Varys: Protecting SGX enclaves from practical side-channel attacks

Oleksii Oleksenko, Bohdan Trach, Robert Krahn, and André Martin, *TU Dresden;* Mark Silberstein, *Technion;* Christof Fetzer, *TU Dresden*

https://www.usenix.org/conference/atc18/presentation/oleksenko

This paper is included in the Proceedings of the
2018 USENIX Annual Technical Conference (USENIX ATC '18).

July 11–13, 2018 • Boston, MA, USA

ISBN 978-1-939133-02-1

Varys

Protecting SGX Enclaves From Practical Side-Channel Attacks

Oleksii Oleksenko[†], Bohdan Trach[†], Robert Krahn[†], Andre Martin[†],
Christof Fetzer[†], Mark Silberstein[‡]
[†]*TU Dresden*, [‡]*Technion*

Abstract

Numerous recent works have experimentally shown that Intel Software Guard Extensions (SGX) are vulnerable to cache timing and page table side-channel attacks which could be used to circumvent the data confidentiality guarantees provided by SGX. Existing mechanisms that protect against these attacks either incur high execution costs, are ineffective against certain attack variants, or require significant code modifications.

We present **Varys**, a system that protects unmodified programs running in SGX enclaves from cache timing and page table side-channel attacks. Varys takes a pragmatic approach of strict reservation of physical cores to security-sensitive threads, thereby preventing the attacker from accessing shared CPU resources during enclave execution. The key challenge that we are addressing is that of maintaining the core reservation in the presence of an untrusted OS.

Varys fully protects against all L1/L2 cache timing attacks and significantly raises the bar for page table side-channel attacks—all with only 15% overhead on average for Phoenix and PARSEC benchmarks. Additionally, we propose a set of minor hardware extensions that hold the potential to extend Varys' security guarantees to L3 cache and further improve its performance.

1 Introduction

Intel Software Guard Extensions (SGX) enclaves provide a shielded environment to securely execute sensitive programs on commodity CPUs in the presence of a privileged adversary. So far, no successful direct attack on SGX has been reported, i.e., none that compromises SGX's security guarantees. However, numerous works demonstrate that SGX is vulnerable to several types of side channel attacks (SCAs), in particular, traditional cache timing and page table SCA that reveal page-level memory accesses [9, 39, 21, 47, 43, 45, 24], as well as speculative attacks [29, 12] that use the side channels as a way of retrieving information. Although Intel explicitly excludes side channels from the SGX threat model, SCAs effectively circumvent the SGX confidentiality guarantees and impede SGX adoption in many real-world scenarios. More crucially, a privileged adversary against SGX can mount much more powerful SCAs compared to the unprivileged one in canonical variants of the attacks. For example, a malicious OS can dramatically reduce the noise levels in cache timing attacks via single-stepping [24] or by slowing down the victim.

In this paper, we investigate *practical* ways of protecting SGX programs from page table and cache timing SCAs. Specifically, we focus on the case where unmodified general-purpose applications are executed in enclaves in order to protect their secrets, as it is the case with Haven [4], Graphene-SGX [11], or SCONE [3].

We postulate that a *practical* solution should have low performance overheads, require no modifications to application source code, and impose no restrictions on the application's functionality (such as restricting multithreading). We assume that recompilation of source code is acceptable as long as it does not require code changes.

Existing mitigation techniques, however, fall short of satisfying our requirements. Traditional hardening techniques against cache timing attacks [5, 23] require rewriting the application; recent defenses [22] based on Intel TSX technology also require code changes; memory accesses obfuscation approaches, such as DR.SGX [8], incur high performance cost (at least 3× and up to 20×); T-SGX [40] prevents controlled OS attacks, but it is ineffective against concurrent attacks on page tables and caches. Déjà Vu [14] protects only against page table attacks and is prone to false positives.

In *Varys*, we strive to achieve both application performance and user convenience while making page table and cache timing SCAs on enclaves much harder or entirely impossible to mount. The basic idea behind Varys design is *trust but verify*. A potentially malicious OS is *requested* to execute the enclave in a *protected environment* that prevents all known cache timing and page fault attacks on

SGX enclaves. However, the Varys trusted runtime inside the enclave verifies that the OS fulfills the request.

Our main observation is that all published page table and L1/L2 cache timing attacks on SGX require either (1) a high rate of enclave exits, or (2) control of a sibling hyperthread on the same CPU core with the victim. Consequently, if an enclave is guarded against frequent asynchronous exits and executes on a dedicated CPU core without sharing it with untrusted threads, it would be protected against the attacks. The primary challenge that Varys addresses is in maintaining such a protected environment in face of a potentially malicious OS. It achieves this goal via two mechanisms: *asynchronous enclave exits monitoring* and *trusted reservation*.

First, Varys monitors when asynchronous enclave exits (AEX) occur (e.g., for scheduling another process on the core or handling an exception) and restricts the frequency of such exits, terminating the enclave once the AEX frequency bound is exceeded. Varys sets the bound to the values that render all known attacks impossible. Notably, the bound is much higher than the frequency of exits in an attack-free execution, thereby minimizing the chances of false positives as we explain in §4.

Second, Varys includes a mechanism for *trusted core reservation* such that the attacker cannot access the core resources shared with the enclave threads while they are running, nor can it recover any secrets from the core's L1 and L2 caches afterward. For example, consider an in-enclave execution of a multi-threaded application with two threads. Assuming a standard processor with hyperthreading (SMT), all it takes to prevent concurrent attacks on L1/L2 caches is to guarantee that the two enclave threads always run *together* on the same physical core. As a result, the threads occupy both hardware threads of the core, and the attacker cannot access the core's caches. Note that this simple idea prevents any concurrent attacks on core's resources shared between its hyperthreads, such as branch predictor and floating point unit. It also prevents exit-less SCAs on page table attributes [43] because they require attacker's access to the core's TLB—available only if the attacker thread is running on that core. Additionally, to ensure that the victim leaves no traces in the caches when de-scheduled from the core, Varys explicitly evicts the caches when enclave threads are preempted.

While conceptually simple, the implementation of the trusted reservation mechanism is a significant challenge. An untrusted OS may ignore the request to pin two enclave threads to the same physical core, may re-enable CPU hyperthreading if disabled prior to enclave execution, and may preempt each of the enclave threads separately in an attempt to break Varys's defense.

Our design offers *a low-overhead mechanism for trusted core reservation under an untrusted OS*. Application threads are grouped in pairs, and the OS is *requested* to co-locate these pairs on the same physical CPU core. The trusted application threads are instrumented (via a compiler pass) to *periodically verify* that they are indeed co-scheduled and running together on the same core. Varys terminates the enclave if co-scheduling is violated or if any of the threads in the pair gets preempted too often. To reduce the frequency of legitimate exits and lower the false positives, Varys uses exitless system calls [3] and in-enclave thread scheduling such that multiple application threads can share the same OS thread. Moreover, Varys configures the OS to reduce the frequency of interrupts routed to the core in order to avoid interference with attack-free program execution. However, if the OS ignores the request, this will effectively lead to denial of service without compromising the enclave's security.

Varys primarily aims to protect multi-threaded programs by reserving complete cores and scheduling the application threads on them, i.e., protection against SCAs translates into an allocation policy that allocates or frees computing resource with a granularity of one core. We believe that Varys exercises a reasonable trade-off between security and throughput for services that require the computational power of one or more cores. For single-threaded applications Varys pairs the application thread with a service thread to reserve the complete core.

Due to the lack of appropriate hardware support in today's SGX hardware, Varys remains vulnerable to timing attacks on Last Level Cache (LLC) as we explain in §8. We suggest a few minor hardware modifications that hold the potential to solve this limitation and additionally, eliminate most of the runtime overhead. These extensions allow the operating system to stay in control of resource allocations but permit an enclave to determine if its resource allocation has changed.

Our contributions include:

- Analysis of attack requirements.
- A set of measures that can be taken to protect against these attacks using existing OS features.
- Varys, an approach to verifying that the OS correctly serves our request for a protected environment.
- Implementation of Varys with 15% overhead across PARSEC [7] and Phoenix [38] benchmark suites.
- Proposal for hardware extensions that improve Varys's security guarantees and performance.

2 Background

2.1 Intel SGX

Intel Software Guard Extensions is an ISA extension that adds hardware support for Trusted Execution Environments. SGX offers creation of *enclaves*—isolated memory regions, with code running in a novel enclave execution mode. Also, Intel SGX provides a local and remote attestation systems that is used to establish whether the software is running on an SGX-capable CPU.

SGX has hardware-protected memory called Enclave Page Cache (EPC). Accesses to the EPC go through the Memory Encryption Engine (MEE), which transparently encrypts and applies MAC to cache lines on writes, and decrypts and verifies cache lines on reads. Access permissions of the pages inside an enclave are specified both in page tables and in EPC Metadata, and permissions can be only restricted via page tables. The enclave's memory contents in the caches are stored as plaintext.

Enclaves can use more virtual memory than can be stored by the EPC. In this case, EPC paging will happen when a page not backed by physical memory is accessed. The CPU, in coordination with the OS kernel, can evict a page to untrusted memory. Currently, the EPC size available to user applications is around 94 MB.

2.2 Side-channel attacks

In this work, we focus mainly on cache timing and page table attacks as they provide the widest channel and thus, are the most practical to be used to attack enclaves.

Cache timing attacks [36, 32, 27, 25, 2, 48] infer the memory contents or a control flow of a program by learning which memory locations are accessed at fine granularity. The adversary uses the changes in the state of a shared cache as a source of information. In particular, she sets the cache to a predefined state, lets the victim interact with the cache for some time, and then reads from the cache again to determine which parts were used by the victim. Accordingly, for this attack to work, the adversary has to be able to access the victim's cache.

Page table attacks reveal page-level access patterns of a program. They are usually considered in the context of trusted execution environments as they are possible only if privileged software is compromised, hence they are also called *controlled-channel attacks* [47].

These attacks can be classified into *page-fault based* and *page-bit based*. Page-fault based attacks [47, 41] intercept all page-level accesses in the enclave by evicting the physical pages from the EPC. Page-bit based attacks [45, 43] use the *Accessed* and *Dirty* page table bits as an indication of access to the page, without page faults. However, these bits are cached in a TLB, so to achieve the required fidelity, the adversary has to do both, i.e., to clear the flags and to flush the victim's TLB.

3 Threat Model

We assume the standard SGX threat model. The adversary is in complete control of privileged software, in particular, the hypervisor and the operating system. She can spawn and stop processes at any point, set their affinity and modify it at runtime, arbitrarily manipulate the interrupt frequency and cause page faults. The adversary can also read and write to any memory region except the enclave memory, map any virtual page to any physical one and dynamically re-map pages. Together, it creates lab-like conditions for an attack: it could be running in a virtually noise-free environment.

4 System footprint of SGX SCAs

In this section we analyze the runtime conditions required for the known SCAs to be successful. Varys mitigates the SCAs by executing an enclave in a protected environment and preventing these conditions from occurring.

Cache attacks [36, 42, 32, 2, 48, 50, 10, 17] can be classified into either *concurrent*, i.e, running in parallel with the victim, or *time-sliced*, i.e., time-sharing the core (or a hyperthread) with the victim.

Time-sliced cache attacks $\Longrightarrow$ high AEX rate. For a time-sliced attack to be successful, the runtime of the victim in each time slice must be short; otherwise, the cache noise will become too high to derive any meaningful information. For example, the attack described by Zhang et al. [49] can be prevented by enforcing minimal runtime of 100us [44], which translates into 10kHz interrupt rate. It is dramatically higher than the preemption rate under normal conditions—below 100Hz (see §5.3). If the victim is an enclave thread, its preemption implies an asynchronous enclave exit (AEX).

Concurrent cache attacks $\Longrightarrow$ shared core. The adversary running in parallel with the victim must be able to access the cache level shared with it. Thus, L1/L2 cache attacks are not possible unless the adversary controls a sibling hyperthread.

We note that with the availability of the Cache Allocation Technology (CAT) [26], the share of the Last Level Cache (LLC) can also be allocated to a hardware thread, preventing any kind of concurrent LLC attacks [31]. However, this defense is ineffective for SGX because the allocation is controlled by an untrusted OS. We suggest one possible solution to this problem in §8.

Page-fault page table attacks $\Longrightarrow$ high AEX rate. These attacks inherently increase the page fault rate, and consequently AEX rate, as they induce page faults to infer the accessed addresses. For example, as reported by Wang et al. [45], a page table attack on EdDSA requires approximately 11000 exits per second. In fact, high exit rates have been already used as an attack indicator [40, 22].

Interrupt-driven page-bit attacks $\Longrightarrow$ high AEX rate. If the attacker does not share a physical core with the victim, these attacks incur a high exit rate because the attacker must flush the TLB on a remote core via Inter-Processor Interrupts (IPIs). The rate is cited to be around 5500Hz [43, 45]. While lower than other attacks, it is still above 100Hz experienced in attack-free execution (§5.3).

Exit-less page-bit attacks $\Longrightarrow$ shared core. The only way to force TLB flushes without IPIs is by running an adversary sibling hyperthread on the same physical core to force evictions from the shared TLB [45]. These attacks involve no enclave exits, thus are called *silent*.

In summary, all the known page table and L1/L2 cache timing SCAs on SGX rely on (*i*) an abnormally high rate of asynchronous enclave exit, or/and (*ii*) an adversary-controlled sibling hyperthread on the same physical core. The only exception is the case when the victim has a slowly-changing working set, which we discuss in §7.2. These observations drive the design of the Varys system we present next.

5 Design

Varys provides a *side-channel protected execution environment* for SGX enclaves. This execution environment ensures that neither time-sliced nor concurrent cache timing as well as page table attacks can succeed. To establish such an environment, we (*i*) introduce a trusted reservation mechanism, (*ii*) combine it with a mechanism for monitoring enclave exits, and (*iii*) present a set of techniques for reducing the exit rate in an attack-free execution to avoid false positives.

5.1 Trusted reservation

The simplest way to ensure that an adversarial hyperthread cannot perform concurrent attacks on the same physical core would be to disable hyperthreading [33]. However, doing so not only hampers application performance but may not be reliably verified by the enclave: One can neither trust the operating system information nor can one execute the `CPUID` instruction inside of enclaves.

An alternative approach is to allow sharing of a core only by benign threads. Considering that in our threat model only the enclave is trusted, we allow core sharing only among the threads from the same enclave. We can achieve this goal by dividing the application threads in pairs (if the number of threads is odd, we spawn a dummy thread) and requesting the OS to schedule the pairs on the same cores. Since we do not trust the OS, we ensure collocation by establishing a covert channel in the L1 cache as follows.

The idea is to determine whether the threads share L1 cache or only last level cache. The later would imply the threads are on different physical cores. We refer to the procedure that determines the threads co-location on the core as *handshake*. To perform the handshake, we establish a simple covert channel between the threads via L1: One of the two threads writes a dummy value to a shared memory location, thus, forcing it to L1. Then, the sibling thread reads the same memory location and measures the timing. If the reading is fast (up to 10 cycles per read), both threads use the same L1 cache, otherwise (more than 40 cycles) they share only LLC, implying they are on different cores. If the threads indeed run on different cores, the OS did not satisfy the scheduling request made by Varys. We conclude that the enclave is under attack and terminate it. Since the current version of SGX does not provide trusted fine-grain time source, we implement our own as we explain in §6.2.

Of course, immediately after we established that the two threads are executing on the same core, the operating system could reschedule these threads on different cores. However, this rescheduling would cause an asynchronous enclave exit (AEX), which we detect via AEX monitoring as we discuss next.

5.2 AEX monitoring

To detect an asynchronous enclave exit, we monitor the SGX State Save Area (SSA). The SSA is used to store the execution state of an enclave upon an AEX. Since some parts of the enclave execution state are deterministic, we can detect an AEX by overwriting one of the SSA fields with an invalid value and monitoring it for changes.

For example, in the current implementation of SGX, the `EXIT_TYPE` field of an SSA frame is restricted to values `011b` and `110b` [26]. Thus, if we write `000b` to `EXIT_TYPE`, SGX will overwrite it with another, predefined value at the next AEX. To detect an AEX, we periodically read and compare this field with the predefined value. Note that it is not the only SSA field that we could use for this purpose; many other registers, such as Program Counter, could be used too.

Now that we have a detection mechanism, it is sufficient to count the AEX events and abort the application if they are too frequent. Yet, to calculate the frequency, we need a trusted time source which is not available inside an enclave. Fortunately, precise timing is not necessary for this particular purpose as we would only use the time to estimate the number of instructions executed between AEXs. It is possible to estimate it through the AEX monitoring routine that our compiler pass adds to the application. Since it adds the routine every few hundred LLVM IR instructions, counting the number of times it is called serves a natural counter of LLVM IR instructions. In Varys, we define the AEX rate as number of AEXs per number of executed IR instructions.

Even though IR instructions do not correspond to machine instructions, one IR instruction maps on average to less than one x86-64 machine instruction[1]. Thus, we overestimate the AEX rate, which is safe from the security perspective.

Originally, we considered using TSX (Transactional Synchronization Extensions) to detect AEXs—similar to the approach proposed by Gruss et al. [22]. The main limitation of TSX is, however, that it does not permit

[1] In our experience with Phoenix and PARSEC benchmark suites, calling the monitoring routine every 100 IR instructions resulted in the polling period of 70–150 cycles.

non-transactional memory accesses within transactions. Hence, a) handshaking is not possible within a TSX transaction—this would lead to a transaction abort, and b) the maximum transaction length is limited and we would need to split executions in multiple transactions.

5.3 Restricting Enclave Exit Frequency

Ensuring that protected applications exit as rarely as possible is imperative for our approach. If the application has a high exit rate under normal conditions, not only does it increase the overhead of the protection, but also makes it harder to distinguish an attack from the attack-free execution. In the worst case, if the application's normal exit rate is sufficiently high (i.e., more than 5500 exits/second, see below), the adversary does not have to introduce any additional exits and can abuse the existing ones to retrieve information. Therefore, we have to analyze the sources of exits and the ways of eliminating or reducing them.

Under SGX, an application may exit the enclave for one of the following reasons: when the application needs to invoke a system call; to handle system timer interrupts, with up to 1000 AEX/s, depending on the kernel configuration; to handle other interrupts, which could happen especially frequently if Varys runs with a noisy neighbor (e.g., a web server); to perform EPC paging when the memory footprint exceeds the EPC size; to handle minor page faults, which could happen frequently if the application works with large files.

We strive to reduce the number of exits as follows. We use asynchronous exit-less system calls implemented, for example, in Eleos [35] and SCONE [3] (which we use in our implementation). Further, we combine asynchronous system calls with user-level thread scheduling inside the enclave to avoid reliance on the OS scheduling. We avoid the timer interrupt by setting the timer frequency to the lowest available —100 Hz—and enabling the DynTicks feature. Regular interrupts are re-routed to non-enclave cores. Last, we prevent minor page faults when accessing untrusted memory via `MAP_POPULATE` flag to `mmap` calls.

To evaluate the overall impact of these changes, we measure the exit frequencies of the applications used in our evaluation (see §7 for the benchmarks' description). The results are shown in Figure 1.

As we see, the rate is (i) relatively stable across the benchmarks and (ii) much lower than the potential attack rate of more than 1000 exits per second. Specifically, the attack presented by Van Bulck et al. [43] has one of the lowest interrupt rates among the published time-sliced attacks. We ran the open-sourced version of the attack and observed the rate of at least 5500 exits per second, which is in line with the rate presented in the paper. Correspondingly, if we detect (see §6.2) that the AEX rate is getting above 100 Hz, we can consider it a potential attack and take appropriate measures. To avoid false positives, we could set the threshold even higher—around 2kHz—without compromising security (see §7.2).

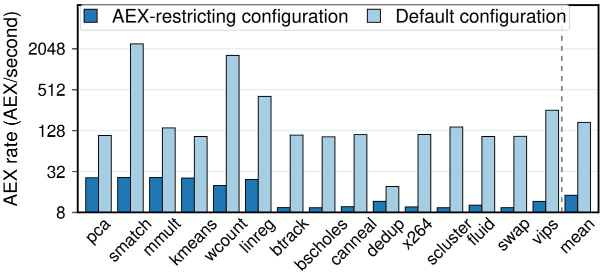


Figure 1: AEX rates under normal system configuration and with re-configured system.

```
while(true):
  wait_for_request()
  if (secret == 0): response = *a
  else: response = *b
```

Figure 2: An example of code leaking information in cache side-channel even with low frequency of enclave exits. If a and b are on different cache lines and the requests are coming infrequently, it is sufficient to probe the cache at the default frequency of OS timer interrupts.

5.4 Removing residual cache leakage

As we explained in §4, even with low frequency of enclave exits some leakage will persist if the victim has a slowly changing working set. Consider the example in Figure 2: the replies to user requests depend on the value of a secret. If requests arrive infrequently (e.g., 1 per second), restricting the exit frequency would not be sufficient; even if we set the bar as low as 10 exits per second (the rate we achieved in §5.3), the victim will touch only one cache line and thus, will reveal the secret.

To completely remove the leakage at AEX, we should flush the cache before we exit the enclave. This would remove any residual cache traces that an adversary could use to learn whether the enclave has accessed certain cache lines. Unfortunately, this operation is not available at user-space on Intel CPUs [26] nor do we have the possibility to request a cache flush at each AEX. Moreover, Ge et al. [18] have proven that the kernel-space flush commands do not flush the caches completely. `CLFUSH` instruction does not help either as it flushes a memory address, not the whole cache set. Thus, it cannot flush the adversary's eviction set residing in a different virtual address space, as it is the case in Prime+Probe attacks.

Instead, on each enclave entry, we write a dummy value to all cache lines in a continuous cache-sized memory region (e.g., 32KB for L1), further called *eviction region*. In case of L1, for which instruction and data are disjoint, we also execute a 32KB dummy piece of code to evict the instruction cache. This way, regardless of what the victim

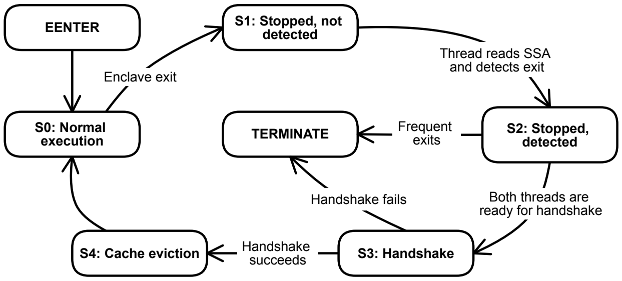


Figure 3: State diagram of a Varys-protected application.

does in between the exits, external observer will see that all the cache sets and all the cache ways were accessed and no information will be leaked.

6 Implementation

We implement Varys as an LLVM compiler pass that inserts periodic calls to a runtime library. We use SCONE to provide us with asynchronous system calls as well as in-enclave threading such that we minimize the need for an application to exit the enclave.

6.1 LLVM compiler pass

The cornerstone of Varys is the enclave exit detection. As discussed in §5.2, it requires all application threads to periodically poll the SSA region. Although we implement the checks as a part of a runtime library (§6.2), calls to the library have to be inserted directly into the application. To do this, we instrument the application using LLVM [30].

The goal of the instrumentation pass is to call the library and do the SSA polling with a predictable and configurable frequency. We achieve it by inserting the following sequence before every basic block: We increment a counter by the length of the basic block (in LLVM IR instructions), call the library if the counter reached a threshold, and skip the call otherwise. If the basic block is longer than the threshold, we add several calls. This way, the checks will be performed each time the application executes a given (configurable) number of IR instructions. We also reserve one of the CPU registers for the counter, as it is manipulated every few instructions and having the counter in memory would cause much higher overheads.

A drawback of SSA polling is that it has a blind zone. If a malicious OS preempts a thread multiple times in a very short period of time, they may happen before the counter reaches the threshold and the thread checks the SSA value. Hence, they will be all counted as a single enclave exit. This allows an adversary to launch stealthy cache attacks on small pieces of code by issuing occasional series of frequent preemptions. Yet, this vulnerability would be hard to exploit because the blind zone is narrow—on the order of dozens of cycles, depending on the configuration—and the adversary must run in tight synchronization with the victim to retrieve any meaningful information.

Optimization. Adding even a small piece of code to every basic block could be expensive as the blocks themselves are often only 4–5 instructions long. We try to avoid this by applying the following optimization.

Consider a basic block B0 with two successors, B1 and B2. In a naive version, in the beginning of each basic block we increment the IR instruction counter by the length of the corresponding basic block. However, if B0 cannot jump into itself, it will always proceed to a successor. Therefore, it is sufficient to increment the counters only in the beginnings of B1 and B2 by, accordingly, `length(B0)+length(B1)` and `length(B0)+length(B2)`. If B1 or B2 have more than one predecessor, it could lead to overestimation and more frequent SSA polling, which only reduces the blind zone.

6.2 Runtime library

Most of Varys' functionality is contained in a runtime library implementing the state machine in Figure 3.

When a program starts, it begins *normal execution* (S0). As long as the program is in this state, it counts executed instructions thus simulating a timer.

When one of the threads is interrupted, the CPU executes an AEX and overwrites the corresponding SSA (S1). As its sibling thread periodically polls the SSA, it eventually detects the exit. Then, if the program has managed to make sufficient progress since the last AEX (i.e., if the IR instruction counter has a large enough value), it transfers to the *detected* state (S2). Otherwise, the program terminates. To avoid false positives, we could terminate the program only if it happens several times in a row.

In S2, the sibling declares that the handshake is pending and starts busy-waiting. When the first thread resumes, it detects the pending handshake, and the pair enters state S3. If the handshake fails, the program is terminated[1]. Otherwise, one of the threads evicts L1 and L2 caches, and the pair continues normal execution.

Software timer. To perform cache measurements during the handshake phase, we need a trusted fine-grained source of time. Since the hardware time counter is not available in the current version of SGX, we implement it in software (similar to Schwarz et al. [39]). We spawn an enclave thread incrementing a global variable in a tight loop, giving us approximately one tick per cycle.

However, the frequency of the software timer is not reliable. An adversary can abuse the power management features of modern Intel CPUs and reduce the timer tick frequency by reducing the operational frequency of the underlying core. If the timer becomes slow enough, the handshake will be always succeeding. To protect against

[1] In practice, timing measurements are noisy and the handshake may fail for benign reasons. Therefore, we retry it several times and consider it failed only if the timing is persistently high.

```
.align 64
label1: jump label2 // jump to the next cache line
.align 64
label2: jump label3
```

Figure 4: A code snippet evicting cache lines in the L1 instruction cache. For evicting a 32 KB cache, the pattern is repeated 512 times.

it, we measure the timing of a constant-time operation (e.g., a series of in-register additions). Then, we execute the handshake only if the measurement matches the expected value.

Instruction cache eviction. Writing to a large memory region is not sufficient for evicting L1 or L2 caches. L1 has distinct caches for data (L1d) and instructions (L1i), and L2 is non-inclusive, which means that evicting L2 does not imply evicting L1i. Hence, the attacks targeting execution path are still possible.

To evict L1i, we have to execute a large piece of code. The fastest way of doing so is depicted in Figure 4. The code goes over a 32 KB region and executes a jump for each cache line thus forcing it into L1i.

L2 cache eviction. Evicting L2 cache is not as straightforward as L1 as it is physically-indexed physically-tagged (PIPT) [46]. For the L2 cache, allocating and iterating over a continuous virtual memory region does not imply access to continuous physical memory, and therefore does not guarantee cache eviction. A malicious OS could apply cache colouring [6, 28] to allocate physical pages in a way that the vulnerable memory locations map to one part of the cache and the rest of the address space—to another. This way, the vulnerable cache sets would not be evicted, and the leakage would persist.

With L2 cache, we do two passes over the eviction region. The first time, we read the region to evict the L2 cache. The second time, we read and measure the timing of this read. If the region is continuous, the first read completely fills the cache and the second read should be relatively fast as all the data is in the cache. However, if it is not the case, some pages of the eviction region would be competing for cache lines and evicting each other, thus making the second read slower. We use this as an indicator that L2 eviction is not reliable and we should try to allocate another region. If the OS keeps serving us non-continuous memory, we terminate the application as the execution cannot be considered reliable anymore.

6.3 SCONE

We base our implementation on SCONE [3], a shielding framework for running unmodified application inside SGX enclaves. Among other benefits, SCONE provides two features that make our implementation more efficient and compact. First, it implements user-level threading, which significantly simplifies thread pairing. As the number of enclave threads is independent of the number of application threads and fixed, it suffices to allocate and initialize thread pairs at program startup. Second, it provides asynchronous system calls. They not only significantly reduce the rate of enclave exits but also make this rate more predictable and application agnostic.

We should note, that Varys is not conceptually linked to SCONE. We could have avoided user-level threading by modifying the standard library to dynamically assign thread pairs. The synchronous system calls are also not an obstacle, but they require a mechanism to distinguish different kinds of enclave exits.

7 Evaluation

In this section, we measure the performance impact of Varys, the efficiency of attack detection and prevention, as well as the rate of false positives.

Applications. We base our evaluation on the Fex [34] evaluation framework, with PARSEC [7] and Phoenix [38] benchmark suites as workloads. The following benchmarks were excluded: *raytrace* depends on the dynamic X Window System libraries not shipped together with the benchmark; *freqmine* is based on OpenMP; *facesim* and *ferret* fail to compile under SCONE due to position-independent code issues. Together with the benchmarks, we recompile and instrument all the libraries they depend upon. We also manually instrument the most frequently used *libc* functions so that at least 90% of the execution time is spend in a protected code. We used the largest inputs that do not cause intensive EPC paging as otherwise, they could lead to frequent false positives.

Methodology. All overheads were calculated over the native SGX versions build with SCONE. The reported results are averaged over 10 runs and the "mean" value is a geomean across all the benchmarks.

Testbed. We ran all the experiments on a 4-core (8 hyperthreads) Intel Xeon CPU operating at 3.6 GHz (Skylake microarchitecture) with 32 KB L1 and 256 KB L2 private caches, an 8 MB L3 shared cache, 64 GB of RAM, and a 1TB SATA-based SSD. The machine was running Linux kernel 4.14. To reduce the rate of enclave exits, we configure the system as discussed in §5.3.

7.1 Performance Evaluation

Runtime. Figure 5 presents runtime overheads of different Varys security features. On average, the overhead is ~15%, but it varies significantly among benchmarks.

A major part of the overhead comes from the AEX detection, which we implement as a compiler pass. Since the instrumentation adds instructions that are not data dependent on the application's data flow, they can run in parallel. Therefore, they highly benefit from instruction

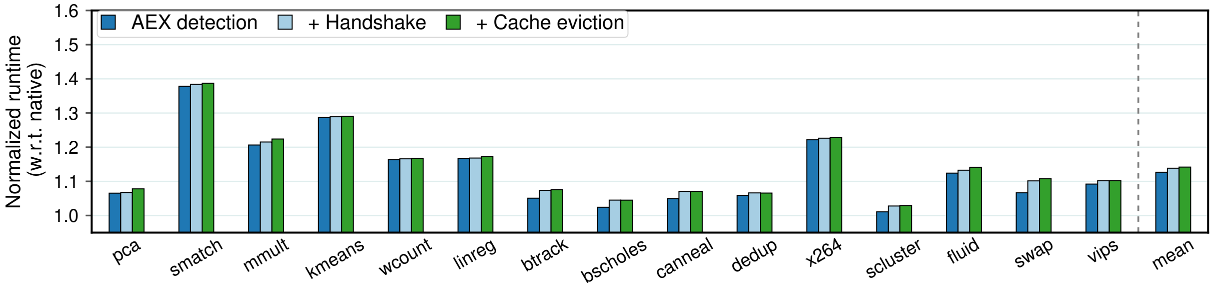


Figure 5: Performance impact of Varys security features with respect to native SGX version. Each next bar includes all the previous features. (Lower is better.)

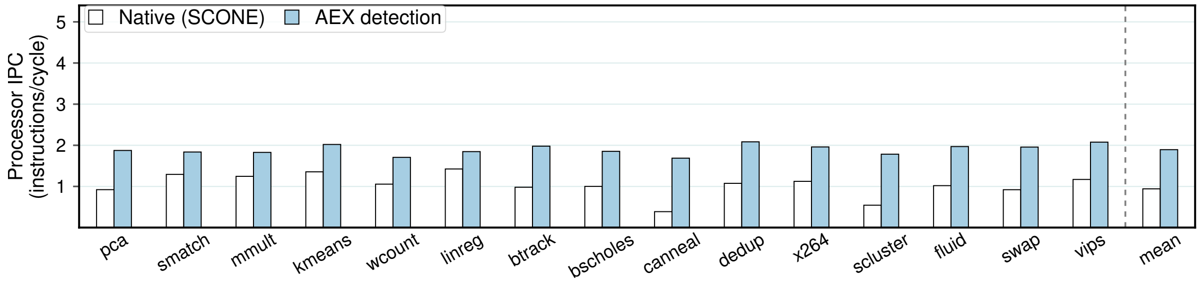


Figure 6: IPC (instructions/cycle) numbers for native and protected versions.

level parallelism (ILP), which we illustrate with Figure 6. The applications that have lower ILP utilization in the native version (e.g., *canneal* and *stream cluster*) can run a larger part of the instrumentation in parallel, thus amortizing the overhead.

Since we apply instrumentation per basic block, another factor that influences the overhead is the average size of basic blocks. The applications dominated by long sequences of arithmetic operations (e.g., *linear regression*) tend to have longer basic blocks and lower number of additional instructions (53% in this case), hence the lower overhead. At the same time, the applications with tight loops on the hot path cause higher overhead. Therefore, *string match* has higher overhead than *kmeans*, even though they have approximately the same level of IPC.

The second source of overhead is trusted reservation. It does not cause a significant slowdown because the handshake protocol is relatively small, including ten memory accesses for the covert channel and the surrounding code for the measurement. The overhead could be higher as the headshake is synchronized, i.e., two threads in a pair can make progress only if both are running. Otherwise, if one thread is descheduled, the second one has to stop and wait. Yet, as we see in Figure 5, it happens infrequently.

Finally, cache eviction involves writing to a 256 KB data region and executing a 32 KB code block. Due to the pseudo-LRU eviction policy of Intel caches, we have to repeat the writing several times (three, in our case). Together, it takes dozens of microseconds to execute, depending on the number of cache misses. Fortunately, we evict only after enclave exits, which are infrequent under normal conditions (§5.3) and the overhead is low.

Multithreading. As Varys is primarily targeted at multithreaded applications, it is crucial to understand its impact on multithreaded performance. To evaluate this parameter, we measured the execution time of all benchmarks with 2, 4, and 8 threads with respect to native versions with the same number of threads. Mind that these are user-level threads; the number of underlying enclave threads is always 4. The results are presented in Figure 7.

Generally, Varys does not have a significant impact on multithreaded scaling. However, there are a few exceptions. First, larger memory consumption required for multithreading causes EPC paging, thus increasing the AEX rate and sometimes even causing false positives. We can see this effect in *dedup* and *x264*: the higher AEX rate makes the flushing more expensive and eventually leads to false positives with higher numbers of threads. For the same reason, we excluded *linear regression*, *string match*, and *word count* from the experiment.

Another interesting effect happens in multithreaded *kmeans*. The implementation of *kmeans* that we use frequently creates and joins threads. Internally, `pthread_join` invokes memory unmapping, which in turn causes a TLB flush and an enclave exit. Correspondingly, the more threads *kmeans* uses, the more AEXs appear and the higher is the overhead.

Case Study: Nginx. To illustrate the impact of Varys on a real-world application, we measured throughput and latency of Nginx v1.13.9 [1] using ab benchmark. Nginx was running on the same machine as previous experiments

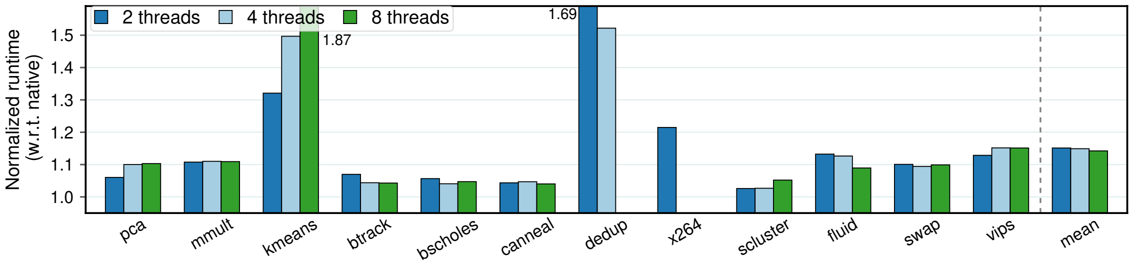


Figure 7: Runtime overhead with different number of threads. (Lower is better.)

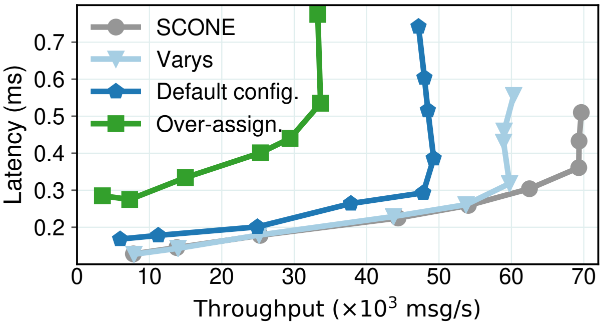


Figure 8: Throughput-latency plots of Nginx. Varys: low-exit system configuration, Default conf.: default configuration of Linux, Over-assign.: another process is competing for a core with Nginx.

and the load generator was connected via a 10Gb network. The results are presented in Figure 8.

In line with the previous measurements, Varys reduces the maximum throughput by 19% if the system is configured for a low AEX rate. Otherwise, the AEX rate becomes higher, cache flushing has to happen more frequently and the overhead increases. The higher rate comes from two sources: disabling DynTicks increases the frequency of timer interrupts and disabling interrupt redirection adds exits caused by network interrupts. Finally, the "Over-assignment" line is the throughput of Nginx in the scenario, when we do not dedicate a core exclusively to Nginx and assign another application that competes for the core (in our case, we use *word_count* from Phoenix). Since the Nginx threads are periodically suspended, the cost of the handshake becomes much higher as both threads in a pair have to wait while one of them is suspended.

7.2 Security Evaluation

Violation of trusted reservation. To evaluate how effective Varys is at ensuring trusted reservation (i.e., if a pair of threads is running on the same physical core), we performed an experiment that emulates a time-sliced attack. We launch a dummy Varys-protected application in normal configuration (all threads are correctly paired) and then, at runtime, change affinity of one of the threads. Additionally, to evaluate the rate of false positives, we run the application without the attack. As trusted reservation is implemented via a periodic handshake, the main configuration parameter is the time limit distinguishing cache hits from cache misses.

Time threshold, SW timer ticks	**False positives,** **%**	**False negatives,** **%**
140	4.0	0.0
160	0.0	0.0
250	0.0	0.1

Table 1: Rate of false positives and false negatives depending on the value of handshake threshold. The threshold is presented for 10 memory accesses.

The results are presented in Table 1. False negatives represent the undetected attacks and false positives—the cases when there was no attack, but a handshake still failed. The results are aggregated over 1000 runs.

As we see, trusted reservation can be considered reliable if the limit is set to 160 ticks of the software timer (§6.2). The fact that we neither have false positives nor false negatives is caused by the difference in timing of L1 and a LLC cache hits. If the threads are on the same core, the handshake will have timing of 10 L1 cache hits. Yet, if they are on different cores, the only shared cache is LLC and all 10 accesses would miss both L1 and L2.

Increased rate of AEX. To evaluate Varys's effectiveness at detecting attacks with high AEX frequencies, we ran a protected application under different system interrupt rates and counted the number of aborts (i.e., detected potential attacks). For the purity of the experiment, the victim was a dummy program that does not introduce additional AEXs on top of the system interrupts. In each of the measurement, we tested several limits on minimal runtime (MRT), inverse of the AEX rate. Similar to the previous experiment, we had 1000 runs.

The results are presented in Table 2. Here, the "Normal rate" is 100Hz (see §5.3); "Low-AEX attack" is 5.5kHz as in the attacks from Wang et al. [45] and Van Bulck et al. [43]; "Common attack" is 10kHz which corresponds to the rate required for cache attacks. We can see that

MRT, IR instructions	**Normal execution**	**Low-AEX attack**	**Common attack**
60M	0.2%	100%	100%
62M	1.2%	100%	100%
64M	10%	100%	100%

Table 2: Varys abort rate depending on the system interrupt rate and on the value of minimum runtime (MRT).

```
for (Set in L1_Cache_Sets):
  for (Very Long):
    for (CLine in CacheLine1..CacheLine8):
      Read(Set, CLine)
```

Figure 9: An example of worst-case victim for a defense mechanism based solely on interrupt frequency.

if we set the threshold on the number of IR instructions between enclave exits to 60 millions, it achieves both low level of false positives (0.2%) and detects all simulated attack attempts.

Residual cache leakage. For small applications (i.e., applications with small or slowly changing working set), cache leakage may persist even after we limit the frequency of enclave exits.

As a worst case, we consider the following application (see Figure 9): it iterates over cache sets, accessing all cache lines in a set for a long time. With such applications, limiting the interrupt frequency will not help, because even a few samples are enough to derive the application state. We use this application to evaluate effectiveness of the cache eviction mechanism proposed in §5.2.

We use a kernel module to launch a time-slicing cache attack on the core running the victim application. The attack delivers an interrupt every 10 ms, and does an L1d cache measurement on all cache sets. We normalize the results into the range of $[0,1]$. Additionally, we disables CPU prefetching both for the victim and attack code to reduce noise. Essentially, it is a powerful kernel-based attack that strives to stay undetected by Varys.

The results of the measurements are on Figure 10a. Without eviction, the attack succeeds and the state of application can be deducted even with a few samples. Then, we apply Varys with L1i and L1d cache eviction to the application (Figure 10b). Even though the amount of information leaked decreases greatly, we can still distinguish some patterns in the heatmap due to residual L2 cache leakage. When we enable L2 eviction in Varys, the results contain no visible information about the victim application (Figure 10b).

8 Hardware Extensions

Many parts of Varys's functionality could be implemented in hardware to improve its efficiency and strengthen the security guarantees. In this section, we propose a few such extensions. We believe that introducing such a functionality would be rather non-intrusive and should not require significant architectural changes.

8.1 Userspace AEX handler

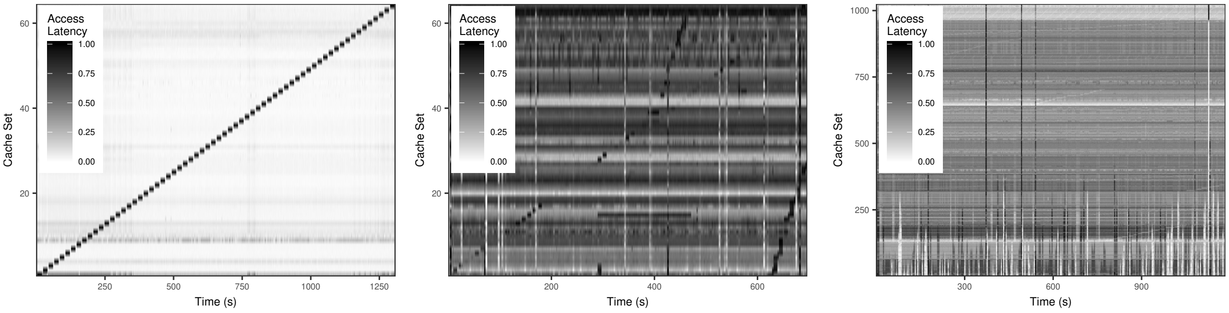
Varys relies on the SGX state saving feature for detection of enclave exits. However, this approach has certain drawbacks: it requires the application to monitor the SSA value, thus increasing the overhead, and it introduces a window of vulnerability (§6.1). An extension to the AEX protocol could solve both of the issues.

Normally during an AEX, the control is passed to the OS exception handler, which further transfers control to the userspace AEX handler, provided by the user. The user AEX handler then executes `ERESUME` instruction, which re-enters the enclave. However, there is no possibility for an in-enclave handler. Our proposed extension adds a hardware triggered callback to the `ERESUME` instruction, specified in the `TCS: TCS.eres_handler`. After each `ERESUME` executed by unprotected code, the enclave is re-entered, and the control is passed to code located at the address `TCS.eres_handler`. To continue executing interrupted in-enclave code, the `ERESUME` handler will execute the `ERESUME` instruction once again, this time, inside the enclave. Note that calling `ERESUME` inside of an enclave is right now not permitted. One difficulty of this extension would be an AEX during the processing of a handler. We would allow recursive calls since handlers could be designed to deal with such recursions.

8.2 Intel CAT extension

Although Intel CAT could be used to prevent concurrent LLC attacks, the OS has complete control over the CAT configuration, which renders the defense ineffective. It can be solved by associating the CAT configuration registers with *version numbers* that are automatically incremented each time the configuration changes. The application could check the version number in the AEX handler after each AEX and thus easily detect the change. In case, no support for AEX handlers is added, the application could perform periodic checks within the enclave instead.

To estimate the potential impact of the extension, we ran an experiment where Nginx was protected by Varys and had a slice of LLC exclusively allocated to it (see Figure 11). As we see, allocating 4 and 2 MB of cache did not cause a significant slowdown for the given workload. The difference in throughput comes mainly from the larger eviction region: Varys had to flush 4 MB instead of 256 KB. However, allocating this large part of the cache can significantly reduce the overall system performance. At the same time, if we try a more modest allocation, we risk causing a much higher rate of cache misses, which is what happened with the 1 MB allocation in our experiment.



(a) No eviction.

(b) L1 eviction.

(c) L2 eviction.

Figure 10: An experiment proving the effectiveness of cache eviction. Without eviction, we can easily see the program behavior. With L1 eviction, the L2 residual leak exposes some information. With L2 eviction, no visible information is exposed. Graphs have different time scales due to different overhead from L1/L2 measurement and presence of eviction mechanism. Color reflects normalized values, with different absolute minimum and maximum values for every graph.

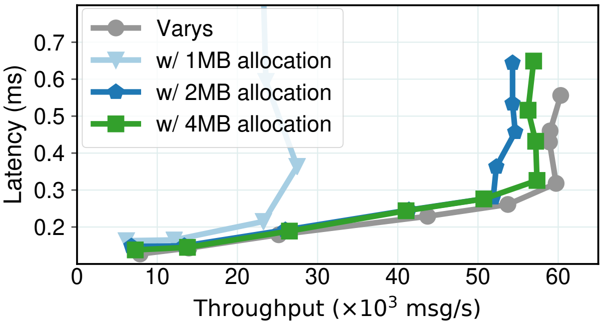


Figure 11: Impact of different cache allocation sizes on throughput and latency of Nginx protected by Varys.

8.3 Trusted HW timer

Since the hardware timer (RDTSC/P instruction) is not available in SGX1, we use a software timer, which wastes a hyperthread. SGX2 is going to introduce the timer, but we cannot rely on it either as privileged software can overwrite and reset its value.

We see two ways of approaching this problem: We may introduce a monotonically increasing read-only timer which could be used as-is. Alternatively, we could introduce a version number that is set to a random value each time the timer is overwritten. To ensure the timer correctness, the application would have to compare the version of this register before and after the measurement.

9 Related Work

The idea of restricting minimal runtime was proposed by Varadarajan et al. [44], although they relied on features of privileged software. Similarly, Déjà Vu [14] relies on measuring execution time of execution paths at run-time.

T-SGX [40] uses Transactional Synchronization Extensions (TSX) to detect and hide page faults from the OS. It protects against page fault attacks, but not page-bit and cache timing attacks. Cloak [22] strives to extend T-SGX guarantees to cache attacks by preloading sensitive data, but requires source code modifications.

Concurrently with our work, an alternative approach to establishing thread co-location was proposed in HyperRace [13]. It uses data races on a shared variable as a way of distinguishing L1 from LLC sharing. Accordingly, it does not require a timer thread.

Zhang et al. [51] and Godfrey at al. [19] employ flushing as a defense against cache attacks, and Cock [15] proposed to used lattice scheduling [16] as an optimization. All of them rely on privileged software.

Among the alternatives, Racoon [37] builds on the idea of oblivious memory [20] and makes enclaves' memory accesses independent of the input by introducing fake accesses, but requires manual changes in code. Dr. SGX [8] automates the obfuscation. Shinde et al. [41] make the accesses deterministic at the page level. Both introduce high overheads (in the range of 3–20×).

10 Conclusion

We presented Varys, an approach to protecting SGX enclaves from side channel attacks. Varys protects from multiple side channels and causes low overheads. Conceptually, Varys protects against side channels by limiting the sharing of core resources like L1 and L2 caches. We have shown that implementing it in software is possible with reasonable overhead. With additional hardware support, we would not only expect a more straightforward implementation of Varys but also lower overhead and protection against a wider range of side channel attacks, including LLC-based ones.

Acknowledgments. We thank our anonymous reviewers for the helpful comments. This work was partly funded by the Federal Ministry of Education and Research of the Federal Republic of Germany (03ZZ0517A, FastCloud) and by Horizon 2020 Research and Innovation Programme (690111, SecureCloud).

References

[1] nginx: The architecture of open source applications. `www.aosabook.org/en/nginx.html`, 2016. Accessed: May, 2018.

[2] ACIIÇMEZ, O. Yet another microarchitectural attack: Exploiting I-cache. In *Workshop on Computer Security Architecture* (2007).

[3] ARNAUTOV, S., TRACH, B., GREGOR, F., KNAUTH, T., MARTIN, A., PRIEBE, C., LIND, J., MUTHUKUMARAN, D., O'KEEFFE, D., STILLWELL, M. L., GOLTZSCHE, D., EYERS, D., KAPITZA, R., PIETZUCH, P., AND FETZER, C. SCONE: Secure Linux Containers with Intel SGX. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (2016).

[4] BAUMANN, A., PEINADO, M., AND HUNT, G. Shielding applications from an untrusted cloud with Haven. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (2014).

[5] BERNSTEIN, D., LANGE, T., AND SCHWABE, P. The security impact of a new cryptographic library. *Progress in Cryptology–LATINCRYPT 2012* (2012).

[6] BERSHAD, B. N., LEE, D., ROMER, T. H., AND CHEN, J. B. Avoiding conflict misses dynamically in large direct-mapped caches. In *ACM SIGPLAN Notices* (1994).

[7] BIENIA, C., AND LI, K. PARSEC 2.0: A new benchmark suite for chip-multiprocessors. In *Proceedings of the 5th Annual Workshop on Modeling, Benchmarking and Simulation (MoBS)* (2009).

[8] BRASSER, F., CAPKUN, S., DMITRIENKO, A., FRASSETTO, T., KOSTIAINEN, K., MÜLLER, U., AND SADEGHI, A.-R. DR. SGX: Hardening SGX Enclaves against Cache Attacks with Data Location Randomization. *arXiv:1709.09917* (2017).

[9] BRASSER, F., MÜLLER, U., DMITRIENKO, A., KOSTIAINEN, K., CAPKUN, S., AND SADEGHI, A.-R. Software Grand Exposure: SGX Cache Attacks Are Practical. *arXiv preprint arXiv:1702.07521* (2017).

[10] BRUMLEY, B. B., AND HAKALA, R. M. Cache-timing template attacks. In *International Conference on the Theory and Application of Cryptology and Information Security* (2009), Springer.

[11] CHE TSAI, C., PORTER, D. E., AND VIJ, M. Graphene-SGX: A practical library OS for unmodified applications on SGX. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)* (2017).

[12] CHEN, G., CHEN, S., XIAO, Y., ZHANG, Y., LIN, Z., AND LAI, T. H. SGXPECTRE Attacks: Leaking Enclave Secrets via Speculative Execution. *arXiv preprint arXiv:1802.09085* (2018).

[13] CHEN, G., WANG, W., CHEN, T., CHEN, S., ZHANG, Y., WANG, X., LAI, T.-H., AND LIN, D. Racing in Hyperspace: Closing Hyper-Threading Side Channels on SGX with Contrived Data Races. In *IEEE Symposium on Security and Privacy* (2018).

[14] CHEN, S., REITER, M. K., ZHANG, X., AND ZHANG, Y. Detecting Privileged Side-Channel Attacks in Shielded Execution with Déjà Vu. In *ASIA CCS '17* (2017).

[15] COCK, D. Practical probability: Applying pGCL to Lattice scheduling. In *Interactive Theorem Proving: 4th International Conference* (2013).

[16] DENNING, D. E. A lattice model of secure information flow. *Communications of the ACM* (1976).

[17] DISSELKOEN, C., KOHLBRENNER, D., PORTER, L., AND TULLSEN, D. Prime+Abort: A Timer-Free High-Precision L3 Cache Attack using Intel TSX. In *Usenix Security* (2017).

[18] GE, Q., YAROM, Y., LI, F., AND HEISER, G. Contemporary Processors Are Leaky – and There's Nothing You Can Do About It. *arXiv preprint arXiv:1612.04474* (2016).

[19] GODFREY, M., AND ZULKERNINE, M. A server-side solution to cache-based side-channel attacks in the cloud. In *IEEE Sixth International Conference on Cloud Computing (CLOUD)* (2013).

[20] GOLDREICH, O., AND OSTROVSKY, R. Software protection and simulation on oblivious RAMs. *Journal of the ACM (JACM)* (1996).

[21] GÖTZFRIED, J., ECKERT, M., SCHINZEL, S., AND MÜLLER, T. Cache attacks on Intel SGX. In *European Workshop on System Security (EuroSec)* (2017).

[22] GRUSS, D., LETTNER, J., SCHUSTER, F., OHRIMENKO, O., HALLER, I., AND COSTA, M. Strong and Efficient Cache Side-Channel Protection using Hardware Transactional Memory. In *26th USENIX Security Symposium (USENIX Security 17)* (2017).

[23] GUERON, S. Intel's new AES instructions for enhanced performance and security. In *Fast Software Encryption: 16th International Workshop* (2009).

[24] HÄHNEL, M., CUI, W., AND PEINADO, M. High-Resolution Side Channels for Untrusted Operating Systems. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)* (2017).

[25] INCI, M. S., GULMEZOGLU, B., IRAZOQUI, G., EISENBARTH, T., AND SUNAR, B. Cache attacks enable bulk key recovery on the cloud. In *International Conference on Cryptographic Hardware and Embedded Systems* (2016).

[26] INTEL CORPORATION. *Intel® 64 and IA-32 Architectures Software Developer's Manual*. 2016.

[27] IRAZOQUI, G., EISENBARTH, T., AND SUNAR, B. S$A: A shared cache attack that works across cores and defies VM sandboxing – and its application to AES. In *2015 IEEE Symposium on Security and Privacy* (2015).

[28] KESSLER, R. E., AND HILL, M. D. Page placement algorithms for large real-indexed caches. *ACM Transactions on Computer Systems (TOCS)* (1992).

[29] KOCHER, P., GENKIN, D., GRUSS, D., HAAS, W., HAMBURG, M., LIPP, M., MANGARD, S., PRESCHER, T., SCHWARZ, M., AND YAROM, Y. Spectre Attacks: Exploiting Speculative Execution. *arXiv preprint arXiv:1801.01203v1* (2018).

[30] LATTNER, C., AND ADVE, V. LLVM: A compilation framework for lifelong program analysis and transformation. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)* (2004).

[31] LIU, F., GE, Q., YAROM, Y., MCKEEN, F., ROZAS, C., HEISER, G., AND LEE, R. B. CATalyst: Defeating Last-Level Cache Side Channel Attacks in Cloud Computing. In *HPCA* (2016).

[32] LIU, F., YAROM, Y., GE, Q., HEISER, G., AND LEE, R. B. Last-level cache side-channel attacks are practical. In *2015 IEEE Symposium on Security and Privacy* (2015).

[33] MARSHALL, A., HOWARD, M., BUGHER, G., AND HARDEN, B. Security best practices for developing windows azure applications. Microsoft Corp, 2010.

[34] OLEKSENKO, O., KUVAISKII, D., BHATOTIA, P., AND FETZER, C. Fex: A Software Systems Evaluator. In *Proceedings of the 47st International Conference on Dependable Systems & Networks (DSN)* (2017).

[35] ORENBACH, M., LIFSHITS, P., MINKIN, M., AND SILBERSTEIN, M. Eleos: ExitLess OS Services for SGX Enclaves. In *EuroSys* (2017).

[36] PERCIVAL, C. Cache missing for fun and profit. 2005.

[37] RANE, A., LIN, C., AND TIWARI, M. Raccoon: Closing digital side-channels through obfuscated execution. In *USENIX Security Symposium* (2015).

[38] RANGER, C., RAGHURAMAN, R., PENMETSA, A., BRADSKI, G., AND KOZYRAKIS, C. Evaluating MapReduce for multi-core and multiprocessor systems. In *Proceedings of the 13th International Symposium on High Performance Computer Architecture (HPCA)* (2007).

[39] SCHWARZ, M., WEISER, S., GRUSS, D., MAURICE, C., AND MANGARD, S. Malware guard extension: Using SGX to conceal cache attacks. *CoRR abs/1702.08719* (2017).

[40] SHIH, M., LEE, S., AND KIM, T. T-SGX: Eradicating controlled-channel attacks against enclave programs. In *NDSS* (2017).

[41] SHINDE, S., CHUA, Z. L., NARAYANAN, V., AND SAXENA, P. Preventing Page Faults from Telling Your Secrets. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security - ASIA CCS '16* (2016).

[42] TROMER, E., OSVIK, D., AND SHAMIR, A. Efficient cache attacks on AES, and countermeasures. *Journal of Cryptology 23*, 1 (2010).

[43] VAN BULCK, J., WEICHBRODT, N., KAPITZA, R., PIESSENS, F., AND STRACKX, R. Telling Your Secrets Without Page Faults: Stealthy Page Table-Based Attacks on Enclaved Execution. In *Usenix Security* (2017).

[44] VARADARAJAN, V., RISTENPART, T., AND SWIFT, M. Scheduler-based defenses against cross-VM side-channels. In *23rd USENIX Security Symposium (USENIX Security 14)* (2014).

[45] WANG, W., CHEN, G., PAN, X., ZHANG, Y., WANG, X., BINDSCHAEDLER, V., TANG, H., AND GUNTER, C. A. Leaky Cauldron on the Dark Land: Understanding Memory Side-Channel Hazards in SGX. *arXiv preprint arXiv:1705.07289* (2017).

[46] WU, Z., XU, Z., AND WANG, H. Whispers in the hyper-space: High-speed covert channel attacks in the cloud. In *USENIX Security Symposium* (2012).

[47] XU, Y., CUI, W., AND PEINADO, M. Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems. In *IEEE Symposium on Security and Privacy* (2015).

[48] YAROM, Y., AND FALKNER, K. FLUSH+RELOAD: A high resolution, low noise, L3 cache side-channel attack. In *USENIX Security Symposium* (2014).

[49] ZHANG, Y., JUELS, A., REITER, M. K., AND RISTENPART, T. Cross-VM side channels and their use to extract private keys. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security* (2012), CCS '12.

[50] ZHANG, Y., JUELS, A., REITER, M. K., AND RISTENPART, T. Cross-tenant side-channel attacks in paas clouds. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security* (2014).

[51] ZHANG, Y., AND REITER, M. K. Düppel: Retrofitting commodity operating systems to mitigate cache side channels in the cloud. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security* (2013).