known on-chip side-channel attacks on SGX, and the related defenses.

2.1 Intel SGX

We choose Intel SGX [39] as the primary attack target because Intel SGX has the most mature implementation and the strongest threat model against untrusted DRAM. SGX is

	Brasser et al. [20]	Schwarz et al. [21]	CacheZoom [22]	FLUSH-based [23]	Controlled [24]	MEMBUSTER
Software-Only	✓	✓	✓	✓	✓	✗
Protected-Access Fix [28–32]	✓	✓	✓	✓	✓	✗
Root Adversary	✓	✗	✓	✓	✓	✓
Noiseless	✗	✗	✗	✓	✓	✓
Lossless	✗	✗	✗	✓	✓	✗
Fine-Grained (64B vs. 4KB)	✓	✓	✓	✗	✗	✓
No Interference (e.g., AEX)	✓	✓	✗	✗	✗	✓
Low Overhead	✓	✓	✗	✗	✗	✓

Table 1: This work (MEMBUSTER) compared to previous side-channel attacks on SGX. The two boldface rows illustrate what we perceive to be the most important distinctions. The colored cell indicates the attacker has the advantage.

a set of instructions for supporting hardware enclaves introduced in the Intel 6th generation processors. SGX assumes that only the processor package is trusted; all the off-chip hardware devices, including the DRAM and peripheral devices, are considered potentially vulnerable or compromised. The threat model of SGX also includes physical attacks such as Cold-Boot Attacks [6], which can observe sensitive data from residuals inside DRAM.

An Intel CPU with SGX contains a memory encryption engine (MEE), which encrypts and authenticates the data stored in a dedicated physical memory range called the *enclave page cache* (EPC). The MEE encrypts data blocks and generates authentication tags when sending the data outside the CPU package to be stored inside the DRAM. To prevent roll-back attacks, the MEE also stores a version tree of the protected data blocks, with the top level of the tree stored inside the CPU. For Intel SGX, EPC is a limited resource; the largest EPC size currently available on an existing Intel CPU is 93.5 MB, out of 128 MB Processor's Reserved Memory (PRM). The physical pages in EPC, or EPC pages, are mapped to virtual pages in enclave linear address ranges (ELRANGEs) by the untrusted OS. If all concurrent enclaves require more virtual memory than the EPC size, the OS needs to swap the encrypted EPC pages into regular pages.

However, even with MEE, Intel SGX does not encrypt the addresses on the memory bus. As previously discussed, changing the CPU to encrypt the addresses requires implementing the encryption logic on DRAM, and thus requires new technologies such as Hybrid Memory Cube (HMC) [36, 37].

The unencrypted address bus opens up a universal threat to hardware enclaves with external encrypted memory. Komodo [40], ARM CryptoIsland [41], Sanctum [5], and Keystone [4] do not encrypt data for an external memory by default. AMD SEV [42] allows hypervisor-level memory encryption, but also does not encrypt addresses.

2.2 Comparison with Existing Attacks

In this section, we discuss how MEMBUSTER can be a substantial threat to hardware enclaves because of its unique traits. We compare MEMBUSTER with various on-chip side-channel attacks on SGX [20–24] in Table 1.

2.2.1 Side Channel Attacks on SGX

PRIME+PROBE. A shared cache hierarchy allows an adversary to infer memory access patterns of the victim using known techniques such as PRIME+PROBE [43, 44]. However, in PRIME+PROBE, the attacker usually cannot reliably distinguish the victim's accesses from *noises* of other processes. The PRIME+PROBE channels are also *lossy*, as the attacker may miss some of victim's accesses while probing.

Brasser *et al.* [20] demonstrate PRIME+PROBE on Intel SGX without interfering with the enclave, but the attack requires running the victim program repeatedly to compensate for its noise and signal loss. Schwarz *et al.* [21] show that the attacker can alleviate the noise by identifying cache sets that are critical to the attack. This technique can be applied to applications that have data-dependent accesses in a small number of cache sets. CacheZoom [22] also uses PRIME+PROBE but minimizes the noise by inducing Asynchronous Exits (AEXs) every few memory accesses in the victim. This incurs a significant overhead on enclaves, and also makes the attack easily detectable [32].

Flush-based Side Channels. Other techniques such as FLUSH+RELOAD [45] and FLUSH+FLUSH [46] use a shared cache block between the attacker and the victim to create a noiseless and lossless side channel. However, these techniques cannot be directly applied to enclave memory, because an enclave does not share the memory with other processes. However, these techniques can still be used to observe the page table walk for enclave addresses [23]. Specifically, the attacker can monitor the target page tables with a tight FLUSH+RELOAD loop. As soon as the loop detects page table activities, the attacker interrupts the victim and infers page-granularity addresses. Similar to CacheZoom, this attack incurs a significant AEX overhead and thus can be detected by the victim.

Controlled Channels. Controlled-channel attacks [24] take advantage of the adversarial memory management of the untrusted OS, to capture the access patterns of an SGX-protected execution. Even though Intel SGX masks the lower 12 bits of the page fault addresses to the untrusted OS, controlled-channel attacks use sequences of virtual page numbers to differentiate memory accesses within the same page. The controlled channel is noiseless and lossless but can be detected and mitigated as it incurs a page fault for each sequence of accesses on the same page [28, 31].

2.2.2 Advantages of MEMBUSTER

As shown in Table 1, MEMBUSTER creates a noiseless side channel by filtering out all of the non-victim memory accesses, leaving only addresses that are useful for the attack. It can observe memory accesses with cache line granularity. Also, MEMBUSTER does not incur interference such as AEX or page fault to the victim and needs not to incur an order-of-magnitude overhead.

Several recent mechanisms, such as Varys [28], Hyperrace [29], Cloak [30], T-SGX [31], or Déjà Vu [32], have been proposed to prevent the attacker from observing memory access patterns in the victim. In general, PRIME+PROBE can be mitigated by partitioning the cache to shield the victim from on-chip attackers. This does not defeat an off-chip attacker who directly observes DRAM requests. T-SGX [31] and Déjà Vu [32] have proposed to use the Intel Transactional Synchronization Extensions (TSX) to prevent AEX or page faults from an enclave. These techniques are based on thwarting the interference (e.g., AEX, page faults) that causes the side channels [22–24]. However, MEMBUSTER does not incur such interference on enclaves, and thus cannot be thwarted through similar approaches. To our best knowledge, there is no reliable way to detect or mitigate MEMBUSTER using existing on-chip measures.

2.2.3 Related Work

Other On-Chip Attacks. Other on-chip attacks worth mentioning are speculative-based execution side channels like Foreshadow [18] or ZombieLoad [47], branch shadowing side channels [48], denial-of-service attacks (e.g., Rowhammer [49, 50]), or rollback attacks [51, 52].

Other Off-Chip Side-Channel Attack. DRAM row buffers can be exploited as side-channels between cores or CPUs, as demonstrated in DRAMA [53]. DRAMA shows that by observing the latency of reading or writing to DRAM, the attacker can infer whether the victim has recently accessed the data stored in the same row. DRAMA shows how a software-only attacker can use DRAM row buffers as covert channels or side channels. MEMBUSTER further explores how the attacker can directly use the address bus as a side channel.

3 MEMBUSTER

In this section, we describe the basic attack model of MEMBUSTER. In further sections, we will refine and improve the attack. At a high level, the attacker first sets up an environment to collect the DRAM signals and waits until the victim executes some code containing data-dependent memory accesses. The attacker translates the collected signals into cache-line granularity virtual addresses.

3.1 Threat Model

We assume the standard Intel SGX threat model in which nothing but the CPU package and the victim program is trusted. Everything else, including the OS or other applications, is untrusted and can be controlled by the attacker. External hardware devices are also untrusted, so the attacker can tap the address bus to the external DRAM. For the advanced techniques discussed in §5, the attacker may also use the root privilege to install the modified SGX driver.

To tap the memory bus, the attacker needs to have physical access to the machine where the victim is running. Such an assumption eliminates the possibility of remote attacks through either cloud environments or network connections. The candidates who may perform MEMBUSTER could be two types. On the server-side, these may include the employees of a cloud provider, or IT administrators of an institution, who act as insiders to leak sensitive information. On the client-side, end users may want to attack the local hardware enclaves, which protect proprietary data (e.g., licenses, digital properties, etc). We assume that the attacker has enough budget and knowledge to acquire and install the DIMM interposer for the attack described in §3.2. This might be an obstacle for the general public, but we claim that the cost is manageable if the attacker has a strong motivation for obtaining the data.

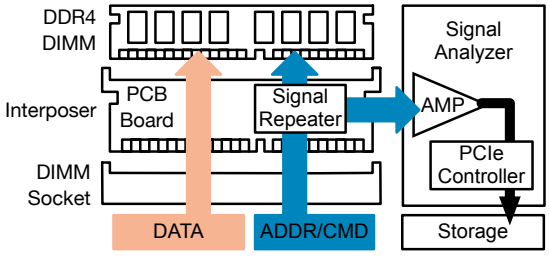

Figure 2: Hardware setup for a memory bus side-channel attack. DIMM interposer collects the bus signals and sends them to the signal analyzer. The attacker can use the analyzed signals to learn the memory access pattern of the victim.

Like in the controlled channel and cache side channels, MEMBUSTER assumes that the adversary has knowledge of the victim application, by either consulting the source code or reverse-engineering the application. The adversary is also aware of the runtime used by the victim application for platform support, such as the SDK libraries, library OSes, or shield systems. In our experiments, we use Graphene-SGX [54] for platform support of the victim applications. Address Space Layout Randomization (ASLR) in the library OSes or the runtimes may complicate the extraction of secret information but generally is insufficient to conceal the access patterns completely [24]. ASLR offered by the host kernel is irrelevant because a hostile host kernel can either control or monitor the addresses where the victim enclaves are loaded.

3.2 Hardware Setup for the Attack

Figure 2 shows a detailed hardware setup for the MEMBUSTER attack. The hardware setup may vary on different CPU models and vendors. The attacker installs an *interposer* on the DIMM socket prior to system boot. The interposer is a custom printed circuit board (PCB) that can be placed between the DRAM and the socket. The interposer contains a signal repeater chip which duplicates the command bus signals and sends them to a *signal analyzer*. The analyzer amplifies the signals and then outputs the signals to a storage server through a PCIe interface.

In the rest of the section, we will highlight the key requirements in successfully performing the attack.

Sampling Rate. The sampling rate of the interposer needs to

be equal or higher than the clock rate of the DIMM in order to capture all the memory requests. A standard DDR4 clock rate ranges from 800 to 1600 MHz, while a DIMM typically supports between 1066 (DDR4-2133) and 1333 (DDR4-2666) MHz. To match with the sampling rate, the attacker can lower the DIMM clock rate if it is configurable in the BIOS.

Recording Bandwidth. The sampling rate also determines the *recording bandwidth*. For example, DDR4-2400 (1200 MHz) has a 32-bit address and a 64-bit data bus, thus the recording bandwidth for the address bus is 1200 Mbps$\times$32 bits = 4.47 GiB/s. For reference, the data bus of a DDR has a 2$\times$ transfer rate, as well as a 2$\times$ transfer size. Hence, the bandwidth for logging all the data on DDR4-2400 will be 17.88 GiB/s.

Acquisition Time Window. The *acquisition time window* (i.e., the maximum duration for collecting the memory commands) determines the maximum length of execution that the attacker can observe. The acquisition time window equals the *acquisition depth* (i.e., the analyzer's maximum capacity of processing a series of contiguous sample) divided by the recording bandwidth of the interposer. For example, with 64 GiB acquisition depth, the analyzer can process and log the commands from DDR4-2400 up to $\sim$ 14 seconds.

We surveyed several vendors which offer DIMM analyzers [55–57] for purchase or rental. Among them, the maximum sampling rate can reach 1200-1600 MHz, and the acquisition depth typically ranges between 4-60 GiB. One of the devices [55] can extend the acquisition time window to $>$ 1 hour by attaching 16 TB SSD and streaming the compressed log via PCIe at 4.8 GiB/s. Another device [57] does not disclose the memory depth but specifies that it can capture up to 1G (10^9) samples. The cost of the analyzer varies depending on the sampling rate and the acquisition depth. At the time of writing, *Kibra 480* [56] (1200 MHz, 4 GiB) costs $6,500 per month, *MA4100* [57] (1600 MHz, 1G-samples) costs $8,000 per month, and *JLA320A* [55] (1600 MHz, 64 GiB) costs $170,000 for purchase.

3.3 Interpreting DRAM Commands

Once the attacker has finished setting up the environment, she can collect the DRAM signals at any point in time, and analyze the trace off-line. As the first step, the attacker interprets the DRAM commands collected from the interposer.

A modern DRAM contains multiple banks that are separated into bank groups. Within each bank, data (often of the same size as the cache lines) are located by rows and columns. Each bank has a row buffer (i.e., a sense amplifier) for temporarily holding the data of a specific row when the CPU needs to read or write in the row. Because only one row can be accessed in a bank at a time, the CPU needs to reload the row buffer when accessing a data block in another row.

The log collected from the DRAM interposer typically consists of the following commands:

- `ACTIVATE(Rank,Bank,BankGroup,Row)`: Activating a specific row in the row buffer for a certain rank, bank, and bank group.
- `PRECHARGE(Rank,Bank,BackGroup)`: Precharging and deactivating the row buffer for a certain rank, bank, and bank group.
- `READ(Rank,Bank,BankGroup,Col)`: Reading a data block at a specific column in the row buffer.
- `WRITE(Rank,Bank,BankGroup,Col)`: Writing a data block at a specific column in the row buffer.

Other commands such as PDX (Power Down Start), PDE (Power Down End), and AUTO (Auto-recharge) are irrelevant to the attack and thus omitted from the logs.

Based on the DRAM commands, we can construct the rank, bank, row, and column of each trace, by simply tracing the activated row within each bank. Note that the final traces are also time-stamped by the clock counter of the analyzer. The result of the translation is a sequence of logs containing the timestamp, access type (read or write), rank, bank, row, and column in the DRAM.

3.4 Reverse-engineering DRAM Addressing

A physical address in the CPU does not linearly map to a DRAM address consisting of rank, bank, row, and column. Instead, the memory controller translates the address to maximize DRAM bank utilization and minimize the latency. The translation logic heavily depends on the CPU and DRAM models, and Intel does not disclose any information. Thus, the attacker needs to reverse-engineer the internal translation rule for the specific set of hardware. This has been also done by a previous study [53].

We use the traces collected from the DRAM interposer to reverse-engineer the addressing algorithm of an Intel CPU. For attacking the enclaves, we only need a part of the addressing algorithm that affects the range of the enclave page cache (EPC). We write a program running inside an enclave, which probes the DRAM addresses translated from the EPC addresses. The probing program allocates a heap space larger than the EPC size (93.5MB). For every cache line in the range, the program generates cache misses by repeatedly flushing the cache line and fetching it into the cache. By accessing each cache line multiple times, we can differentiate the traces caused by probing from other memory accesses in the background and minimize the effect of re-ordering by the CPU's memory controller. The techniques in §3.5 are also needed for translating the probed virtual addresses to physical addresses.

Using the DRAM traces generated by probing cache lines inside the EPC, we can create a direct mapping between the physical addresses and DRAM addresses (ranks, banks, bank groups, rows, and columns). We further deduce the addressing function of the target CPU (i5-8400), by observing the changing bits in the physical addresses when DRAM addresses change. We conclude that the addressing function on i5-8400 is as shown in Figure 3. Other CPU models may implement a different addressing function, and reverse-engineering should

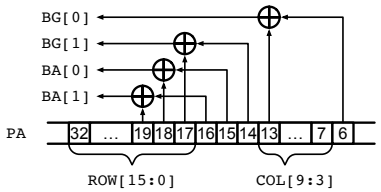

Figure 3: The reverse engineered addressing function of the i5-8400 CPU. The function translate a physical address (PA) to the Bank Group (BG), Bank Address (BA), Row (ROW) and Column (COL) within the DRAM.

be done for each CPU model beforehand.

3.5 Translating PA to VA

In order to extract the actual memory access pattern of the victim, we need to further translate the physical addresses into more meaningful virtual addresses. In general, a root-privileged attacker has multiple ways of obtaining the physical-to-virtual mappings: either by parsing the proc file `/proc/[PID]/pagemap` (assuming Linux as the OS), or using a modified driver. However, paging in an enclave is controlled by the SGX driver, and the vanilla driver forbids poking the physical-to-virtual mappings through the proc file system. Nevertheless, the attack can still modify the SGX driver to retrieve the mappings, and this is what we do.

Hence, we print the virtual-to-physical mappings in the dmesg log and ship the log together with the memory traces. During our offline analysis, we use the dmesg log as an input to the attack script. The dmesg log also contains system timings of paging, and can be further calibrated to the timestamps of the collected traces. Because paging in an enclave needs to copy the whole pages in and out of the EPC a sequential access pattern of a whole or partial page will appear in the memory traces. After calibration, we successfully translate all the physical addresses to virtual addresses.

4 Attack Examples

We show how MEMBUSTER exploits two example applications: (1) spell checking of a confidential document using *Hunspell*, and (2) email indexing cache using *Memcached*.

4.1 Hunspell

Hunspell is an open-source spell checker library widely used by LibreOffice, Chrome, Firefox and so on [58]. The controlled-channel attack [24] has shown that Hunspell is exploitable by page-granularity access patterns, which motivated us to use it as the first target of MEMBUSTER. We make the same assumptions as described in [24]; the attacker tries to infer the contents of a confidential document owned by a victim while Hunspell is spell-checking. The attacker knows the language of the document, and therefore can also obtain the same dictionary, which is publicly available.

The side-channel attacks on Hunspell are based on observing the access patterns for searching words in a hash table

```
1  // add a word to the hash table
2  int HashMgr::add_word(const std::string& word) {
3    struct hentry* hp = (void*) malloc(sizeof(struct
         hentry) + word->size());
4    struct hentry* dp = tableptr[i]; // Populate hp
5    while (dp->next != NULL) {
6      if (strcmp(hp->word, dp->word) == 0) {
7        free(hp); return 0;
8      }
9      dp = dp->next;
10   }
11   dp->next = hp;
12   return 0;
13 }
14 // lookup a word in the hash table
15 struct hentry* HashMgr::lookup(const char* word) {
16   struct hentry* dp;
17   if (tableptr) {
18     dp = tableptr[hash(word)];
19     for (; dp != NULL; dp = dp->next) {
20       if (strcmp(word, dp->word) == 0) return dp;
21     }
22   }
23   return NULL;
24 }
```

Figure 4: The Hunspell code which leaks access patterns with controlled-channel attacks and MEMBUSTER.

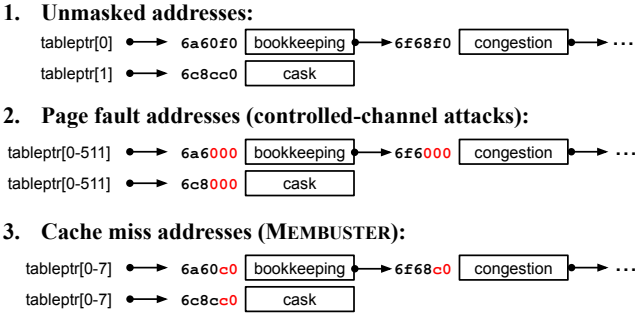

Figure 5: Observerable address patterns in Hunspell by different attacks. Controlled-channel attacks only see page-fault addresses without the lower 12 bits, whereas MEMBUSTER can see LLC-miss addresses without the lower 6 bits.

created from the dictionary. A simplified version of the vulnerable code is shown in Figure 4. The Hunspell execution starts with reading the dictionary file and inserting the words into the hash table by calling `HashMap::add_word()`. For each word from the dictionary, `HashMap::add_word()` allocates a `hentry` node and inserts it to the end of the linked list in the corresponding hash bucket. Then, Hunspell reads the words for spell-checking and calls `HashMap::lookup()` to search the words in the hash table. Both `HashMap::add_word()` and `HashMap::lookup()` leak the hash bucket of the word currently being inserted or searched, and all the `hentry` nodes before the word is found in the linked list.

The controlled-channel attack leaks different access patterns from those that we observe on our memory bus attack, as the example shown in Figure 5. Controlled-channel attacks

leak access patterns through page fault addresses, which are masked by SGX in the lower 12 bits. However, for applications like Hunspell, controlled-channel attacks can use sequences of page fault addresses to infer more fine-grained access patterns within a page. For example, although the nodes for `bookkeeping` and `booklet` are on the same page, the controlled-channel attacks can differentiate the accesses by the page addresses accessed before reading the nodes.

On the other hand, our memory bus channel can leak the addresses of each cache line being read from and written back to DRAMs, making the attacks more fine-grained than controlled-channel attacks. The attacks can differentiate the access patterns based on the addresses of each node accessed during lookups, instead of inferring through the address sequences. The granularity of memory bus attacks makes it possible to extract sensitive information even if the access patterns are partially lost due to caching.

4.2 Memcached

Memcached [59] is an in-memory key-value database, which is generally used to speed up various server applications by caching the database. Memcached is used in various services such as Facebook [60] and YouTube [61]. In this example, we assume that Memcached runs in an SGX enclave, as part of a larger secure system (e.g., secure mail server).

We consider the scenario discussed by Zhang *et al.* [62], where a mail server indexes the keywords in each of the emails and the attacker can inject an arbitrary email to the victim's inbox by simply sending an email to the victim. As shown in Figure 6, we assume that the index data is stored in Memcached running in an SGX enclave. Since the attacker owns the machine, she can also perform MEMBUSTER by observing the memory bus. The attacker's goal is to use his abilities to reveal the victim's secret emails `A`, `B`, and `C`.

Memcached does not have any data-dependent control flow, but the attacker can use the memory bus side channel to infer the query sent to Memcached. Memcached stores all keys in a single hash table `primary_hashtable` defined in `assoc.c` using the Murmur3 hash of a key as an index. Each entry of the hash table is linearly indexed by the Murmur3 hash of the key. Thus Memcached will access an address within the hash table whenever it searches for a key. By observing the address, the attacker can infer the hash of the key.

Memcached dynamically allocates the hash table at the beginning of the application. The attacker can easily find out the address of the hash table by sending a malicious email to make Memcached access the hash table. For example in Figure 6, the attacker sends an email `D` which contains a word "Investment". Memcached accesses the entry, and the attacker observes the address. Since the attacker already knows the hash value of the key, she can easily find out the address of the hash table.

Next, the attacker keeps observing the memory accesses within the hash table. Once the attacker figures out the hash table address, she can reveal the hash values of the query, by observing the virtual addresses accessed by Memcached. To match the hash values with words, the attacker pre-computes some natural words and creates a hash-to-word mapping. Even though hashes can conflict, we show that the attacker can recover most of the words by just picking a most-common word based on the statistics.

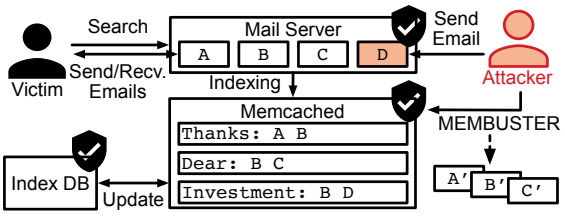

Figure 6: An example attack scenario where a mail server uses Memcached as an index database. `A`, `B`, `C` and `D` are the emails.

5 Increasing Critical Cache Misses

As previously discussed, the basic attack model of MEMBUSTER can observe memory transactions with cache-line granularity when the memory transactions cause cache misses in the last-level cache (LLC). Such an attack model is weakened in a modern processor with a large LLC ranging from 4 MB to 64 MB, causing only a small fraction of memory transactions to be observable on the DRAM bus.

In this section, we introduce techniques to increase cache misses of the target enclaves. In a realistic scenario, an attacker only cares about increasing the cache misses within the virtual address range which leaks the side-channel information. Take the attack on Hunspell for example, the attacker only needs to observe the access on the nodes which store the dictionary words. We called a memory address as *critical* if the address is useful for the attack. Our goal is to increase the cache misses on critical addresses, to improve the success rate of the MEMBUSTER attack.

5.1 Can We Disable Caching?

A simple solution to increase cache misses is to disable caching in the processor. On x86, entire cacheability can be disabled by enabling the CD bit and disabling the NW bit in the control register CR0 ([63], Section 11.5.3). Some architectures allow disabling caching for a specific address range, primarily for serving uncacheable DMA requests or memory-mapped I/Os. For instance, on x86, users can use the Memory Type Range Register (MTRR) to change the cacheability of a physical memory range. Newer Intel processors also support page attribute table (PAT) to manage page cacheability with the attribute field in page table entries.

However, besides disabling the entire cacheability, neither MTRR or PAT can overwrite the cacheability of SGX's processor-reserved memory (PRM) [39]. The cacheability of PRM is specifically controlled by a special register called Processor-Reserved Memory Range Register

(PRMRR), which can be only written by BIOS during booting. Since there is no proprietary BIOS that allows the user to modify PRMRR, the attacker effectively has no way to change the cacheability of the encrypted memory. However, since the BIOS is untrusted in the threat model of SGX, in theory, one can reverse-engineer the existing BIOS or build a custom BIOS to overwrite PRMRR. We do not choose this route because disabling cacheability will incur significant slowdown, making the attack easy to detect by the victim.

5.2 Critical Page Whitelisting

We observed that after paging (swapping), memory access in the swapped pages becomes unobservable to the attacker. Such a phenomenon is common for SGX since SGX has to rely on the OS to swap pages in and out of the EPC. Both swap-in and swap-out causes the page to be loaded into the cache hierarchy (LLC, L2, and L1-D caches), because the SGX instructions for swap-in and swap-out, i.e., `eldu` and `ewb`, require re-encrypting the page from/to a regular physical page [39]. After the instructions, the cache lines stay in the cache hierarchy until being evicted by other memory access. Currently, an Intel CPU with SGX only has up to 93.5MB in the EPC, making paging the primary obstacle to observing critical transactions on the memory bus.

On the other hand, paging also complicates the virtual-to-physical address translation, as the mappings can change midst execution. We observe certain patterns in the memory bus log to identify the paging events. However, these patterns can also become unobservable if the page is recently swapped and most of the cache lines are still in the LLC.

Therefore, to eliminate the side effect of paging, we pin the EPC pages for the critical address range, by modifying the SGX driver. We start by identifying the critical address range of each target program. Take the Hunspell program for example. The critical memory transactions come from accessing the dictionary nodes, which are allocated through `malloc()`. For simplicity, we disable Address Space Layout Randomization (ASLR) inside the enclave (controlled by the library OS [54]), although we confirmed that ASLR can be defeated by identifying contiguous memory access pattern in the traces. Next, we calculate the number of EPC pages needed for pinning the critical pages. For a Hunspell execution using an en_US dictionary, the total `malloc()` range is 5,604 KB. Finally, we need to give the critical address range as an input to the modified SGX driver. When the driver allocates an EPC page, it checks if the virtual address is in the critical address range and use an in-kernel flag to indicate if the page has to be pinned. The driver will never swap out a pinned page.

5.3 Priming the Cache

We explore ways to actively contaminate the caches by accessing contentious addresses. This technique is called *cache priming*, which is used in the PRIME+PROBE attack [44]. Previous work has established priming techniques for either same-core or cross-core scenarios. Some priming techniques are restricted by CPU models, especially since many recent CPU models have employed designs or features that raise the bar for cache-based side-channel attacks. However, recent studies also show that, even with these defenses, attackers continue to find attack surfaces within the CPU micro-architectures, such as priming the cache directory in a non-inclusive cache [64].

We focus on cross-core priming since same-core priming requires interrupting the enclaves using AEX or page faults. The usage of cache priming in MEMBUSTER is distinctly different from existing cache-based side-channel attacks since MEMBUSTER does not require resetting the state of the cache or synchronizing with the victim. The goal of cache priming in MEMBUSTER is to simply evict the critical addresses from the cache to increase the cache misses. Also, with cache squeezing, we only have to prime the cache sets dedicated to the critical addresses. These differences make it easy to apply multiple priming attacks simultaneously, as long as they all eventually contribute to increasing cache misses.

Cross-Core Cache Priming We run multiple priming processes on other cores to evict the critical cache lines from the LLC. These processes will repeatedly access the cache sets that are shared with the critical addresses of the victim. The attacker will start by identifying the critical addresses and the cache sets to prime. Then, the attacker starts the priming processes before the victim enclave, to actively evict the cache lines during execution. Take the Hunspell attack for example. Since its critical addresses are spread over all cache sets, the attacker needs to repeatedly prime all cache sets. No synchronization is required between the attack processes and the victim. We do not prime the L1 and L2 caches across cores, but cross-core priming on private caches is demonstrated on Intel CPUs [64].

A potential hurdle for cross-core priming is to obtain sufficient memory bandwidth to evict the critical cache lines. Based on our experiments, a priming process that sequentially accesses the LLC has around 100–200MB/s memory bandwidth. Priming a 9MB LLC with 2,048 sets requires about 100 milliseconds, which is too slow to evict the critical cache lines before the lines are accessed by the victim again. For instance, Hunspell accesses a word every 2 thousand DRAM cycles ($<$ 1 microseconds), and Memcached accesses a word every 5 million DRAM cycles ($<$ 2.5 milliseconds). We will discuss, however, how an attacker can evict all the critical cache lines within a few milliseconds by pinpointing the priming process to target only 64–128 sets (See §5.4.2).

Page-Fault Cache Priming Potentially, an attacker can prime the LLC, L2, and L2-D caches on the same core with the victim, by interrupting the victim periodically. To do so, the attacker can take a similar approach to the Controlled-Channel Attack: The attacker identifies two code pages containing code around the critical memory accesses, and then alternatively protects the pages to trigger page faults. To increase cache misses, the attacker needs not to prime the cache

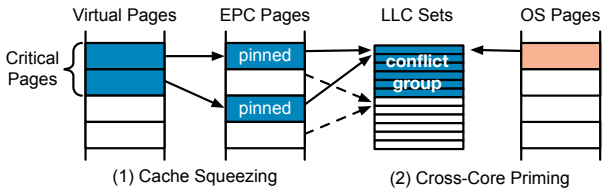

Figure 7: Techniques used to increase the cache miss rate with minimal performance overhead.

at every page fault, but rather can prime at a low frequency. However, such a page-fault priming technique still causes a lot of interference and overhead to the victim, making it easy to detect [22] or to mitigate [31,32]. For example, priming the cache on every 10-20 page faults incurs about 3× overhead to the victim. In addition, known countermeasures, such as T-SGX [31], can effectively prevent page faults using transactional instructions. Therefore, we do not use this technique.

5.4 Shrinking the Effective Cache Size

As previously discussed, cache priming alone cannot create sufficient memory access bandwidth for evicting the critical cache lines in time. Therefore, we introduce a novel technique called *cache squeezing*, which shrinks the effective cache size to incur more cache misses for a specific address range. We show that the technique can be combined with non-intrusive techniques like cross-core cache priming to make MEMBUSTER a more powerful side channel.

5.4.1 Cache Squeezing

As the name suggests, cache squeezing can shrink the effective cache size for a given set of critical pages. By squeezing the cache that an enclave can use, the attacker can incur both *conflict misses* and *capacity misses* on LLC, therefore becoming able to observe more cache misses on the bus.

In modern processors, the L2 cache and LLC are physically-indexed. The lowest 6 bits of the physical address are omitted, given that each cache line is 64 bytes. The next s lower bits are taken as the *set index*. Each set then consists of W ways to store multiple cache lines of the same set index. For an enclave, an OS-level attacker can control the physical pages that are mapped to the enclave's virtual pages. This allows the attacker to manipulate the physical frame number (PFN) of each virtual address of the enclave, and subsequently, the higher $s-(12-6)=s-6$ bits of the set index.

Figure 7(1) shows how cache squeezing works in combination with page pinning. The attacker first defines the critical addresses of the victim, then maps these pages to EPC pages that share the minimum amount of cache sets. This technique requires cache pinning so that these pages will never be swapped out from the EPC. Since the OS only controls the higher $s-6$ bits of the set indices, the smallest group of physical pages that will evict each other share exactly $2^6=64$ sets. We called such a group of physical pages a *conflict group*. Since the maximum size of EPC is 93.5 MB, the entire cache can be partitioned to 2^{s-6} conflict groups where each conflict group can accommodate 93.5 MB/4 KB/2^{s-6} EPC pages. In our experiment, $s=11$ (2048 sets) and $W=12$, so each conflict group can accommodate at most 748 pages (2,992 KB). The critical address range of Hunspell, for example, is the whole `malloc()` space, which is 5,604 KB and thus requires two conflict groups. Finally, the attacker gives the critical address range to a modified SGX driver, which will only map physical pages from the selected conflict groups to any critical virtual address.

Using cache squeezing to increase cache misses has many benefits. First of all, it does not require interrupting the victim enclaves, nor does it need to incur more memory accesses in the background. All memory accesses used to push cache lines out of the L2 cache and LLC are legitimate accesses from the victim enclaves. Therefore, cache partitioning cannot defeat cache squeezing because there is no cross-context cache sharing. In fact, way-partitioning features such as Intel CAT [65] can be exploited to further shrink the effective cache sizes in combination with cache squeezing.

5.4.2 Cross-Core Priming with Cache Squeezing

As we mentioned in § 5.3, cross-core cache priming may not have sufficient bandwidth to evict the critical cache lines in time. However, we found that cache squeezing makes the priming more effective by shrinking the effective cache size. Instead of priming all the cache sets, the attacker now only has to prime the sets of the targeted conflict groups containing the critical addresses (Figure 7(2)). Each group of 64 cache sets contains $W\times 4$KB, allowing the priming process to evict the part of cache within a millisecond. The priming process can run in parallel and does not affect the victim execution except causing cache contention.

5.4.3 Limitation

Although cache squeezing can increase the cache misses among critical addresses, it could be less effective if the victim has only a few critical addresses or a small memory footprint. If the critical addresses can only fill a small part of a conflict group ($W\times 4$ KB), the victim enclave may not be able to cause enough cache misses to benefit the attacker. For example, Memcached only has 2 MB (500 pages) of the critical address range. To fill all of the 748 pages, we identify the top 248 frequently-accessed pages (in addition to the critical addresses) through simulation, and assign these extra pages to the same conflict group.

Note that the LLC of a modern CPU usually has a *cache slice* feature that distributes the addresses across multiple cache banks using an undocumented, model-specific mapping function. Reverse-engineering the slicing function of the target CPU is useful for further reducing the effective cache space for an enclave if the enclave has a smaller memory footprint. Reverse-engineering of slicing functions is already explored by prior papers [64], so we will not discuss this technique in this paper.

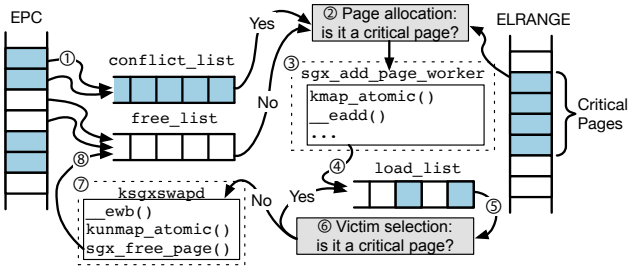

Figure 8: Implementation of critical page whitelisting and cache squeezing in a modified SGX driver. To ensure no swapping in the sensitive memory range, EPC pages are set aside in a separate queue. The attackers can further select the EPC pages based on set indexes or other logistics.

One can detect the cache squeezing by testing if critical addresses are mapped in an adversarial way. Since the enclave is not aware of physical address mappings by itself, it needs to experimentally detect such mapping by accessing the addresses and measure latency. However, we claim that it is challenging because (1) the victim needs to know the critical address range to detect the mapping, and (2) the enclave cannot tell if the mapping was accidental or intentional.

5.4.4 Implementation

We use a modified SGX driver to implement both critical page whitelisting and cache squeezing as shown in Figure 8. The driver accepts parameters to specify a sensitive range within the victim application, and calculates how many conflict groups are required for the attack. ① When the driver initializes, it inserts conflicting EPC pages to a separate queue (i.e., `conflict_list`). ② When adding enclave pages, the driver checks if the virtual page number is in the critical address range. ③ The driver maps the critical pages to pages popped from `conflict_list`. ④ All of the mapped pages are added to the list of loaded pages (`load_list`). ⑤ When the driver needs to evict an EPC page, it searches the victim from the list of loaded pages. ⑥ If the selected page is a critical page, it searches again. ⑦ Only non-critical pages are evicted and the enclave continues to run. Other enclaves are not affected by the modification and can function normal with marginal overheads.

Our change to the SGX driver contains only 290 lines. The SGX driver uses the `fault` operation in vm_operations_struct to handle EPC paging. We use a customized `fault` function, which checks the faulting virtual addresses of the enclave and then applies different paging strategies to critical and non-critical addresses. We hard-code the range of critical addresses for each application and thus require switching the drivers for a different target. Potentially, the driver can export an API to the attackers for specifying the critical addresses. Our driver also only supports one single victim enclave at a time. However, we can extend the driver to target multiple enclaves simultaneously as long as the total memory usage can fit into the EPC (required for pinning).

6 Extracting Sensitive Access Patterns

OS techniques including critical page whitelisting, cache squeezing, and cross-core priming effectively increase the cache misses on the cache misses on critical addresses. However, the traces collected from the memory bus are still full of noise and contain no marker for splitting the critical memory accesses into iterations. Unlike controlled-channel attacks, MEMBUSTER cannot rely on repeated code addresses (e.g., from a loop) to mark and then split the critical accesses because these code addresses tend to be accessed too frequently to be evicted by our techniques. Therefore, the attacker needs to deeply analyze the memory traces offline to distill the sensitive information.

To extract the sensitive access patterns, we identify four techniques for filtering the critical memory addresses and matching with a known oracle for the target application: (1) offline simulation; (2) searching the beginning of sensitive accesses; (3) fuzzy pattern matching, and (4) exploiting cache prefetching. We use the two examples to explain how to analyze memory bus traces.

6.1 Offline Simulation

Side-channel attacks often require attackers to have some knowledge about the behaviors of the victim. For example, the controlled-channel attack on Hunspell requires the attacker to extract the virtual page addresses of the linked list nodes of each dictionary word, during an online *training phase* while attacking the victim. However, MEMBUSTER cannot perform online training with the victim as the analysis of the memory traces is performed offline. Instead, the attacker needs to generate an oracle of the victim behavior, using offline simulation of the target application.

We observe that, for each application, we can use a deterministic oracle, given that users have adopted some publicly available data (e.g., the `en_US` dictionary). For example, during the simulation, we run a modified Hunspell in an enclave, which prints out the indexes and the addresses of linked list nodes visited for each word. Then, we reuse the output as an oracle, to be used in analyzing any traces based on the same en̂_US dictionary. We assume that there are only a finite amount of English dictionaries in the world.

As discussed earlier, ASLR in the enclaves does not invalidate an oracle, since ASLR can be easily defeated by observing the specific patterns related to binary loading. The addresses in the oracle can simply be shifted by a certain offset to be usable again.

6.2 Searching Sensitive Accesses

Finding the first sensitive access is critical for deciding where to start matching access patterns. Note that not all accesses to the critical addresses are sensitive. For Hunspell, allocating nodes for each word emits a long sequence of monotonically increasing virtual addresses that can be used to identify the sensitive addresses. We match the virtual addresses to the

oracle, to find the *longest increasing subsequence (LIS)* of addresses as accessed in the dictionary order. After finding the LIS, the next critical access is the beginning of the sensitive addresses.

6.3 Fuzzy Pattern Matching

In MEMBUSTER, we observe that a part of memory addresses in a sensitive access pattern is likely to be missing due to caching. Even with cache squeezing and cross-core priming, it is almost impossible to force page misses on every critical memory access. Therefore, to analyze lossy traces, we use fuzzy pattern matching to flexibly match the traces with only parts of access patterns. As long as at least one or a few accesses of a pattern cause LLC misses, we can identify the pattern as a possible result for recovery.

In fuzzy pattern matching, one address may be parsed as different access patterns of the victim for two reasons. First, within a data structure such as a linked list or a tree, the same address (an inner node) may be accessed while traversing or searching other nodes. Second, a cache line may contain multiple nodes and thus can be accessed when visiting one of the nodes. For either of the reasons, a single memory trace may be accounted for multiple possible access patterns in the oracle.

We use a simple strategy to select the best interpretation for a set of memory traces. We assign a score to each possibility based on how *complete* the traces have matched with an access pattern in the oracle. For the addresses of a tree or a linked list, we assign lower scores to the root and the first few nodes and assign higher scores to nodes that are closer to leaves or the end of the list. By collecting the top-ranking interpretations of the memory traces, an attacker can generate a list of the most probable options of the target secret. Potentially, a grammar checker or any semantic-based heuristic can help to validate or to rank the recovery results. We leave the exercise of applying more context-aware heuristics for future work.

6.4 Exploiting Cache Prefetching

Finally, we observe that the cache prefetching features of CPUs can help increase the accuracy of the attack. For example, a recent Intel CPU includes *Next-line Prefetcher* and *128-byte Spatial Prefetcher*. The Next-line Prefetcher, belonging to the L2 cache, will preload the cache line next to the one that is currently accessed. The 128-bit Spatial Prefetcher, which also belongs to the L2 cache, prefetches the pairing cache line that completes the accessed cache line to a 128-byte aligned chunk into the LLC. Both prefetchers increase the number of memory accesses relevant to the secret data. Therefore, we expand the range of pattern matching based on our knowledge of cache prefetching, including extending the addresses representing each secret by 64 bytes, both backward and forward. As a result, even if the CPU has cached a line, the prefetched lines may still cause cache misses and be observed on the memory bus.

CPU	
Model	Intel i5-8400 (Coffee lake)
LLC Size	9 MB
LLC # Slice	6 Slices
LLC # Associativity	12-way set associative
LLC # Sets	2048
Memory	
DIMM Type	DDR4-2400 UDIMM (Non-ECC)
Capacity	8 GB
Channel/Rank/Bank/Row	1/1/16/65536
Page Size	8 KB (1 KB/package)
Max Bus Frequency	1200 MHz

Table 2: Hardware specification for the experiment

Other cache prefetchers such as *Stream Prefetcher* can monitor an ascending or descending sequence of addresses from the L1 or L2 cache and can prefetch up to 20 cache lines ahead of the loaded address. Such a prefetcher generally will not improve the accuracy of the pattern matching. However, these prefetchers can cause space pressure to caches, making cache squeezing more effective.

7 Evaluation

In this section we present the evaluation results of the MEMBUSTER attack, based on the two vulnerable applications described in §4. The evaluation mainly answers the following questions regarding the MEMBUSTER attacks:

- How accurate can MEMBUSTER extract the secrets from applications that are vulnerable to such an attack?
- How do the attack techniques of MEMBUSTER impact the attack accuracy?
- How much slowdown (or interference) the various techniques will incur on the applications?
- What is the limitation of MEMBUSTER?
- How sensitive are the attack results of MEMBUSTER to the last-level cache (LLC) size of the target CPU?

We evaluate the MEMBUSTER attack in various settings: (1) the basic attack without any techniques (**None**); (2) the optimized attack with cache squeezing (**SQ**); (3) the optimized attack with cache squeezing combined with cross-core cache priming (**SQ+PR**).

7.1 Experiment Setup

In this section, we describe the experimental setup of the MEMBUSTER attack. We use both physical and simulated experiments to evaluate the effectiveness of MEMBUSTER.

7.1.1 Physical Experiment

Hardware Setup. The hardware setup we used for the experiment is shown in Table 2. We use a machine equipped with an Intel SGX CPU. In the machine, we connect the DIMM to a signal analyzer via a DIMM interposer. We configure BIOS to slightly increase the DRAM supply voltage to offset the voltage drop caused by the interposer. The bus frequency is set to 1066 MHz, so the bandwidth of the analyzer is 3.97 GiB/s. With a 64 GiB acquisition depth, we can log the memory bus for up to $\sim$ 16 seconds. All of our experiments have

finished in a few seconds, and thus the acquisition depth is sufficient for logging all the memory requests. To achieve a wider time window, the attacker can choose an analyzer which can filter the requests by addresses [57], or which has a higher acquisition depth [55].

Victim Setup. The victim machine is running Ubuntu 16.04 and Linux kernel 4.4. To execute the victim applications inside enclaves, we use Graphene-SGX [54] to run unmodified binaries with SGX. The victim may also choose other frameworks [66] or port the applications with the SDK [67], but the choices of the frameworks do not eliminate the patterns since they do not change the program logic of the victim.

Sample Size. We collaborate with SK Hynix to use its proprietary analyzer for the experiments. Due to the limited access to the device, we run the attack only *once* for each setting. However, we were able to successfully perform the attack despite the small sample size because the results match well with our expectations learned from the simulation.

7.1.2 Microarchitectural Simulation

We also implemented a software simulator to simulate the attack prior to an actual attack because the hardware setup requires costly devices. We use the simulator for exploring the attack and getting preliminary results. The results are then cross-validated with the results from the actual hardware setup, to verify the functional correctness of the simulation. The attacker can also use the same strategy to save the expenses for renting the devices. We modify QEMU [38], a machine emulator, to trace all the physical memory accesses of the guest. To capture cache misses, we make QEMU emits all the memory requests to a cache simulator we integrated from Spike [68]. The cache simulation does not implement any cycle-accurate hardware model as well as cache slicing and pseudo-LRU replacement. However, the simulation was sufficiently faithful for developing the attack scripts to analyze the real memory traces.

7.1.3 Enclave Simulation

We also simulate an enclave environment without memory encryption, using a modified Graphene-SGX library OS and a dummy SGX driver. We consider simulating Intel's Memory Encryption Engine (MEE) unnecessary because MEE does not affect the memory addresses accessed within the EPC. MEE generates additional access patterns for the integrity tree or EPC metadata, both of which are stored in the Processor Reserved Memory outside the EPC. Our attack does not rely on any access pattern outside the EPC.

The modified Graphene-SGX library OS and the dummy SGX driver primarily simulate the transition in and out of the enclave and the paging of enclave memory, to generate similar memory access patterns as observed on the memory bus. For simulating enclave entry and exit, we modify the user-tier SGX instructions, `EENTER` and `EEXIT`, in the Graphene-SGX runtime, to directly jump to addresses that are originally given as the enclave entry. We also simulate the AEX.

Technique	Attack Accuracy	Normalized Exec. Time
None	34.1%	1.00×
SQ	82.1%	0.92×

Table 3: MEMBUSTER results for attacking Memcached on an SGX machine

For simulating EPC paging, we modified the SGX driver to replace the system-tier SGX instructions, including the `ELDU` and `EWB` instructions, which swap and re-encrypt pages in and out of the EPC. We simply replace these two instructions with memory copy without encryption. We compare the memory traces from the real enclaves and from the simulation to confirm that the results are identical.

7.1.4 Applications: Hunspell

We run Hunspell v1.6.2 to evaluate the effectiveness of the MEMBUSTER attack. We use a standard `en_US` dictionary [69] with two document samples: a random non-repetitive document with 10,000 words (**Random**), and a natural-language document "Wizard of Oz" with 39,342 words (**Wizard**). For simplicity, we normalize the samples based on `en_US` dictionary, by converting non-existing words in the samples to the closet words in the dictionary. MEMBUSTER does not recover words that are reported as misspelt by Hunspell. In addition, we disabled affix detection in Hunspell.

We use the pattern matching algorithm described in §6 to recover the target document from the DRAM traces collected from the Hunspell program running inside the enclave. We also enable the hardware prefetching by configuring the BIOS. To verify the result, we select an interpretation of the DRAM traces that is closet to the target document, from a set of highest-ranking results generated from our algorithm.

7.1.5 Application: Memcached

We run Memcached v1.5.12 as another target of the MEMBUSTER attack. In this attack, the "secrets" are the data being looked up in the Memcached cache. We used the Enron email dataset [70] as a realistic workload for Memcached. First, we compute the 4-byte hash of each word that appears in emails in the "sent mail" directory of each user. In total, there are about 7000 unique word entries in the dataset, which include articles and propositions. During the *training phase*, assuming the attacker is monitoring a Memcached server, the attacker can determine both the hash table address and the hash value of each word using the traces of a few queries. Then, during the *attack phase*, the attacker monitors the memory bus traffic of an enclave-protected Memcached server receiving caching requests from an trusted email server. The email server parses emails from a test data set that contains randomly selected emails with around 1000 words in total. As the Memcached server processes the caching requests from the email server, the attacker can extract the words in the emails using the MEMBUSTER attack.

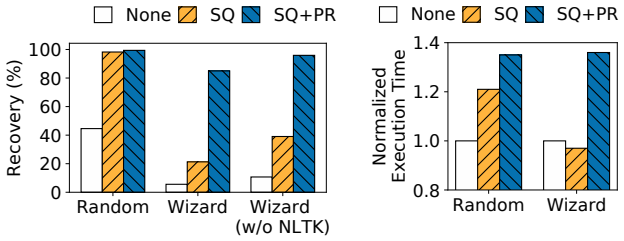

Figure 9: Hunspell document recovery rate (left) and normalized execution time (right) on two documents: Random document (Random) and Wizard of Oz (Wizard). The comparison is between without any techniques (None); with cache squeezing (SQ); and with cache squeezing and cross-core priming (SQ+PR). For Wizard of Oz, we also show the recovery rate of uncommon words only (w/o NLTK).

7.2 Effectiveness of the Attack

7.2.1 Data Recovery Accuracy

Figure 9 (left) and Table 3 show the accuracy of MEMBUSTER for recovering the victim's data. We measure the accuracy based on the number of words recovered from the collected traces, compared to the number of words in the original samples. The recovery rate is higher in a non-repetitive (Random) or high-interval access pattern (Memcached) than in a repetitive access pattern (Wizard). Even without any techniques (None), Memcached and Random show 34% and 44% recovery rates, respectively. With cache squeezing, we recover 96% of the random document and 82% of the Memcached query.

However, for Wizard of Oz, None or SQ can only achieve up to 21% recovery rate. The main reason is that the document contains many repetitive words, including common words such as "you" and "the" and uncommon words such as "Oz" and "scarecrow". The memory accesses for these words are likely to be cached in the LLC cache without emitting any DRAM requests. On average, each unique word in Wizard of Oz repeats 15.5 times. We found that without cache squeezing and cross-core priming, the attack recovers about 0.3 occurrences of each word on average. Even with cache squeezing, the attack only recovers about 2.6 occurrences.

Since cache squeezing shrinks the effective cache size for the critical addresses, cross-core priming becomes more efficient by only priming the sets of the critical addresses. We show that combining cache squeezing and cross-core priming (SQ+PR) achieves 85% recovery accuracy on Wizard of Oz.

Furthermore, the attacker is most likely to need only the *uncommon* words to be recovered. To exclude common words, we use *stopword*s from the NLTK dataset [71] which includes 179 common words (e.g., "the"). Excluding these words, MEMBUSTER can recover Wizard of Oz up to 95% (Figure 9 Wizard w/o NLTK).

7.2.2 Overhead and Interference

We show that MEMBUSTER does not incur an orders-of-magnitude overhead that can be distinguishable by the victim.

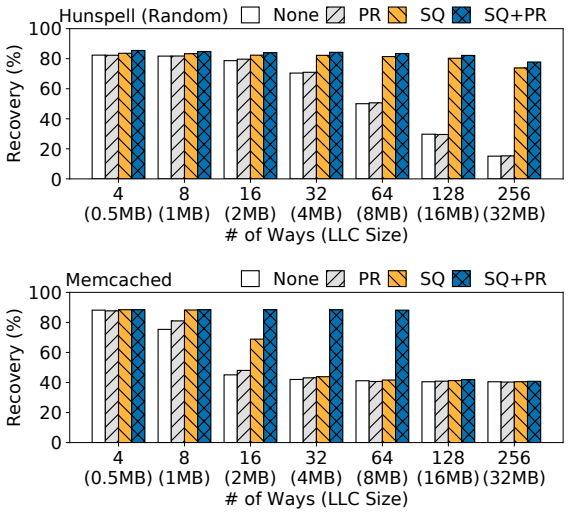

Figure 10: Simulation results of the attack on Hunspell (top) and Memcached (bottom).

Figure 9 (right) shows the normalized execution time with different attack techniques with respect to the baseline. In general, both cache squeezing and cross-core priming have a low performance impact on the victim program, since these techniques do not interrupt the victim program. For Hunspell, cache squeezing causes up to 21% overhead to the victim, and up to 36% if combined with cross-core priming. The overheads are mainly caused by the increase of cache misses inside the victim program.

Table 3 also shows the end-to-end execution time of Memcached for processing the whole test set. Similar to Hunspell, the basic attack incurs no overhead on Memcached. Interestingly, cache squeezing reduces the execution time by 8% for Memcached. We observe that, on a physical machine, critical page whitelisting consistently reduces the average LLC miss rate (2.9% vs. 3.6%) as well as the page fault rate. Because the physical pages of Memcached's hash table are pinned inside the enclave, and thus never get swapped out from the EPC. Thereby, within the hash table, there is no expensive paging and context switching cost that generally plagues enclave execution.

7.2.3 Scalability on # of Ways

We simulated the attack on our simulation environment to show the scalability of MEMBUSTER. We fixed the number of sets $s = 2048$ that most Intel CPUs choose to have. Since we did not simulate the LLC slices, we increased the size of the cache by increasing the number of ways, W. To clarify, increasing the number of ways does not reflect the actual behavior of LLC with multiple slices. Even if the LLC has multiple slices, each cache line will compete with W other cache lines. Thus, increasing W makes the attack much harder, by reducing the chance of eviction of critical addresses. Note that a typical W value is between 4 and 16.

As shown in Figure 10, cache squeezing makes cross-core

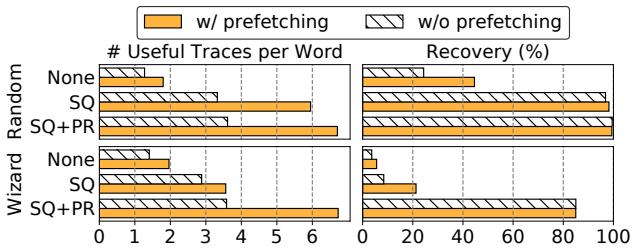

Figure 11: The number of useful traces per word and the document recovery rate for each experiment. We compare the cases with or without the hardware prefetcher.

priming much more effective in general by reducing the effective cache size. Cache squeezing was more scalable on Hunspell than Memcached, because Hunspell has a larger critical address range. With $W = 64$, MEMBUSTER recovered up to 83% of the random document in Hunspell and 88% of the emails in Memcached when both cache squeezing and cross-core priming have been used. Even assuming an unrealistic number of ways $W = 256$, which results in 32 MB of LLC, the attack accuracy was 77% and 40% respectively.

7.3 Per-Application Detailed Analysis

7.3.1 Hunspell: Advantage of Cache Prefetching

We also show the advantage of exploiting cache prefetching for MEMBUSTER. For Hunspell, the attacker recovers each word based on multiple memory accesses. If the attacker observes more traces relevant to each word, recovering the word becomes easier. Hence, if the attacker knows the presence of cache prefetchers in advance, she can use the information to correlate the prefetched addresses with each word (§6).

As shown in Figure 11, cache prefetching increases the average number of useful traces per word. Including prefetched addresses increases the recovery rate especially when there are very few useful traces (None and SQ). Although the improvement is marginal in our experiment, the attacker can potentially use the additional memory requests made by the cache prefetchers to extract more information from the victim.

7.3.2 Memcached: Advantage of Fine-Grained Addresses

To show the advantage of observing fine-grained addresses, we simulated the controlled-channel attack on Memcached example. We first obtained the entire memory trace from Memcached without simulating the cache. We then masked the lower 12-bits of all addresses assuming each page is 4 KB. With this post-processing, we were able to simulate the memory trace that the controlled-channel attacker will observe. We also reconstruct the attacker's hash table such that each page-granularity address maps to multiple entries in the hash table. If the attacker sees an address, she simply chooses the most common word among the possible entries.

The simulated controlled-channel attack achieves only 29% accuracy, and the recovered document was uninterpretable as it only contained common words such as "the" and "of". This shows that MEMBUSTER leverages fine-grained addresses by providing more side-channel information than coarse-grained addresses.

8 Discussion

In this section, we discuss the limitations, generalization, implications, and mitigations of the MEMBUSTER attack.

Limitations. MEMBUSTER leaks only memory access patterns at LLC misses. Thus, MEMBUSTER cannot observe repeated accesses to the same address within a short period. For instance, the former RSA implementation of GnuPG [72] is known to leak a private key through code addresses in the ElGamal algorithm [45]. This type of attack relies on *data-dependent branches*, as the attacker detects different code paths executed inside the victim to infer the secret. However, these vulnerabilities are difficult to exploit by MEMBUSTER, due to these code addresses being frequently executed and thus cached in the CPU. Even cache priming techniques cannot efficiently evict the code addresses in time to help the attacker retrieve the secret with high accuracy but keep the performance impact low.

In general, MEMBUSTER is more suitable for leaking *data-dependent memory loads* over a large heap or array. For instance, both the attacks on Hunspell and Memcached rely on the access patterns within a large hash table and/or linked-list objects. If the victim program only has data-dependent memory access patterns within a small region, or if the memory access is not evenly distributed, the accuracy of MEMBUSTER is likely to worsen. Besides, if the application only leaks a secret through *stores* that are dependent on the secret, MEMBUSTER may not observe the memory requests immediately. The reason is that the CPU tends to delay *write-back* of dirty data until the cache lines are evicted, making the timing of the memory requests appearing on the memory bus unpredictable. We leave the exploration of such scenario for future work.

Timing Information. Although not explored in this paper, an attacker may exploit the timing information to attack the victim. The DRAM analyzer logs a precise timestamp for each memory request based on counting its clock cycles. Potentially, an attacker can measure the time difference between two memory traces, to infer the execution time of operation in the victim as a way of timing attacks. We leave the demonstration of these attacks for future work.

Traffic Analysis. Potentially, the memory bus traffic recorded by the DRAM analyzer can be used for traffic analysis if the victim is vulnerable to this type of attacks. For instance, the attacker may analyze either the density or the volume of requests on a specific address to infer the activity or secret of the application. A complete mitigation of the attack should eliminate the timing information and has a constant traffic flow on the memory bus [36].

Multiple DIMMs or Multi-Socket. Our current attack does not explore the possibility of having multiple DIMMs or multiple CPU sockets (currently not supported by SGX). However, potentially, the attacker can attach multiple DIMM interposers, and then correlate the DRAM traces using timestamps or common patterns.

Memory Controllers. A memory controller arbitrates all transactions to main memory such that it maximizes the throughput while minimizing latencies. One of the key features that may make MEMBUSTER more challenging is *transaction scheduling* where the *arbiter* reorders the transaction requests to maximize the performance. In other words, the order of the memory transactions observed by the attacker may differ from the actual order of memory accesses.

We observe that the arbitration of the memory controller does not stop an enclave from leaking sensitive access patterns. First, even if transactions are reordered, the critical addresses will still eventually appear on the memory bus. Also, the memory controller only reorders transactions within a very small time window (e.g., tens of bus cycles), which is not enough to obfuscate the critical memory accesses that occur at least every hundreds of instructions.

Generalization. Intel SGX is not the only platform affected by MEMBUSTER. Other existing platforms of hardware enclaves [4, 5, 40, 41] also do not encrypt the addresses on the memory bus. Thus, these platforms are also vulnerable to MEMBUSTER as long as the CPU stores encrypted data in external memory (e.g., DRAM). The attacker can also use the same techniques such as cache squeezing to induce cache misses on other platforms. For example, Komodo [40] allows the OS to affect the virtual address mapping, which enables the attacker to use cache squeezing. Keystone [4] measures the initial virtual address mapping for attestation, thus cache squeezing cannot be applied. However, it provides cache partitioning which can reduce the effective cache size of the enclave.

Implications and Disclosure. Potentially, MEMBUSTER can be used in two scenarios: (1) a malicious user attacking an end device to retrieve secret data from a local enclave; (2) a malicious cloud provider or employee attacking a cloud machine to retrieve secret data from the tenants. The existence of MEMBUSTER shows the importance of physical security to enclaves just on par with software security. Ideally, in a secure cloud, one may want to separate the person who has physical access to the machine from the person who has administrative privileges. This may be achieved by a secure boot system that prevents people who have physical access from overwriting system privileges.

We have disclosed the details of this attack to Intel, who has acknowledged its validity.

Mitigations. There are several ways to mitigate MEMBUSTER, but they are generally expensive. Oblivious RAM (ORAM) [34, 73] can make the applications execute in an oblivious manner so that the attacker cannot infer secret data based on the memory access pattern. The high performance overhead of ORAM makes it less attractive for applications that have strong performance requirements. Alternatively, we can also encrypt the address bus as proposed by InvisiMem [36] and ObfusMem [37]. However, adding such a feature to commodity DRAM would be very expensive; take the cost of techniques such as Hybrid Memory Cube (HMC) [74] for an example. In-package memory such as high bandwidth memory (HBM) may relieve the needs for protection against untrusted DRAM [75], but remains an expensive alternative for production.

9 Conclusion

In this paper, we introduced MEMBUSTER, which is a non-interference, fine-grained, stealthy physical side-channel attack on hardware enclaves based on snooping the address lines of the memory bus off-chip. The key idea is to exploit OS privileges to induce cache misses with minimal performance overhead. We also demystify the physical bus-based side channel by reverse-engineering the internals of several hardware components. We then develop an algorithm that can retrieve application secrets from memory bus traces. We demonstrated the attack on an actual SGX machine; the attack achieved similar accuracy with much lower overhead than previous attacks such as controlled-channel attacks. We believe the attack technique is prevalent beyond Intel SGX and can apply to other secure processors or enclave platforms, which do not protect memory buses.

Acknowledgments

We thank our shepherd, Daniel Genkin, and the anonymous reviewers for their insightful comments. We thank Krste Asanović and Martin Maas for sharing their ideas. Jeongseok Son from UC Berkeley also contributed to the early stage of the project. We also thank SK Hynix, especially Dongha Jung, Taeksang Song, and Yongtak Song for providing the facility for DRAM signal analysis, collecting physical experiment data, and explaining the technical details of DRAM. This work was supported in part by NSF grants CNS-1228839, CNS-1405641, CNS-1700512, NSF CISE Expeditions Award CCF-1730628, as well as gifts from the Sloan Foundation, Alibaba, Amazon Web Services, Ant Financial, ARM, Capital One, Ericsson, Facebook, Google, Intel, Microsoft, Scotiabank, Splunk, and VMware.

References

[1] Intel Software Guard Extensions. https://software.intel.com/sgx. Last accessed: December 2, 2019.

[2] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V. Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R. Savagaonkar. Innovative Instructions and Software Model for Isolated Execution. In *HASP*, 2013.

[3] David Lie, Chandramohan A Thekkath, and Mark Horowitz. Implementing an Untrusted Operating System on Trusted Hardware. *ACM SIGOPS Operating Systems Review*, 37(5):178–192, 2003.

[4] Dayeol Lee, David Kohlbrenner, Shweta Shinde, Dawn Song, and Krste Asanović. Keystone: A framework for architecting tees. *arXiv preprint arXiv:1907.10119*, 2019.

[5] Victor Costan, Ilia A Lebedev, and Srinivas Devadas. Sanctum: Minimal Hardware Extensions for Strong Software Isolation. In *USENIX Security*, 2016.

[6] J Alex Halderman, Seth D Schoen, Nadia Heninger, William Clarkson, William Paul, Joseph A Calandrino, Ariel J Feldman, Jacob Appelbaum, and Edward W Felten. Lest we remember: cold-boot attacks on encryption keys. *Communications of the ACM*, 2009.

[7] Christian Priebe, Kapil Vaswani, and Manuel Costa. EnclaveDB - A Secure Database using SGX. In *IEEE S&P*, 2018.

[8] Felix Schuster, Manuel Costa, Cedric Fournet, Christos Gkantsidis, Marcus Peinado, Gloria Mainar-Ruiz, and Mark Russinovich. VC3: Trustworthy Data Analytics in the Cloud. In *IEEE S&P*, 2015.

[9] Tien Tuan Anh Dinh, Prateek Saxena, Ee-Chien Chang, Beng Chin Ooi, and Chunwang Zhang. M2R: Enabling Stronger Privacy in MapReduce Computation. In *USENIX Security*, 2015.

[10] Stefan Brenner, Colin Wulf, David Goltzsche, Nico Weichbrodt, Matthias Lorenz, Christof Fetzer, Peter Pietzuch, and Rüdiger Kapitza. SecureKeeper: Confidential ZooKeeper Using Intel SGX. In *Middleware*, 2016.

[11] Joshua Lind, Oded Naor, Ittay Eyal, Florian Kelbert, Emin Gün Sirer, and Peter Pietzuch. Teechain: A Secure Payment Network with Asynchronous Blockchain Access. In *SOSP*, 2019.

[12] Mitar Milutinovic, Warren He, Howard Wu, and Maxinder Kanwal. Proof of Luck: An Efficient Blockchain Consensus Protocol. In *SysTEX*, 2016.

[13] Fan Zhang, Ethan Cecchetti, Kyle Croman, Ari Juels, and Elaine Shi. Town crier: An authenticated data feed for smart contracts. In *CCS*, 2016.

[14] R. Cheng, F. Zhang, J. Kos, W. He, N. Hynes, N. Johnson, A. Juels, A. Miller, and D. Song. Ekiden: A Platform for Confidentiality-Preserving, Trustworthy, and Performant Smart Contracts. In *EuroS&P*, 2019.

[15] Iddo Bentov, Yan Ji, Fan Zhang, Yunqi Li, Xueyuan Zhao, Lorenz Breidenbach, Philip Daian, and Ari Juels. Tesseract: Real-Time Cryptocurrency Exchange using Trusted Hardware. In *CCS*, 2017.

[16] Olga Ohrimenko, Felix Schuster, Cedric Fournet, Aastha Mehta, Sebastian Nowozin, Kapil Vaswani, and Manuel Costa. Oblivious multi-party machine learning on trusted processors. In *USENIX Security*, 2016.

[17] Shruti Tople, Karan Grover, Shweta Shinde, Ranjita Bhagwan, and Ramachandran Ramjee. Privado: Practical and Secure DNN Inference. *ArXiv*, 2018.

[18] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-order Execution. In *USENIX Security*, 2018.

[19] Wenhao Wang, Guoxing Chen, Xiaorui Pan, Yinqian Zhang, XiaoFeng Wang, Vincent Bindschaedler, Haixu Tang, and Carl A. Gunter. Leaky Cauldron on the Dark Land: Understanding Memory Side-Channel Hazards in SGX. In *CCS*, 2017.

[20] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. Software Grand Exposure: SGX Cache Attacks Are Practical. In *WOOT*, 2017.

[21] Michael Schwarz, Samuel Weiser, Daniel Gruss, Clémentine Maurice, and Stefan Mangard. Malware Guard Extension: Using SGX to Conceal Cache Attacks. In *DIMVA*, 2017.

[22] Ahmad Moghimi, Gorka Irazoqui, and Thomas Eisenbarth. CacheZoom: How SGX Amplifies the Power of Cache Attacks. In *CHES*, pages 69–90. Springer, 2017.

[23] Jo Van Bulck, Nico Weichbrodt, Rüdiger Kapitza, Frank Piessens, and Raoul Strackx. Telling Your Secrets Without Page Faults: Stealthy Page Table-based Attacks on Enclaved Execution. In *USENIX Security*, 2017.

[24] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems. In *S&P*, 2015.

[25] Martin Maas, Eric Love, Emil Stefanov, Mohit Tiwari, Elaine Shi, Krste Asanovic, John Kubiatowicz, and Dawn Song. PHANTOM: Practical Oblivious Computation in a Secure Processor. In *CCS*, 2013.

[26] Victor Costan and Srinivas Devadas. Intel SGX Explained. Cryptology ePrint Archive, Report 2016/086, 2016. http://eprint.iacr.org/2016/086.

[27] Andrew Huang. Keeping Secrets in Hardware: The Microsoft Xbox™ Case Study. In *CHES*, 2003.

[28] Oleksii Oleksenko, Bohdan Trach, Robert Krahn, Andre Martin, Christof Fetzer, and Mark Silberstein. Varys: Protecting sgx enclaves from practical side-channel attacks. In *USENIX ATC*, 2018.

[29] Guoxing Chen, Wenhao Wang, Tianyu Chen, Sanchuan Chen, Yinqian Zhang, XiaoFeng Wang, Ten-Hwang Lai, and Dongdai Lin. Racing in Hyperspace: Closing Hyper-Threading Side Channels on SGX with Contrived Data Races. In *S&P*, 2018.

[30] Daniel Gruss, Julian Lettner, Felix Schuster, Olya Ohrimenko, Istvan Haller, and Manuel Costa. Strong and Efficient Cache Side-Channel Protection using Hardware Transactional Memory. In *USENIX Security*, 2017.

[31] Ming-Wei Shih, Sangho Lee, Taesoo Kim, and Marcus Peinado. T-SGX: Eradicating Controlled-Channel Attacks Against Enclave Programs. In *NDSS*, 2017.

[32] Sanchuan Chen, Xiaokuan Zhang, Michael K. Reiter, and Yinqian Zhang. Detecting Privileged Side-Channel Attacks in Shielded Execution with DéJà Vu. In *AsiaCCS*, 2017.

[33] Xiao Shaun Wang, Kartik Nayak, Chang Liu, T.-H. Hubert Chan, Elaine Shi, Emil Stefanov, and Yan Huang. Oblivious Data Structures. In *CCS*, 2014.

[34] Emil Stefanov, Marten van Dijk, Elaine Shi, Christopher Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. Path ORAM: An Extremely Simple Oblivious RAM Protocol. In *CCS*, 2013.

[35] Pratyush Mishra, Rishabh Poddar, Jerry Chen, Alessandro Chiesa, and Raluca Ada Popa. Oblix: An efficient oblivious search index. In *S&P*, 2018.

[36] Shaizeen Aga and Satish Narayanasamy. InvisiMem: Smart Memory Defenses for Memory Bus Side Channel. In *ISCA*, 2017.

[37] Amro Awad, Yipeng Wang, Deborah Shands, and Yan Solihin. ObfusMem: A Low-Overhead Access Obfuscation for Trusted Memories. In *ISCA*, 2017.

[38] QEMU: the FAST! processor emulator. https://www.qemu.org/. Last accessed: December 2, 2019.

[39] Intel Software Guard Extensions Programming Reference. https://software.intel.com/sites/default/files/managed/48/88/329298-002.pdf. Last accessed: December 2, 2019.

[40] Andrew Ferraiuolo, Andrew Baumann, Chris Hawblitzel, and Bryan Parno. Komodo: Using verification to disentangle secure-enclave hardware from software. In *SOSP*, 2017.

[41] ARM Security IP CryptoIsland Family. https://www.arm.com/products/silicon-ip-security/cryptoisland. Last accessed: December 2, 2019.

[42] AMD Secure Encrypted Virtualization. https://developer.amd.com/amd-secure-memory-encryption-sme-amd-secure-encrypted-virtualization-sev/. Last accessed: December 2, 2019.

[43] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache Attacks and Countermeasures: The Case of AES. In *CT-RSA*, 2006.

[44] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. Last-Level Cache Side-Channel Attacks Are Practical. In *S&P*, 2015.

[45] Yuval Yarom and Katrina Falkner. FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-channel Attack. In *USENIX Security*, 2014.

[46] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. Flush+Flush: A Fast and Stealthy Cache Attack. In *DIMVA*, 2016.

[47] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. ZombieLoad: Cross-Privilege-Boundary Data Sampling. In *CCS*, 2019.

[48] Sangho Lee, Ming-Wei Shih, Prasun Gera, Taesoo Kim, Hyesoon Kim, and Marcus Peinado. Inferring Fine-grained Control Flow Inside SGX Enclaves with Branch Shadowing. In *USENIX Security*, 2017.

[49] Yeongjin Jang, Jaehyuk Lee, Sangho Lee, and Taesoo Kim. SGX-Bomb: Locking Down the Processor via Rowhammer Attack. In *SysTEX*, 2017.

[50] Victor van der Veen, Yanick Fratantonio, Martina Lindorfer, Daniel Gruss, Clementine Maurice, Giovanni Vigna, Herbert Bos, Kaveh Razavi, and Cristiano Giuffrida. Drammer: Deterministic Rowhammer Attacks on Mobile Platforms. In *CCS*, 2016.

[51] Marcus Brandenburger, Christian Cachin, Matthias Lorenz, and Rüdiger Kapitza. Rollback and Forking Detection for Trusted Execution Environments using Lightweight Collective Memory. In *DSN*, 2017.

[52] Sinisa Matetic, Mansoor Ahmed, Kari Kostiainen, Aritra Dhar, David Sommer, Arthur Gervais, Ari Juels, and

Srdjan Capkun. ROTE: Rollback Protection for Trusted Execution. In *USENIX Security*, 2017.

[53] Peter Pessl, Daniel Gruss, Clémentine Maurice, Michael Schwarz, and Stefan Mangard. DRAMA: Exploiting Dram Addressing for Cross-CPU Attacks. In *USENIX Security*, 2016.

[54] Chia-che Tsai, Donald E. Porter, and Mona Vij. Graphene-SGX: A Practical Library OS for Unmodified Applications on SGX. In *ATC*, 2017.

[55] JKI Inc. JLA320A. https://www.jkic.co.kr/ddr4-protocol-analyzer. Last accessed: December 2, 2019.

[56] Kibra 480 Analyzer. http://cdn.teledynelecroy.com/files/pdf/lecroy_kibra480_datasheet.pdf. Last accessed: December 2, 2019.

[57] Nexus Technology MA4100. https://www.nexustechnology.com/products/memory-analyzers/ma4100-series-memory-analyzer/. Last accessed: December 2, 2019.

[58] Hunspell. http://hunspell.github.io/. Last accessed: December 2, 2019.

[59] Brad Fitzpatrick. Distributed caching with memcached. *Linux journal*, 2004(124):5, 2004.

[60] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C. Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, David Stafford, Tony Tung, and Venkateshwaran Venkataramani. Scaling Memcache at Facebook. In *NSDI*, 2013.

[61] James Langston. Enhancing the Scalability of Memcached. https://software.intel.com/en-us/articles/enhancing-the-scalability-of-memcached. Last accessed: December 2, 2019.

[62] Yupeng Zhang, Jonathan Katz, and Charalampos Papamanthou. All Your Queries Are Belong to Us: The Power of File-Injection Attacks on Searchable Encryption. In *USENIX Security*, 2016.

[63] Intel 64 and IA-32 Architectures Software Developer's Manual Volume 3A: System Programming Guide, Part 1. https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-vol-3a-part-1-manual.pdf. Last accessed: December 2, 2019.

[64] Mengjia Yan, Read Sprabery, Bhargava Gopireddy, Christopher Fletcher, Roy Campbell, and Josep Torrellas. Attack Directories, Not Caches: Side Channel Attacks in a Non-Inclusive World. In *S&P*, 2019.

[65] Khang T Nguyen. Introduction to Cache Allocation Technology in the Intel® Xeon® Processor E5 v4 Family. https://software.intel.com/en-us/articles/introduction-to-cache-allocation-technology, Febuary 2016.

[66] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Daniel O'Keeffe, Mark L. Stillwell, David Goltzsche, Dave Eyers, Rüdiger Kapitza, Peter Pietzuch, and Christof Fetzer. SCONE: Secure Linux Containers with Intel SGX. In *OSDI*, 2016.

[67] Software Guard Extenstion (SGX) SDK for Linux. https://github.com/intel/linux-sgx. Last accessed: December 2, 2019.

[68] RISC-V ISA Simulator. https://riscv.org/software-tools/risc-v-isa-simulator/. Last accessed: December 2, 2019.

[69] Spell Checker Oriented Word Lists. http://wordlist.aspell.net/. Last accessed: December 2, 2019.

[70] Enron Email Dataset. https://www.cs.cmu.edu/~./enron/. Last accessed: December 2, 2019.

[71] NLTK data 3.4.5 documentation. https://www.nltk.org/data.html. Last accessed: December 2, 2019.

[72] GNU Privacy Guard. http://www.gnupg.org. Last accessed: December 2, 2019.

[73] Sajin Sasy, Sergey Gorbunov, and Christopher W. Fletcher. ZeroTrace : Oblivious Memory Primitives from Intel SGX. In *NDSS*, 2017.

[74] J Thomas Pawlowski. Hybrid Memory Cube (HMC). In *2011 IEEE Hot Chips 23 Symposium (HCS)*, 2011.

[75] Oliver Kömmerling and Markus G Kuhn. Design Principles for Tamper-Resistant Smartcard Processors. In *Smartcard*, 1999.