High-Resolution Side Channels for Untrusted Operating Systems

Marcus Hähnel, ***TU Dresden, Operating Systems Group;***
Weidong Cui and Marcus Peinado, ***Microsoft Research***

https://www.usenix.org/conference/atc17/technical-sessions/presentation/hahnel

This paper is included in the Proceedings of the 2017 USENIX Annual Technical Conference (USENIX ATC '17).

July 12–14, 2017 • Santa Clara, CA, USA

ISBN 978-1-931971-38-6

Open access to the Proceedings of the 2017 USENIX Annual Technical Conference is sponsored by USENIX.

High-Resolution Side Channels for Untrusted Operating Systems

Marcus Hähnel [*, †]
TU Dresden
mhaehnel@tudos.org

Weidong Cui
Microsoft Research
wdcui@microsoft.com

Marcus Peinado
Microsoft Research
marcuspe@microsoft.com

Abstract

Feature-rich mass-market operating systems have large trusted computing bases (TCBs) and a long history of vulnerabilities. Systems like Overshadow, InkTag or Haven attempt to remove the operating system (OS) from the TCB of applications while retaining its functionality. However, the untrusted OS's control of most physical resources puts it in a much better position to launch side-channel attacks than traditional unprivileged side-channel attackers. Initial attacks focused on the page-fault channel, demonstrating significant information leakage for three *legacy* applications.

We present two new side channels for an untrusted OS which use timer interrupts and cache misses to achieve higher temporal and spatial resolution than the page-fault channel. We leverage the untrusted OS's control over hardware to reduce noise in the side channels to enable successful attacks in just a single run of the target. We demonstrate that our side channels enable attacks against *new* SGX applications such as VC3 that were designed *not* to trust the OS. We also show a new attack against `libjpeg` that extracts images with two orders of magnitude more information than the page-fault channel attack.

1 Introduction

Traditionally, the operating system (OS) protects the integrity and confidentiality of application data and is considered part of the trusted computed base (TCB) of an application. However, decades of experience have shown that it is extremely hard to protect large, feature-rich and widely deployed commodity operating systems. The emergence of cloud hosting services has added the new threat of adversarial cloud operators.

Recently, systems like Overshadow [14], InkTag [27] and Haven [9] were proposed to change the protection paradigm by excluding the OS from the TCB and directly protecting applications. These systems use a trusted hypervisor or hardware to provide applications with memory that is protected from the untrusted OS and with a controlled mechanism for transferring control between applications and the untrusted OS. Haven can also protect applications from Iago attacks [13].

However, Xu *et al.* [44] demonstrate that an adversarial OS can launch deterministic side-channel attacks against protected applications running on SGX. Their attacks can steal documents and outlines of JPEG images from single runs of three legacy applications protected by Haven and InkTag. The attacks are more powerful than traditional side-channel attacks by unprivileged attackers because the untrusted OS retains control of most of the hardware. They exploit the fact that some memory accesses of an application depend on secret data. An adversarial OS observes these accesses by making pages inaccessible in the page table. Each resulting page fault interrupts the application at the moment of the access and reveals the page address to the OS, allowing the adversary to infer secrets that influence the sequence of memory accesses.

Functions within a single code page (e.g., tight loop in *strlen*) and data accesses within a single page (e.g., indexing into small arrays) cannot be observed, as the granularity of the page-fault channel is limited by the 4 KB page size. This imposes a fundamental limit on the temporal and spatial resolution of page-fault based attacks.

In this paper, we present two new side channels that significantly improve the temporal and spatial resolution of attacks launched from an adversarial OS. We demonstrate that new attacks can be launched against applications that are immune to attacks based solely on the page-fault channel. To improve temporal resolution, we use a high-precision timer to approximate *single stepping*. We use a cache side channel to improve the spatial resolution from 4 KB to 64 byte cache lines. Our cache side channel has much higher accuracy than the traditional cache side channel controlled by an unprivileged attacker because the OS controls the hardware. The OS can break into an application right before and after the memory access of interest and reduce cache pollution through its control over scheduling.

To demonstrate the power of our new side channels, we build new attacks against VC3 [39] and `libjpeg` [1]. VC3 is a secure MapReduce framework that protects the confidentiality of distributed MapReduce computations by running them inside SGX enclaves [29]. The unmodified legacy applications attacked in [44] had been written under the assumption that the OS was trusted. However,

[*] Work done in part during an internship at Microsoft Research.

[†] Supported in part by the German Research Foundation (DFG) within the CRC 912 - HAEC.

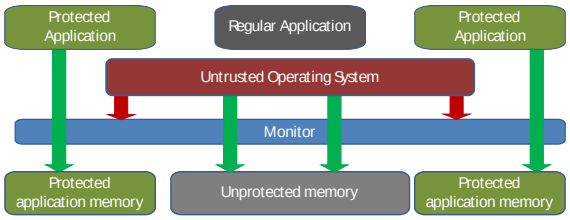

Figure 1: System model: A monitor constrains the OS and prevents it from accessing the protected applications' memory.

VC3 is new code designed to have a small trusted computing base and to run in an SGX enclave on an adversarial OS. Our attack on VC3 is a sequence of attacks that extract various pieces of information from different parts of the VC3 code. None of the individual attacks could have been performed using only the page-fault channel. Our attack on VC3 recovers almost two thirds of the input documents of the WordCount application. Compared to the earlier attack against `libjpeg` [44], our new attack can extract two orders of magnitude more information and recover images with richer detail.

This paper makes the following contributions: We demonstrate

- two new high-resolution side channels for an untrusted OS to attack protected applications;
- a significantly improved attack against `libjpeg` and a new attack against VC3;
- the increased significance of side-channel attacks for untrusted operating systems.

2 System Model

As outlined in Figure 1, the system consists of one or more applications portected by a monitor that constrains the untrusted OS.

The Monitor can be a hypervisor or a part of the CPU. It imposes the following constraints on the OS:

1. It guarantees safe memory to a trusted application. The OS cannot read or write such memory.
2. It controls transitions between an application and the OS, ensuring that the OS can invoke protected code only at well defined entry points. This also allows the monitor to protect the application from leaking CPU state during the transition. On unexpected transitions (traps, interrupts), the monitor saves and scrubs CPU registers.
3. The monitor takes anti side-channel provisions.

We assume the following list of anti side-channel measures. It is based on SGX [28], which, at this time, is the only technology that takes serious measures against side channels.

- The bottom twelve bits of page fault addresses (which specify the offset within the page) are hidden from the OS.
- Hardware debugging facilities (e.g., debug registers, single stepping) are disabled.
- Hardware performance counters are disabled.

The Untrusted OS is considered adversarial—either as a result of malware or because it is under the control of an adversarial administrator. Within the constraints imposed by the monitor, it can take any action to corrupt the application or extract information from it. It has access to all hardware on the system, except as prohibited by the monitor. This includes all unprotected memory, page tables and system devices such as timers.

Applications are protected by the monitor. We target native applications that process secret information and display memory access patterns that depend on the secret information. We assume that at least some parts of the application binaries are public, either because they belong to unmodified legacy applications [9,14,27,44] or because they are a part of a public application platform [39].

3 Background

We describe SGX [29], the page-fault channel [44], and prime-and-probe cache side-channel attacks [36].

3.1 Intel SGX

Intel SGX [6, 26, 29, 34] is a CPU technology recently introduced by Intel. SGX allows the construction of isolated memory regions for applications (*enclaves*) that are protected by the CPU from all other software running on the system, including the OS. In terms of our system model, SGX constitutes a monitor. Enclaves are restricted to running in user mode, but are constructed by the untrusted OS using new privileged instructions. Remote attestation ensures that the owner of the application can detect tampering by the OS before revealing any secrets.

SGX leaves the hardware interface for the OS largely unchanged, enabling compatibility with legacy operating systems, but also providing the OS with a large tool chest for building side channels, such as the page-fault channel [44].

However, SGX includes a number of anti side-channel measures. SGX makes it impossible to use the following CPU features against enclaves: Hardware breakpoint registers (DR0-DR3), single stepping (RFLAGS.TF), Last Branch Record (LBR), Precise Event-Based Sampling (PEBS), and hardware performance counters. Upon an exception or fault inside an enclave, SGX masks the twelve least-significant bits of the faulting address. Thus, the untrusted OS only receives page-granular fault information.

3.2 The Page-Fault Channel

Xu *et al.* describe side-channel attacks based on page faults [44].The attacks take advantage of *input-dependent*

```
size_t strlen(char* str) {
  size_t len = 0;
  while (*str++ != '\0') len++;
  return len;
}
```

(a) `strlen`

```
char* agentNames[] = {"James R. Clapper", "John Doe",
                      /* 510 more entries */  };
char* getAgentName(int agentCode) {
  return agentNames[agentCode];
}
```

(b) Array Access

Figure 2: The secret-dependent memory access does not leak information if the attacker can only observe at page granularity.

memory accesses to infer an application's secret input. For instance, if a global variable is incremented for each user login, an adversary can count the number of accesses to this variable to infer the number of logins. To detect that the global variable is accessed, the adversary can make the page where it resides inaccessible. Any access to the page will trigger a page fault.

In the simplest case, the adversary can use the fault address to decide if the application tried to access the global variable of interest. However, since the offset in the fault address is hidden, the adversary has to infer it. Xu *et al.* propose using page fault *sequences* that can uniquely identify a data access or control transfer.

Attacks based on page-fault sequences still face a fundamental limit: *Page faults only work at page granularity*. The following two examples are immune to attacks via the page-fault channel.

Temporal Limit: Figure 2a shows a simple `strlen` implementation. Assume the binary code is on a single code page and the string `str` is on a single data page. The `len` variable is usually in a register for optimization and thus not usable by an adversary. One instruction in `strlen` reads a character of `str` from the data page. To execute it, the attacker must make both code and data page accessible. But after this, `strlen` execution will not cause further page faults, preventing the adversary from counting the number of iterations and inferring the length of the string.

Spatial Limit: Assume table `agentNames` in Figure 2b is on a single data page. Since the offset of the page-fault address is hidden, an adversary cannot use an access to the table to infer the value of the variable `agentCode`.

3.3 Prime-and-Probe Cache Side-Channel Attacks

Small, fast caches in CPUs are used to mitigate the performance impact of memory accesses. Upon a memory access, the CPU copies the memory contents into the cache. Subsequent accesses to the same address can be serviced from the cache at a much lower cost. This copying takes place at the granularity of 64 byte *cache lines*.

A memory location is mapped to a small group of cache lines based on some of its address bits (bits 6-11 for the Intel L1 data cache used in this paper). The Intel L1 cache uses groups of 8 cache lines (*8-way set associative*). Upon a memory access not currently in the cache, one line in the group must be *evicted* to make space for the new contents.

Caches are typically shared by all code running on a core or even the entire CPU package. An attacker may make a sequence of memory accesses loading the contents into the cache and filling it completely. A subsequent memory access by another program will also cause data to be loaded into the cache, evicting one of the attacker's cache lines.

The attacker can measure the time it takes to access the memory locations he previously loaded into the cache to detect such evictions. Increased access times indicate an eviction, telling the attacker not only that an access took place, but also revealing bits 6-11 of the address of the access.

This procedure is known as *prime and probe* [36]. Filling the cache with the attacker's content is called the *prime* step. Measuring the memory access times is called the *probe* step.

4 Design

This section describes our techniques for performing synchronous, high-resolution side-channel attacks. Abstractly, the attacker has two capabilities: *break*, to set conditional breakpoints on the application and *observe*, to inspect artifacts of its execution (e.g., memory accesses). A technique may also provide both capabilities. Making a page inaccessible [44] allows breaking (page fault) and observing (page fault address).

We introduce two new techniques to overcome the limitations of the page-fault channel (Section 3.2) and significantly broaden the class of application code subject to side-channel attacks:

- A technique to single-step protected applications.
- A cache side channel to observe memory accesses at cache-line granularity.

Both techniques work even if the attacker can observe only a single application run. Thus, any protected application is a potential target for the attack.

We use page-fault sequences to infer memory accesses when possible, allowing us to deploy high-overhead attacks only during the short times they are required.

4.1 Noise reduction

The techniques require a very low level of system noise. We use the OS's control over hardware to reduce noise by disabling interference sources (turbo-boost, prefetching, power management) and preventing preemption of the

victim application. These adjustments can also be made by a compromised OS or a malicious admin.

Like page-fault based attacks [44], our attacks rely on exceptions and interrupts to provide the *break* mechanism. Handlers run on a victims's core to gain access to the private resources (e.g., caches) of the core for *observation*.

4.2 Single Stepping

In our system model (and under SGX), the monitor prevents the OS from using the single stepping features of the hardware. However, the OS can approximate such functionality using hardware timers.

For this work, we use the x86 local APIC timer, due to its high resolution and easy programmability. We use the timer in single-shot mode at the highest available frequency (divider=1). In this mode, the OS writes a target value x into a register. The timer triggers an interrupt x timer ticks after the register write. The timer tick frequency is significantly lower than the TSC frequency. On the Skylake system used in the evaluation, the former is 24 MHz, while the latter is 167 times higher. While this means we can only trigger an interrupt every 167 CPU cycles, we know at which TSC value the interrupt will arrive, allowing us to wait an appropriate time should the desired TSC value be too soon.

Assume we have interrupted the application and want to single-step forward to the next instruction. This may not be easy for all instructions. Here, we focus on instructions with memory operands, which is sufficient for our attacks. The goal is to have the next interrupt arrive during the execution of the next such instruction. If the interrupt arrives during the right time window, the processor will delay the interrupt until execution of the instruction is complete. During regular execution, this time window is extremely short. However, the attacker has an array of tools that can make the instruction very slow and, thus, expand the time window to hundreds of cycles. We only rely on a TLB flush, which causes instructions with memory operands to incur a page-table walk. Additional options include flushing the cache to force page-table walks to incur the full memory latency, manipulating the memory clock to increase memory latency or disabling the cache completely to make all subsequent instructions slow.

The time between starting the timer (writing to the timer register) and executing the victim's next instruction includes the interrupt return path (privilege level change) and the re-entry path of the protected application (crossing the trust boundary, restoring registers). While CPU specific, this time can be measured using a protected application on the attacker-controlled target system. Based on this measurement, we determine the number of timer ticks x. This estimate can only be sufficiently precise if jitter is low, making the noise reduction techniques described above vital.

This mechanism can be used to implement any conditional breakpoint that depends only on attacker-observable information. Single stepping and observing the system after each step allow the attacker to perform a detailed analysis of the system's behavior. While possible for the whole application, it is also quite slow, as each instruction causes at least one interrupt. The overhead can be reduced by using a cheaper breakpoint to narrow down the region of interest (page fault, coarse-grained timer interrupts) and switch to single-stepping there.

We synchronize our single steps with observations about the memory accesses made by the application. For example, the `strlen` code in Figure 2a accesses the string exactly once per iteration. Observing an access to `str` after a single step informs us that the application must be at (close to) the instruction following the access. We obtain the number of loop iterations by counting the number of `str` accesses. Memory accesses are observable by reading the dirty and accessed bits in the page table entries.

4.3 Cache Side-Channel Attack

A cache side-channel allows observations at the granularity of 64 byte cache lines. It has the potential of revealing information about accesses to small (sub-page sized) arrays for which the page-fault channel is ineffective. We use a prime-and-probe attack against the core-local, 32 KB large, 8-way set associative L1 cache.

The key challenge for cache side-channel attacks is *cache pollution* caused by other accesses than the ones of interest during prime and probe. An unprivileged attacker has little control over when his code will execute. Probing may observe the results of both the memory accesses of interest and potentially many unrelated accesses, a problem exacerbated by the small size of the L1 cache.

Traditional cache side-channel attacks mitigate the problem by averaging over many prime-and-probe observations. This technique does not apply to applications that usually execute over each unique input once. Instead, we use our control over hardware to tackle the problem.

Our attack proceeds as follows. We break the application shortly before the memory access we wish to observe, prime the L1 cache, and resume the application. We break again shortly after the access of interest, probe the L1 cache, log the result and resume the application.

Thanks to the OS's control over hardware, we can use page faults or single-stepping to break right before and after the memory access of interest, thus avoiding unrelated memory accesses during prime and probe. To further reduce cache pollution, we prevent other applications from running on the victim's physical core.

Even after applying these mitigation techniques, we still cannot eliminate cache pollution completely for two reasons. First, some code is executed between priming and the resumption of the application and between the interruption of the application and probing. Any memory accesses by such code may result in spurious cache evictions. Second, TLB flushes happen when transitioning into and out of protected applications to protect application memory. This leads to page-table walks when executing subsequent instructions. This in turn causes one memory access per page-table level.

However, we can predict and to some degree even control which cache addresses are being polluted and adjust our evaluation accordingly. Given the 8-way set associativity of the L1 cache, a polluted cache address will also not lose all information about the access we are trying to observe. If the access of interest falls into a cache address polluted by one extraneous access, we should observe L1 misses for two of the eight ways for that address. This allows us to observe the access of interest even in the presence of deterministic cache pollution.

We determine the set of cache lines that will be polluted deterministically on every prime-probe observation. This includes the memory accesses made by the page-table walk for the (known) target page of the access of interest and the (known) page of the page-fault handler, as well as some known accesses made by the handler itself. In our analysis, we subtract these L1 misses from our observations. We call this step *deterministic filtering*.

When probing, we must measure access times very precisely, as L2 hits take only a few cycles longer than L1 hits. We disable interrupts to gain exclusive use of the core for our measurement code.

5 Attacks

We now describe our attacks against VC3 and `libjpeg`.

5.1 VC3

VC3 [39] allows users to run MapReduce jobs [18] in the cloud without exposing their code or data to the provider. VC3 shields computations from the provider using SGX. The provider and all his software (OS, hypervisor) are assumed to be adversarial. Just as in regular MapReduce, the user writes a *map* and a *reduce* function. The functions are encrypted, packaged together with the VC3 framework and sent to the cloud to be run in SGX enclaves. VC3 is designed to plug into an existing untrusted MapReduce framework such as Hadoop [7]. A VC3 job will begin with encrypted input splits that the untrusted MapReduce framework feeds into *mapper* enclaves. Inside a mapper enclave, the VC3 framework decrypts the input and the user's map function, and invokes it. The

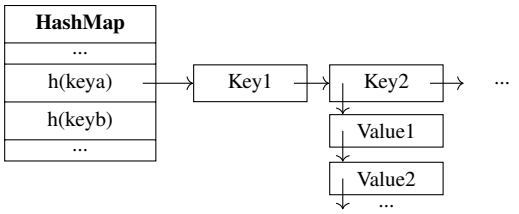

Figure 3: Layout of the hash map used in the VC3 reducers

VC3 framework encrypts intermediate key-value pairs produced by the map function and feeds them into the untrusted MapReduce framework, which distributes them to *reducer* enclaves.

VC3 provides tamper detection for the untrusted communication channel (e.g., removal or duplication of intermediate key-value pairs). Randomized encyption prevents frequency analysis on intermediate key-value pairs. But it also stops the untrusted MapReduce framework from grouping them. Since grouping is required by the *intermediate key-value rule* [18, Sec. 2] VC3 has to implement it inside the reducer enclaves.

The grouping implementation is hash-map based (Figure 3). Each hash key is mapped to an index in an array of 8-byte pointers. Hash collisions are resolved through a linked list of colliding keys for each array index. As the array is 64-byte aligned, eight consecutive 8-byte pointers fill up a cache line. Conversely, observation of a cache line access identifies the corresponding eight array slots. This grouping operation is the only part of VC3 whose memory accesses depend on user data, because most mappers use user data to compute intermediate keys.

5.1.1 Attack Overview

Our attack targets MapReduce applications that have one or more English documents as the input and words in the documents as the intermediate keys. WordCount and Inverted Index are such applications [18, Section 2]. Our goal is to recover as much of the document as possible.

Following the VC3 model, we consider the user's map and reduce functions to be secret, but the VC3 framework to be public, and thus only attack the latter. It would negate the value proposition of VC3 if users were forced to admit code into their TCB that they cannot inspect.

The high-level idea of our attack works as follows. We use our single stepping and cache side channel techniques to infer the length and the (approximate) hash array index for each input word (i.e., intermediate key). We also track words through the various stages of processing in MapReduce to remember their positions in the input document. Finally, we use an English language model that contains a dictionary of words (unigrams) and word pairs (bigrams) as well as their weights (measured by their popularity)

to recover the input document based on the length, hash array index and position of each word.

5.1.2 Word Length

VC3 hashes all intermediate keys (i.e., input words) to insert the key-value pairs into the hash map. The hash function loops over the characters of the key until it finds a null character. The length of the input word is obtained by breaking on the hash function and observing the number of loop iterations it performs using the technique of Section 4.2.

5.1.3 Cache Line Address

We use the cache side channel of Section 4.3 to observe for each word the cache line of the hash array slot into which it is inserted. We break right before the array lookup by making the array pages inaccessible. Upon the page fault, we log the page number, make the page accessible, make the reducer's stack inaccessible and prime the cache. After resuming execution, VC3 accesses the array and, immediately after that, page faults when trying to access its stack. We probe and log the cache state, resolve the page fault, make the array page inaccessible again and resume execution.

5.1.4 Word Position

As input words move through the mappers and reducers, we must keep track of their positions in the input document. Our English language model relies on word order and, more generally, the output of the attack appears much more useful if it presents the words in the correct order.

If there is only a single mapper and a single reducer, the problem is trivial because the words arrive at the reducer (where we observe word lengths and hash slots) in input order. However, multiple mappers and reducers will be sending and receiving intermediate key-value pairs concurrently. Furthermore, VC3 mappers internally determine the reducer for each word, buffer the word and only send buffers containing many words to each of the reducers.

The first problem (concurrency) is easily solved by observing that the attacker controls the communication channel between all mapper and reducer enclaves. In particular, the attacker can observe the order of all messages sent from mappers to reducers. The second problem is more complex. Using the page-fault channel to monitor the mappers, we track for each input word the buffer into which the mapper inserts it and its position in the buffer (details omitted due to space constraints). This information allows us to recover the original word positions when a reducer finally processes the words from the buffer; i.e., when we extract the length and hash slot for each word from the buffer.

5.1.5 Word Recovery

We can represent the information recovered so far as a version of the input document in which each word has been replaced by its length and its hash slot (covering 8 array indices). It remains to map this information back to the original words.

As a first step, we group the words in our language model by length and hash slot. The result is a candidate list for each length, hash slot combination and, thus, for each position in our input document. Next, we refine the candidate list for each input word with the help of context: the bigrams from our language model. For each candidate word at a position, if there is no candidate word in the subsequent position to construct a word pair (bigram) that is contained in our language model, we eliminate the candidate word. We repeat this pruning step on all candidate words iteratively until no candidate words can be removed. Finally, we sort all remaining candidate words for each position based on their weights.

5.2 JPEG

JPEG is one of the most widely used image compression standards. JPEG compression cuts the image into blocks of 8 by 8 pixels and performs a discrete cosine transform (DCT) on each block, followed by other compression steps. JPEG decompression reverses these steps, performing an inverse DCT as one of the last steps. In our attack, we target the `libjpeg` library [1], the most widely used JPEG implementation. Specifically, we exploit the last stage of the inverse DCT function which computes the final values of the 8 by 8 output matrix by means of array lookups. The array has 1024 single byte entries and lies typically on a single page, which makes page-granular observation useless. The final output values are the values read from the array.

Our attack strategy is to observe the cache line accessed in each of the array lookups. However, even at cache-line granularity, we are unable to distinguish between the 64 adjacent array indices that fall into each cache line. For example, if the array is 64 byte aligned, array slots 0 to 63 fall into the same cache line, and our observations do not let us distinguish among them. If the array was filled with random numbers, observing cache lines would be unlikely to reveal useful information.

Fortunately, the array values are either constant or linearly increasing over large ranges. Thus, the average over all array values that lie in the same cache line contains useful information. For cache lines that cover array regions where the values increase linearly, the avarage contains the two most-significant bits of the 8-bit array

values, as we are losing the 6 ($= log_2(64)$) least significant bits due to 64-bit cache-line resolution. For cache lines that cover constant regions, the average contains the same information as the individual array values.

Upon observing an array access at a particular cache line, we use the average over all array values that are covered by that cache line as our inferred output value. We feed these recovered values directly into the last phases of JPEG decompression to obtain the final image. We recover the image dimensions and the color space using the method described in [44].

6 Implementation

This section describes how we implemented the attacks and the target applications. Our prototype was designed for x86-64 PCs running Windows and using Intel SGX as the monitor. The choice of Windows was not essential. A dedicated attacker might choose to write a special attack OS or even build special hardware that gives him easy access to the required functionality. We used the shortcut of adding attack functionality to an existing OS by means of a kernel driver. We chose SGX as the monitor for three reasons. (1) VC3 only runs on SGX. (2) SGX has a detailed, public specification. (3) SGX includes defenses against side-channels, making it a more interesting target.

6.1 Implementation on Windows

We used a Windows driver to implement the techniques from Section 4 and the page fault channel in approximately 1,200 lines of C++ code and 250 lines of assembly code. All binaries were compiled with the Microsoft C/C++ compiler version 18.00.21005.1 with full optimization (`/Ox`) and inlining (`/Ob2`).

The driver hooks the timer handler and the page fault handler in the interrupt descriptor table (IDT). This causes the processor to invoke our handlers, rather than the OS's. Our driver processes all events intended for it and forwards all others to the Windows handlers.

We pinned the target application to a single core and set its scheduling priority to `REALTIME` in order to minimize interference from other OS activity. The core still receives interrupts and other system events which are a source of residual noise. For ease of implementation, we disabled hyper-threading, turbo-boost, pre-fetching and power management in the BIOS. The same can be done in code by the OS.

6.2 SGX

We used the SGX simulator that was used in the evaluation of VC3 [39]. The simulator tries to faithfully implement the essential functionality of SGX in software. For the purposes of our attacks, only the SGX behavior on transitions into and out of enclaves is relevant. In particular, the simulator adds TLB flushes and delay cycles on all transitions. We simulated the saving of the register state by SGX by saving all the registers to a memory page and putting them in a defined state. An equivalent alternative would have been to use the OpenSGX simulator [31].[1]

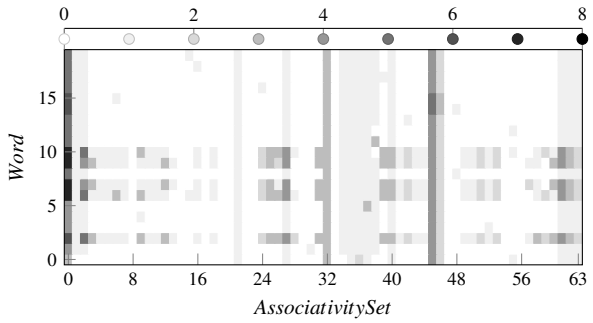

Figure 4: Results of prime-probe observations for 20 distinct words (rows). Darker fields indicate more evicted ways within an 8-way associativity set. Vertical lines identify cache addresses evicted in every observation.

6.3 Single Stepping

As Windows uses the local APIC timer as its system timer we had to share it. We programed the timer, such that it continues to trigger the relatively low-frequency periodic interrupts Windows expects. When single stepping, we set up the timer for single-shot mode with a divider of one.

6.4 Cache Side-Channel Attacks

At initialization, we allocate a page-aligned 32 KB buffer consisting of eight 4KB pages, which correspond to the eight ways of the L1 data cache. Our *prime* procedure loops over the buffer and performs one write operation for each cache line, thus filling the entire L1 data cache.

Our *probe* procedure also iterates over the cache lines covered by our buffer. For each cache line, it times the corresponding memory read using `rdtsc` using serializing and fence instructions to prevent reordering. Accesses that take at most 10 cycles are considered L1 hits.

Our IDT hooks let us control all code executed when transitioning into and out of the victim application. The separation of the L1 cache into an instruction cache and a data cache ensures that code execution by itself does not pollute the L1 data cache. We carefully chose all assembly instructions in our fault handler to control its data memory accesses.

Figure 4 shows the results of 20 prime-probe observations. Each row shows a cache fingerprint for a different

[1] Intel has released an SGX SDK, which only allows enclaves to run in debug mode. This mode disables most protections, thus providing limited additional value over the existing simulator.

word. Thus, we would expect each row to only differ in one gray dot for the accessed cache line. This does not explain all observed differences between the rows. The vertical lines (e.g., at index 32) stem from deterministic cache pollution since they are evicted in every prime-probe observation. We are uncertain as to the exact cause of the noise patterns between rows 5 and 10.

While the pollution is not constant across observations, there appears to be a relatively small number of pollution patterns. For example, lines 6, 7, 9 and 10 have roughly the same pollution pattern. Our strategy was to group our observations by their pollution patterns and to remove cache misses that are common within each group from each of the observations in the group. For this, we computed for each group the average number of observed misses for each cache line and subtracted these averages from our observations, setting negative results to zero. This *noise filtering* removed most of the pollution, significantly reducing the number of candidate cache lines.

6.5 VC3

We used the original VC3 code [39], implemented WordCount [18, Sec. 2.1] in 100 lines of C, and used the VC3 packing tool to generate the binary that is loaded into the enclaves. This tool encrypts the binary containing the mapper and reducer functions and embeds it into a second (plaintext) binary containing the trusted VC3 framework which is later loaded into the enclaves. We used the VC3 preprocessing tool to convert the input document into encrypted input splits for the mappers.

Rather than using Hadoop, we wrote a small program to send inputs into the mappers and reducers and to receive their outputs. This implementation shortcut is valid, since Hadoop is considered untrusted and under attacker control in the VC3 security model. For simplicity, our program runs the mappers and reducers sequentially, storing the intermediate key-value pairs of each mapper on disk.

6.6 JPEG

We built a self-contained application around `libjpeg` that can be run inside an SGX enclave and that decodes an image. We had to provide a small runtime library to satisfy the external dependencies of `libjpeg` (memory allocation, file I/O). We reused the memory allocator from the VC3 runtime and provided just enough file I/O to read the input JPEG file from enclave RAM.

We compiled the unmodified source code of `libjpeg` version 9a, the runtime and the `main` function into a single Windows PE binary without external dependencies. We constructed an enclave consisting of this binary, an encrypted JPEG file (loaded into enclave memory) and heap memory. Upon invocation of the enclave, the `main` function decrypts the file in enclave memory, initializes the runtime and calls `libjpeg` to decompress the image.

cntr. increase	0	1	> 1
mean	64,393,418	204,040,806	1,231
CV	0.1	0.03	0.73

Figure 5: Single stepping experiment: interrupt count by observed counter increase. The mean is taken over 20 repetitions. CV is the coefficient of variation (mean divided by standard deviation).

7 Evaluation

We ran the experiments on a Windows 10 system based on an MSI Z170A motherboard with a 4.0 GHz quadcore Intel i7-6700K Skylake CPU, 8 GB of RAM and a 128 GB SanDisk X300 SSD. We disabled several unnecessary devices (DVD drive, audio, dedicated graphics card).

7.1 Single Stepping

We ran the following microbenchmark to evaluate the effectiveness of single stepping. We set up a victim application that increments an in-memory counter in a tight loop. The compiled loop code consists of three instructions: `add [rdi],1` (increment the counter in memory), `dec rax` (decrement the loop variable) and `jne -9` (conditional jump to the first instruction).

We shared the counter variable with the interrupt handler in our driver and made it record its value at each interrupt. This allowed us to observe the number of loop iterations that the victim had executed between consecutive interrupts. The driver also recorded the address of the interrupted instruction. Before returning, the interrupt handler restarted the timer and flushed the TLB. This experiment requires careful tuning. Space limitations force us to omit many details.

We ran the experiment until $2^{28} \approx 268$ million interrupts had occurred. We repeated the experiment 20 times for a total of more than 5 billion observations. The results are displayed in Fig. 5. About 24% of the interrupts occurred before the next counter increment. They waste cycles, but do not affect accuracy, as we can detect this case in real attacks. More than 99.9993% of the remaining interrupts break into the loop at consecutive iterations. This level of accuracy is more than sufficient for our attacks.

In most cases, the interrupt occurred directly after the `add [rdi],1` instruction. The delay due to the page-table walk for the memory operand (caused by the TLB flush) appears sufficient to absorb most of the timer jitter. We repeated the experiment, but invalidated only the application's TLB mapping to the counter instead of the entire TLB. This change did not have a significant im-

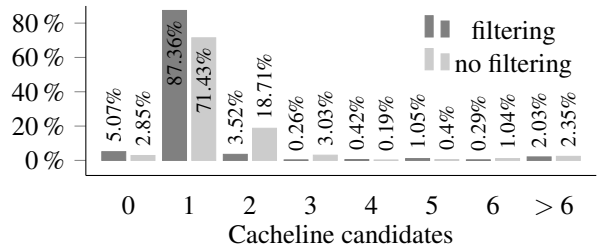

Figure 6: Accuracy of the cache observations of random array accesses with and without deterministic filtering.

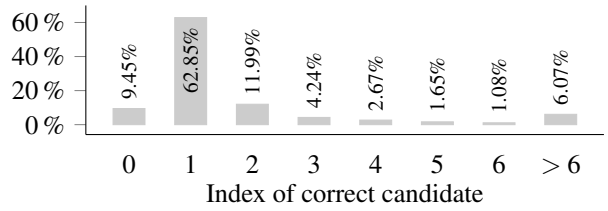

Figure 7: Position of the correct word in the ranked candidate list. Zero means the correct word was not in the candidate list

pact on the results, providing further evidence that the memory operand and its TLB mapping are the keys to the experiment.

7.2 Cache Side Channel

We evaluated the effectiveness of the cache side channel using a microbenchmark application that repeatedly accesses a 4 KB page-aligned array at random indices. We executed the application in an enclave and ran our driver to observe the cache lines of the array accesses.

Figure 6 shows the distribution of the number of cache line candidates returned by the attack over 35,000 accesses with and without applying deterministic filtering (Section 4.3). Both cases use noise filtering (Section 6.4).

With deterministic filtering, we obtained a unique cache line candidate in 87% of the observations, where 99.4% of these unique candidates identified the correct cache line. This level of accuracy is considerable, given that each observations is based on only a single memory access by the victim. The number also reveals the noticable presence of residual noise. As we will show next, this noise degrades the results of our attacks only moderately.

7.3 VC3

We used 20 English books from Project Gutenberg [2] and an English language model with 124,758 unigrams and 912,125 bigrams to evaluate our attack on VC3.

Effectiveness We ran our WordCount application on VC3 inside an enclave for each of the 20 books. We used our driver to perform the attack steps specified in Section 5. The driver produced a log file containing its observations, which we processed as described in Section 5.1.5.

Figure 8: A Sample of the text recovered by the attack: white background: 1st candidate; light-grey background: 2nd candidate; dark-grey background: 3rd candidate; black background: 4th or higher; solid black: word not in candidate list

We evaluated the accuracy of the resulting candidate lists by looking up where each word from the input document appeared in its candidate list. Figure 7 summarizes the results for one book [8] totaling 35,718 words. It shows the position of the correct word in each word's candidate list. For almost two thirds of the words (63%), the first candidate in the list is exactly the word in the document.

Figure 8 displays a sample of the recovered text. While recovery is not perfect, most of the words have been recovered uniquely or nearly uniquely. Overall, the content of the input text is revealed and comprehensible, in spite of our crude lanugage model which lacks explicit grammar rules. A better model is bound to help recover even more of the input.

Performance The end-to-end attacks slow down the applications' execution significantly. Following the approach of [44], our goal is to keep this slowdown at a level that can plausibly be explained by network and scheduling delays and various other cloud and internet glitches.

The runtime for our example book [8] incresases from 0.33 s to 123.5 s when deploying our attack, a 374x overhead similar to the previous attack [44]. The delay can be reduced by extracting only part of the document or by running several mappers and reducers in parallel. Word-length recovery (single stepping 560,128 times) and observing the hash slots (cache side channel) each take slightly less than half of the overhead (57 s and 54.7 s respectively), while 11.6 s are spent handling page faults.

7.4 JPEG

We used 20 images from Wikipedia [5] to evaluate the attack on `libjpeg`. Our test set included some of the images used in an earlier attack on `libjpeg` [44].

Effectiveness Figure 9 shows the result for an image from the earlier attack. The image recovered by our attack shows two expected artifact types. First, loss of detail due to cache-line granular observations. Second, noise,

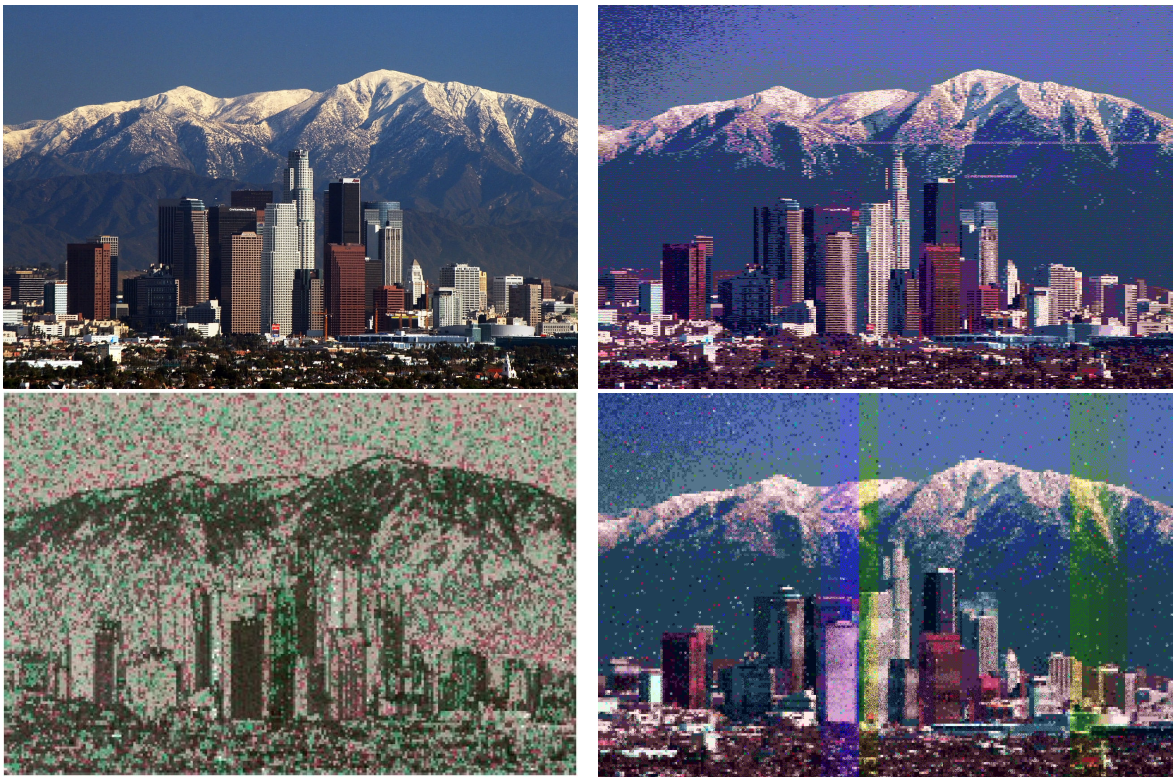

Figure 9: Example of a JPEG image recovered by the attack. top left: the original image; top right: recovered by the full attack; bottom left: recovered through the page fault channel [44, Figure 11]; bottom right: recovered by our attack, sampling 1 to 64.

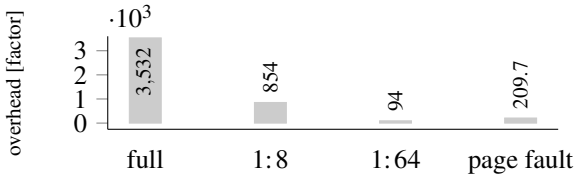

Figure 10: Overhead over baseline of various versions of the attack on libjpeg on the image in Figure 9.

resulting from incorrect cacheline observations. Despite these artifacts, the image recovered by our attack (Figure 9 top right) shows far more detail and looks much more like the original (Figure 9 top left) than the image recovered by the earlier attack (Figure 9 bottom left).[2]

Performance The attack incurs a substantial overhead due to the large number of prime-probe observations (up to three per pixel) and their relatively high cost. Our full attack on the image in Figure 9 incurs a 3,532x overhead (219 s vs. 62 ms), which is substantially higher than the 209.6x-354.9x for the page-fault channel [44, Fig.14].

However, the attacker can easily trade off overhead against accuracy by performing prime-probe observations only for a subset of the application's array lookups. We implemented this sampling strategy by allowing the attacker to specify how many values should be sampled in each 8x8 block. We repeated the attack sampling eight times per block and once per block, corresponding to 1:8 and 1:64 sampling ratios, and summarize the overheads in Figure 10. We show the median overhead for ten measurements. The standard deviation was less than 5% of the mean in all cases. The *page-fault* value is the 209.6x value reported for the previous page-fault channel attack [44, Fig.14].

The two bottom-row images in Figure 9 show clearly that even the image recovered at a 1:64 sampling ratio contains significantly more detail than the image recovered in [44], despite our significantly smaller overhead.

The overheads for the other images in our test set are similar to those of Figure 10, ranging from 2,595x to 3,532x for the full attack and 72x-94x for 1:64 sampling. The attack delays range from less than 10 s to about 4 min for the full attack, and 1.8 s-6 s for 1:64 sampling.

8 Mitigations

Cache side channels have been known for a long time, and a variety of defenses has been designed against them [15, 17, 19, 21, 32, 43, 47], working at the hardware, hypervisor, OS or compiler level. One approach is to partition caches so that the cache assigned to a sensitive

[2]Full resolution and additional images at tudos.org/~mhaehnel/SGX/

application cannot be accessed by a malicious program (e.g., [19, 32]). The other approach is to introduce noise so that a malicious program cannot tell if a cache miss is due to a real or random memory access (e.g., [21, 47]).

None of these defenses appears to be widely used or deployed, possibly due to their cost. In addition, traditional cache side channels have been targeting almost exclusively a small collection of cryptographic algorithms. These have been protected by eliminating all secret-dependent memory accesses from their implementations, thus obviating the need for more general defenses. However, attacks such as those presented in this paper demonstrate that a much broader class of code is potentially subject to cache side-channel attacks when the operating system is the adversary, and general defenses like those listed above may be required.

Shih *et al.* [41] propose a technique called T-SGX to disable side channels based on page-faults and interrupts. T-SGX is a compiler-based approach that automatically wraps computations in Intel TSX transactions. Since TSX [16] aborts transactions upon interrupts and exceptions, T-SGX can use the frequency of such aborts to detect side-channel attacks. T-SGX appears effective, but requires the application source code and incurs a noticeable overhead.

9 Related Work

Untrusted OSs Using feature-rich commodity OSs while removing them from applications' TCB has attracted considerable attention in research and industry. Hardware such as the Trusted Platform Module [3], Intel Trusted Execution Technology [22] or ARM TrustZone [4] as well as software hypervisors have long formed the basis for such systems. Some require applications to be specifically written for the new environment [20], others have the ambitious goal of securing legacy applications [14, 27].

Recently, the goal of protecting user applications in the cloud from the hosting provider's privileged software together with the introduction of Intel SGX have resulted in renewed activity in this area [6, 9, 26, 29, 34, 39].

Xu *et al.* [44] recognized the OS to be significantly more powerful than the traditional unprivileged attacker assumed by most side channel attacks. Their page-fault channel attack extracts complete text documents and outlines of JPEG images from a single run of the victim.

The channels presented in this paper offer much higher spatial and temporal resolution than the page-fault channel. This is significant because it shows that the collection of vulnerable application code is far broader than suggested by the page-fault channel.

Cache side channels Cache-based timing and trace-driven attacks are closely related to our work [11, 36, 37]. They generally assume an attacker with low privilege such as an unprivileged process [37], a virtual machine attacking its neighbors [38] or an attacker measuring server response time across the network [11]. Our attack is trace-driven, as the attacker can observe the victim's memory accesses. Trace-driven attacks typically reveal more fine-grained information than timing attacks. Most trace-based attacks are based on one of two techniques: prime-and-probe [36] and flush-and-reload [24].

The prime-and-probe technique has been used in synchronous [36] and asynchronous attacks [30, 33], targeting the L1 cache [37] and well as the LLC [30, 33]. This line of work typically assumes an unprivileged attacker who has little control over code running between the prime and the probe step. In contrast, control over scheduling and the ability to break into the victim at will let our attacker observe individual memory accesses at high time resolution. Several recent papers study cache side channels for enclaves using techniques that are different from ours and focusing primarily on crypto targets [12, 23, 40]. While preparing the camera-ready version of this paper, we became aware of recent, unpublished work that uses techniques similar to ours to attack crypto code [35].

Recently, the flush-and-reload method has enabled an array of stronger attacks [10, 25, 42, 45, 46]. Flush-and-reload can be used if the attacker and the victim share memory such as read-only code pages. While applicable in traditional cloud scenarios or with memory deduplication, our security model (and SGX) prevent such sharing. Furthermore our attacks are not limited to shared read-only code and data pages.

10 Conclusion

We have described two general techniques that can be combined to build high-resolution side channels for untrusted OSs, overcoming the main limitations of the page-fault channel.

This work shows that a much wider range of application code than suggested by the page-fault channel is subject to side-channel attacks. We demonstrate this with attacks against application code that cannot be attacked using only the page-fault channel. Whereas previous attacks have focused on unmodified legacy code, our main attack successfully targets an application that was designed not to trust the OS. This work highlights the increased importance of side-channels for privileged attacker scenarios. It takes us closer to understanding the full scope of the side-channel problem for untrusted OSs and highlights the need for robust mitigations.

Acknowledgments

We would like to thank our shepherd Nadav Amit and the anonymous reviewers for their valuable input.

References

[1] libjpeg. http://libjpeg.sourceforge.net/.

[2] Project Gutenberg. http://www.gutenberg.org/.

[3] Trusted Platform Module (TPM). http://www.trustedcomputinggroup.org/.

[4] TrustZone. http://www.arm.com/products/processors / technologies / trustzone / index.php.

[5] Wikipedia. http://www.wikipedia.org/.

[6] Ittai Anati, Shay Gueron, Simon Johnson, and Vincent Scarlata. Innovative technology for cpu based attestation and sealing. In *Workshop on Hardware and Architectural Support for Security and Privacy (HASP 2013)*, June 2013.

[7] Apache Software Foundation. Hadoop. http://wiki.apache.org/hadoop/, 2011.

[8] L Frank Baum. *The wonderful wizard of Oz*.

[9] Andrew Baumann, Marcus Peinado, and Galen Hunt. Shielding Applications from an Untrusted Cloud with Haven. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.

[10] Naomi Benger, Joop van de Pol, Nigel Smart, and Yuval Yarom. "Ooh aah... just a little bit": A small amount of side channel can go a long way. In *Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, 2014.

[11] Daniel J. Bernstein. Cache-timing attacks on AES. Available at: http://cr.yp.to/antiforgery/cachetiming-20050414.pdf, 2005.

[12] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. Software grand exposure: SGX cache attacks are practical. https://arxiv.org/abs/1702.07521, February 2017.

[13] Stephen Checkoway and Hovav Shacham. Iago attacks: Why the system call API is a bad untrusted RPC interface. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2013.

[14] Xiaoxin Chen, Tal Garfinkel, E. Christopher Lewis, Pratap Subrahmanyam, Carl A. Waldspurger, Dan Boneh, Jeffrey Dwoskin, and Dan R. K. Ports. Overshadow: A virtualization-based approach to retrofitting protection in commodity operating systems. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2008.

[15] Bart Coppens, Ingrid Verbauwhede, Koen De Bosschere, and Bjorn De Sutter. Practical mitigations for timing-based side-channel attacks on modern x86 processors. In *Proceedings of the 2009 IEEE Symposium on Security and Privacy*, pages 45–60, 2009.

[16] Intel Corporation. Intel 64 and ia-32 architectures software developer's manual.

[17] Stephen Crane, Andrei Homescu, Stefan Brunthaler, Per Larsen, and Michael Franz. Thwarting cache side-channel attacks through dynamic software diversity. In *Proceedings of the 2015 Network and Distributed System Security Symposium (NDSS 2015)*, 2015.

[18] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In *Proceedings of the 6th USENIX Symposium on Operating System Design and Implementation (OSDI'04)*, December 2004.

[19] Leonid Domnitser, Aamer Jaleel, Jason Loew, Nael Abu-Ghazaleh, and Dmitry Ponomarev. Non-monopolizable caches: Low-complexity mitigation of cache side channel attacks. *ACM Transactions on Architecture and Code Optimization*, 8(4):35:1–35:21, January 2012.

[20] Paul England, Butler Lampson, John Manferdelli, Marcus Peinado, and Bryan Willman. A trusted open platform. *Computer*, 36(7):55–62, 2003.

[21] Adi Fuchs and Ruby B. Lee. Disruptive Prefetching: Impact on Side-Channel Attacks and Cache Designs. In *Proceedings of the 8th ACM International Systems and Storage Conference*, May 2015.

[22] Matthew Gillespie. Intel trusted execution technology: A primer. https://software.intel.com/en-us/articles/intel-trusted-execution-technology-a-primer/.

[23] Johannes Götzfried, Moritz Eckert, Sebastian Schinzel, and Tilo Müller. Cache atacks on SGX.

In *2017 European Workshop on Systems Security (EuroSec'17)*.

[24] David Gullasch, Endre Bangerter, and Stephan Krenn. Cache Games – bringing access-based cache attacks on AES to practice. In *Proceedings of the 2011 IEEE Symposium on Security and Privacy*, pages 490 –505, May 2011.

[25] Berk Gülmezoglu, Mehmet Sinan Inci, Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. A faster and more realistic flush+reload attack on AES. In *Proceedings of Constructive Side-Channel Analysis and Secure Design COSADE (2015)*, 2015.

[26] Matthew Hoekstra, Reshma Lal, Pradeep Pappachan, Carlos Rozas, Vinay Phegade, and Juan del Cuvillo. Using innovative instructions to create trustworthy software solutions. In *Workshop on Hardware and Architectural Support for Security and Privacy (HASP 2013)*, June 2013.

[27] Owen S. Hofmann, Alan M. Dunn, Sangman Kim, Michael Z. Lee, and Emmett Witchel. InkTag: Secure applications on an untrusted operating system. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2013.

[28] Intel Corp. *Software Guard Extensions Programming Reference*, September 2013. Ref. #329298-001 `http://software.intel.com/sites/default/files/329298-001.pdf`.

[29] Intel Corp. *Software Guard Extensions Programming Reference*, October 2014. Ref. #329298-002US `https://software.intel.com/sites/default/files/managed/48/88/329298-002.pdf`.

[30] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. S$A: A shared cache attack that works across cores and defies vm sandboxingand its application to AES. In *Proceedings of the 36th IEEE Symposium on Security and Privacy*, 2015.

[31] Prerit Jain, Soham Desai, Seongmin Kim, Ming-Wei Shih, JaeHyuk Lee, Changho Choi, Youjung Shin, Taesoo Kim, Brent B. Kang, and Dongsu Han. Opensgx: An open platform for sgx research. In *Proceedings of the 2016 Network and Distributed System Security Symposium (NDSS 2016)*, 2016.

[32] Taesoo Kim, Marcus Peinado, and Gloria Mainar-Ruiz. STEALTHMEM: System-Level Protection Against Cache-Based Side Channel Attacks in the Cloud. In *USENIX Security Symposium*, 2012.

[33] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. Last-level cache side-channel attacks are practical. In *Proceedings of the 36th IEEE Symposium on Security and Privacy*, 2015.

[34] Frank Mckeen, Ilya Alexandrovich, Alex Berenzon, Carlos Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday Savagaonkar. Innovative instructions and software model for isolated execution. In *Workshop on Hardware and Architectural Support for Security and Privacy (HASP 2013)*, June 2013.

[35] Ahmad Moghimi, Gorka Irazoqui, and Thomas Eisenbarth. Cachezoom: How SGX amplifies the power of cache attacks. `https://arxiv.org/abs/1703.06986`, March 2017.

[36] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: the case of aes. In *Topics in Cryptology–CT-RSA 2006*, pages 1–20. Springer, 2006.

[37] Colin Percival. Cache missing for fun and profit. In *BSDCan 2005*, Ottawa, 2005.

[38] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds. In *ACM Conference on Computer and Communications Security (CCS)*, 2009.

[39] Felix Schuster, Manuel Costa, Cedric Fournet, Christos Gkantsidis, Marcus Peinado, Gloria Mainar-Ruiz, and Mark Russinovich. Vc3: Trustworthy data analytics in the cloud using sgx. In *36th IEEE Symposium on Security and Privacy*. IEEE Institute of Electrical and Electronics Engineers, May 2015.

[40] Michael Schwarz, Samuel Weiser, Daniel Gruss, Clémentine Maurice, and Stefan Mangard. Malware guard extension: Using SGX to conceal cache attacks. In *14th Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA'17)*, 2017.

[41] Ming-Wei Shih, Sangho Lee, Taesoo Kim, and Marcus Peinado. T-SGX: Eradicating Controlled-Channel Attacks Against Enclave Programs. In *Proceedings of the 2017 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2017.

[42] Joop van de Pol, Nigel Smart, and Yuval Yarom. Just a little bit more. In *CT-RSA*, 2015.

[43] Zhenghong Wang and Ruby B. Lee. New cache designs for thwarting software cache-based side channel attacks. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, ISCA '07, pages 494–505, 2007.

[44] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy*, 2015.

[45] Yuval Yarom and Katrina Falkner. FLUSH+RELOAD: A high resolution, low noise, L3 cache side-channel attack. In *USENIX Security Symposium*, 2014.

[46] Yinqian Zhang, Ari Juels, Michael Reiter, and Thomas Ristenpart. Cross-tenant side-channel attacks in PaaS clouds. In *ACM Conference on Computer and Communications Security (CCS)*, 2014.

[47] Yinqian Zhang and Michael K Reiter. Düppel: Retrofitting commodity operating systems to mitigate cache side channels in the cloud. In *ACM Conference on Computer and Communications Security (CCS)*, 2013.