Racing in Hyperspace: Closing Hyper-Threading Side Channels on SGX with Contrived Data Races

Guoxing Chen*, Wenhao Wang†‡, Tianyu Chen†, Sanchuan Chen*, Yinqian Zhang*,
XiaoFeng Wang†, Ten-Hwang Lai*, Dongdai Lin‡
*The Ohio State University, †Indiana University Bloomington
‡SKLOIS, Institute of Information Engineering, Chinese Academy of Sciences
{chen.4329,chen.4825,zhang.834,lai.1}@osu.edu, {wangwenhao,ddlin}@iie.ac.cn, {chen512,xw7}@indiana.edu

***Abstract*—In this paper, we present HYPERRACE, an LLVM-based tool for instrumenting SGX enclave programs to eradicate all side-channel threats due to Hyper-Threading. HYPERRACE creates a shadow thread for each enclave thread and asks the underlying untrusted operating system to schedule both threads on the same physical core whenever enclave code is invoked, so that Hyper-Threading side channels are closed completely. Without placing additional trust in the operating system's CPU scheduler, HYPERRACE conducts a physical-core co-location test: it first constructs a communication channel between the threads using a shared variable inside the enclave and then measures the communication speed to verify that the communication indeed takes place in the shared L1 data cache—a strong indicator of physical-core co-location. The key novelty of the work is the measurement of communication speed without a trustworthy clock; instead, relative time measurements are taken via contrived data races on the shared variable. It is worth noting that the emphasis of HYPERRACE's defense against Hyper-Threading side channels is because they are open research problems. In fact, HYPERRACE also detects the occurrence of exception- or interrupt-based side channels, the solutions of which have been studied by several prior works.**

I. INTRODUCTION

The growing demands for secure data-intensive computing and rapid development of hardware technologies bring in a new generation of hardware support for scalable trusted execution environments (TEE), with the most prominent example being Intel Software Guard Extensions (SGX). SGX is a set of CPU instructions that enable a user-land process to allocate a chunk of private memory, called an *enclave*, to protect its execution from the untrusted operating system (OS) and even a rogue system administrator. Sensitive data outside the enclave are encrypted, and only decrypted within the enclave, when they are loaded into the CPU, to avoid direct exposure of their content to the untrusted parties (*i.e.*, the OS and the administrator). With all such protection in place, however, today's SGX design has been found to still leak out the program's runtime traces through various *side channels*, allowing the OS-level adversary to *infer* sensitive data processed inside the enclave.

One example of such side channels is the page-fault channels [1], [2] in which the adversary with full control of the OS can induce page faults (by manipulating the page tables inside the kernel) during an enclave program's runtime, so as to identify the secret data the program's page access pattern depends upon. The page-fault attacks have been improved recently [3], [4] by monitoring the updates of *accessed* flag in the page table entries (PTEs) by the enclave program to infer its page access pattern without causing page faults. Besides, traditional micro-architectural side channels also exist in the SGX context, including the CPU cache attacks [5], [6], [7], [8], branch target buffer (BTB) attacks [9], cache-DRAM attacks [4], *etc*. A comprehensive list of memory side channels in SGX has been summarized in a prior paper [4].

Same-core side channels. To collect information through any of these side channels, the adversary needs to either run the attack program on the same core executing the enclave program (same-core side channels) or monitor the victim's operations from a different core (cross-core side channels), depending on the nature of the channel he uses. A prominent example of cross-core channels is the last-level cache (LLC) [10], [11], [12], [13], [14], in which the attack program operates on another core and measures its own use of the LLC to infer the victim's cache usage. Cross-core side channels in SGX are no different from those in other contexts, which tend to be noisy and often harder to exploit in practice (*e.g.*, to synchronize with the victim). By comparison, the noise-free and easy-to-exploit same-core side channels are uniquely threatening under the SGX threat model. Conventional ways to exploit same-core channels are characterized by a large number of exceptions or interrupts to frequently transfer the control of a core back and forth between an enclave process and the attacker-controlled OS kernel, through a procedure called *Asynchronous Enclave Exits (AEX)*. Such *AEX-based side-channel attacks* have been intensively studied [1], [2], [15], [9] and new defense proposals continue to be made, often based upon detection of high frequency AEXs [16], [17]. This feature, however, is found to be evadable through exploiting a set of side channels enabled or assisted by Hyper-Threading (called *Hyper-Threading side-channel attacks*), which do not trigger a large number of interrupts. To the best of our knowledge, no prior work has successfully mitigated Hyper-Threading side channels in SGX.

This paper reports a study that aims at filling this gap, understanding and addressing the security threats from Hyper-Threading side channels in the SGX setting, and deriving

The two lead authors contribute equally to the work and are ordered alphabetically. Corresponding author: Wenhao Wang (wangwenhao@iie.ac.cn).

novel protection to close all Hyper-Threading side channels. In addition, our solution seamlessly integrates with a method to detect AEXs from within the enclave, and thus completely *eliminates* all same-core side channels on SGX.

Challenges. Hyper-Threading is Intel's simultaneous multi-threading (SMT) technologies implemented in many of its mainstream processors today (*e.g.*, Xeon and Core 'i' Series). While Hyper-Threading greatly increases the degree of instruction-level parallelism, by allowing two threads to share the same physical core and hence many per-core resources, it also enables or assists a variety of side-channel attacks. For example, because micro-architectural resources, such as the BTB, the translation lookaside buffer (TLB), the L1 instruction cache and data cache, the unified L2 cache, and the floating-point units (FPU), are shared between the two logical cores of the same physical core, side-channel attacks that leverage these shared resources to extract secrets are enabled [18], [19], [20], [21], [22], [23], [24], [25]. Moreover, Hyper-Threading facilitates some types of side-channel attacks. For example, in the SPM attacks that monitor the *accessed* flag of the PTEs, an adversary may take advantage of Hyper-Threading to flush the TLB entries of the victim enclave, forcing a page table walk and a PTE update when the memory page is visited by the enclave program again [4].

Defending against the Hyper-Threading side-channel leaks is challenging. Simply disabling Hyper-Threading is not an option, because it greatly degrades the performance of the processors, making the SGX systems less suitable for data-intensive computing. Moreover, even if it is reasonable for the owner of the enclaves to request the code to run only on CPUs not supporting Hyper-Threading or with the feature disabled, there is no effective way for software programs running inside SGX enclaves to verify this artifact: The enclave code cannot execute the `cpuid` instruction directly to learn the number of available cores; the `rdtscp` and `rdpid` instructions return the current processor ID from the `IA32_TSC_AUX` register [26], which, however, is controlled by the untrusted OS. Furthermore, these instructions are not currently supported in the enclave mode. Remote attestation does not cover information about Hyper-Threading, either. One viable solution is to create a *shadow* thread from the enclave program and ask the OS to schedule it on the other logical core, so that no other process can share the same physical core as the enclave program. However, it is very challenging to reliably verify such a scheduling arrangement performed by the untrusted OS. To make this approach work, we need an effective *physical-core co-location test* to determine whether two threads are indeed scheduled on the same physical core.

HYPERRACE. A micro-architecture feature we can leverage to conduct reliable physical-core co-location tests is that the two enclave threads running on the same physical core can communicate (through a shared variable inside the same enclave) with each other much faster through per-core caches (*e.g.*, the L1 cache) than the communication between physical cores (or CPU packages) through the L3 cache or the memory. However, this fast communication channel requires a reliable and trustworthy clock to measure the communication speed, which, unfortunately, is absent inside SGX enclaves: the SGX version 1 processors do not support the `rdtsc/rdtscp` instructions in the enclave mode; and although SGX version 2 plans to introduce support for the `rdtsc/rdtscp` instructions, the clock seems to be untrusted and can still be changed by the OS [26, Chapter 38.6.1]. Without such a clock, measurements of the communication speed, which are critical for verifying the co-location of two threads on the same core, become difficult.

To address this problem, we present in this paper a unique technique that utilizes *contrived data races between two threads of the same enclave program to calibrate their inter-communication speed using the speed of their own executions*. More specifically, data races are created by instructing both threads to simultaneously read from and write to a shared variable. By carefully constructing the read-write sequences (Sec. IV), it is ensured that when both threads operate on the same core, they will read from the shared variable the value stored by the other thread with very high probabilities. Otherwise, when the threads are scheduled to different cores, they will, with high probabilities, only observe values stored by themselves.

The contrived data races establish an *"authenticated"* communication channel because, first, the shared variable is located *inside* the enclave's protected memory so that its confidentiality and integrity are protected by SGX, and, second, the measurement of the channel's communication speed is *verified* by the execution speed of the communication code. The security guarantee of this verification lies in the adversary's inability to arbitrarily manipulate the relative speed between the threads' execution speed and their inter-communication speed. Our security analysis demonstrates that even an adversary that controls the entire OS cannot schedule the two threads on different physical cores while ensuring they will observe data races on the shared variable with high probabilities.

Using this technique, we designed and implemented an LLVM-based tool, called HYPERRACE, which compiles an enclave program from the source code and instruments it at the intermediate representation (IR) level to conduct frequent AEX and co-location tests during the execution of the enclave program. The resulting binary is an enclave program that automatically protects itself from all Hyper-Threading side-channel attacks (and other same-core side-channel attacks), completely closing such side channels. We combine an analytical security model with empirical measurements on SGX processors to conduct a thorough security analysis on our scheme. We also empirically conducted several attacks to subvert the co-location tests and found all of them can be effectively detected by HYPERRACE. Our performance evaluation is conducted by protecting an SGX version of *nbench* and Intel's SGX SSL library. The results suggest that the runtime overhead for *nbench* applications due to the HYPERRACE's instrumentation in each basic block (for detecting AEXs) ranges from 42.8% to 101.8%. The runtime overhead due to co-location tests is about

3.5% (when the co-location tests were conducted 250 times per second, triggered by benign, period system interrupts), which grows linearly in the number of times co-location tests are conducted. The combined runtime overhead for various cryptographic algorithms in the SGX SSL library is 36.4%.

Contributions. We outline the contributions of the paper as follows:

- *A viable solution to an open problem*. We propose a solution to the open research problem of defending against SGX side-channel attacks on Hyper-Threading-enabled processors, and demonstrated its effectiveness.
- *A novel approach to physical-core co-location tests*. We developed a new technique to conduct physical-core co-location tests, by leveraging contrived data races to calibrate the communication speed between threads with the pace of program executions.
- *A turn-key solution*. We developed an LLVM-based tool, HYPERRACE, to protect enclave programs by automatically instrumenting them with AEX and co-location detection code.

Roadmap. The rest of the paper is organized as follows: Sec. II provides the background of our research; Sec. III presents an overview of HYPERRACE; Sec. IV describes our physical-core co-location test technique; Sec. V presents the security analysis of the co-location tests; Sec. VI elaborates the design and implementation of HYPERRACE; Sec. VII provides the results of performance evaluation on our prototype; Sec. VIII reviews the related prior research and Sec. IX concludes the paper.

II. BACKGROUND

In this section, we describe background knowledge on cache coherence protocols, store buffers, Intel SGX and Hyper-Threading.

Cache and memory hierarchy. Modern processors are equipped with various buffers and caches to improve their performance. Relevant to our discussion are cache coherence protocols and the store buffer.

- *Cache coherence protocols.* Beginning with the Pentium processors, Intel processors use the `MESI` cache coherence protocol to maintain the coherence of cached data [26]. Each cache line in the L1 data cache and the L2/L3 unified caches is labeled as being in one of the four states defined in Table I. When writing to a cache line labeled as *Shared* or *Invalid*, a Read For Ownership (RFO) operation will be performed, which broadcasts invalidation messages to other physical cores to invalidate the copies in their caches. After receiving acknowledgement messages from other physical cores, the write operation is performed and the data is written to the cache line.
- *Store Buffer.* The RFO operations could incur long delays when writing to an invalid cache line. To mitigate these delays, store buffers were introduced. The writes will be pushed to the store buffer, and wait to be executed when the acknowledgement messages arrive. Since the writes are buffered, the following reads to the same address may not see the most up-to-date value in cache. To solve this problem, a technique called store-to-load forwarding is applied to forward data from the store buffer to later reads.

Intel SGX. Intel Software Guard Extensions (SGX) is new hardware feature available on recent Intel processors that provides an shielded execution environment, called an enclave, to software applications, which protects confidentiality and integrity of enclave programs against privileged attackers, such as the operating system (OS). The enclaves' code and data is stored in Processor Reserved Memory (PRM), a region of the DRAM. Accesses to the memory regions belonging to an enclave inside the PRM from any software outside of the enclave are denied.

To switch between enclave mode and non-enclave mode, SGX provides `EENTER` and `EEXIT` instructions to start and terminate enclave execution. During the enclave execution, interrupts or exceptions will cause the processor to transition out of the enclave mode, which is called an Asynchronous Enclave eXit (AEX). To protect the security of the enclave, an AEX will perform a series of operations, including flushing TLBs and saving the state of certain registers in a State Save Area (SSA) inside the enclave memory. An `ERESUME` operation resumes the enclave execution after an AEX occurs.

Intel Hyper-Threading. Hyper-Threading Technology is Intel's proprietary implementation of simultaneous multi-threading (SMT), which enables a single physical processor to execute two concurrent code streams [26]. With Hyper-Threading support, a physical core consists of two logical cores sharing the same execution engine and the bus interface. Each logical core has a separated architectural state, such as general purpose registers, control registers, local APIC registers, *etc*.

Beside the shared execution engine and bus interface, the following resources are also shared between two logical cores of the same physical core supporting Hyper-Threading.

- Caches: the private caches (*i.e.*, L1/L2) of a physical core are shared between the two logical cores.
- Branch prediction units (BPU): the two logical cores share the branch target buffer (BTB) which is a cache storing a target address of branches.
- Translation lookaside buffers (TLB): data TLBs are shared between two logical cores, while the instruction TLB may be shared or duplicated depending on specific processors.
- Thermal monitors: the automatic thermal monitoring mechanism and the catastrophic shutdown detector are shared.

Most processors with SGX also support Hyper-Threading. We surveyed a list of Intel processors that supports SGX and listed the results in Table VIII (see Appendix A).

III. HYPERRACE OVERVIEW

Before diving into the design details, in this section, we highlight the motivation of the paper, an overview of HYPERRACE's design, and the threat model we consider in this paper.

TABLE I
MESI CACHE LINE STATES.

Cache Line State	M(Modified)	E(Exclusive)	S(Shared)	I(Invalid)
This line is valid?	Yes	Yes	Yes	No
Copies exists in other processors' cache?	No	No	Maybe	Maybe
A read to this line	Cache hit	Cache hit	Cache hit	Goes to system bus
A write to this line	Cache hit	Cache hit	Read for ownership	Read for ownership

TABLE II
HYPER-THREADING SIDE CHANNELS.

Side Channels	Shared	Cleansed at AEX	Hyper-Threading only
Caches	Yes	Not flushed	No
BPUs	Yes	Not flushed	No
Store Buffers	No	N/A	Yes
FPUs	Yes	N/A	Yes
TLBs	Yes	Flushed	Yes

A. Motivation

Although Hyper-Threading improves the overall performance of processors, it makes defenses against side-channel attacks in SGX more challenging. The difficulty is exhibited in the following two aspects:

Introducing new attack vectors. When the enclave program executes on a CPU core that is shared with the malicious program due to Hyper-Threading, a variety of side channels can be created. In fact, most the shared resources listed in Sec. II can be exploited to conduct side-channel attacks. For example, prior work has demonstrated side-channel attacks on shared L1 D-cache [20], [21], L1 I-cache [22], [23], [27], BTBs [18], FPUs [19], and store buffers [28]. These attack vectors still exist on SGX processors.

Table II summarizes the properties of these side channels. Some of them can only be exploited with Hyper-Threading enabled, such as the FPUs, store buffers, and TLBs. This is because the FPU and store-buffer side channels are only exploitable by concurrent execution (thus N/A in Table II), and TLBs are flushed upon AEXs. Particularly interesting are the store-buffer side channels. Although the two logical cores of the same physical core have their own store buffers, false dependency due to 4K-aliasing introduces an extra delay to resolve read-after-write hazards between the two logical cores [28], [29]. The rest vectors, such as BPU and caches, can be exploited with or without Hyper-Threading. But Hyper-Threading side channels provide unique opportunities for attackers to exfiltrate information without frequently interrupting the enclaves.

Creating challenges in SGX side-channel defenses. First, because Hyper-Threading enabled or Hyper-Threading assisted side-channel attacks do not induce AEX to the target enclave, these attacks are much stealthier. For instance, many of the existing solutions to SGX side-channel attacks detect the incidences of attacks by monitoring AEXs [17], [16]. However, as shown by Wang *et al.* [4], Hyper-Threading enables the attacker to flush the TLB entries of the enclave program so that new memory accesses trigger one complete page table walk and update the *accessed* flags of the page table entries. This allows attackers to monitor updates to *accessed* flags without triggering any AEX, completely defeating defenses that only detect AEXs.

Second, Hyper-Threading invalidates some defense techniques that leverage Intel's Transactional Synchronization Extensions (TSX)—Intel's implementation of hardware transactional memory. While studies have shown that TSX can help mitigate cache side channels by concealing SGX code inside of hardware transactions and detecting cache line eviction in its write-set or read-set (an artifact of most cache side-channel attacks) [30], it does not prevent an attacker who share the same physical core when Hyper-Threading is enabled (see Sec. VIII). As such, Hyper-Threading imposes unique challenges to defense mechanisms alike.

While disabling Hyper-Threading presents itself as a feasible solution, disabling Hyper-Threading and proving this artifact to the owner of the enclave program through remote attestation is impossible. Modern micro-architectures do not provide such a mechanism that attests the status of Hyper-Threading. As such, enclave programs cannot simply trust the OS kernel to disable Hyper-Threading.

B. Design Summary

To prevent Hyper-Threading side-channel leaks, we propose to create an auxiliary enclave thread, called shadow thread, to occupy the other logic core on the same physical core. By taking over the entire physical core, the Hyper-Threading enabled or assisted attacks can be completely thwarted.

Specifically, the proposed scheme relies on the OS to schedule the protected thread and its shadow thread to the same physical core at the beginning, which is then verified by the protected thread before running its code. Because thread migration between logical cores requires context switches (which induce AEX), the protected thread periodically checks the occurrence of AEX at runtime (through SSA, see Sec. VI-A) and whenever an AEX is detected, verifies its co-location with the shadow thread again, and terminates itself once a violation is detected.

Given the OS is untrusted, the key challenge here is how to reliably verify the co-location of the two enclave threads on the same physical core, in the absence of a secure clock. Our technique is based upon a carefully designed data race to calibrate the speed of inter-thread communication with the pace of execution (Sec. IV).

C. Threat Model

Here, we outline a threat model in which an adversary aims to extract sensitive information from an enclave program protected by SGX through same-core side channels. We assume the adversary has every capability an OS may have over a hosted application (excluding those restricted by SGX), including but not limited to:

- Terminating/restarting and suspending/resuming the enclave program; interrupting its execution through interrupts; intercepting exception handling inside enclaves.
- Scheduling the enclave program to any logical cores; manipulating kernel data structures, such as page tables.
- Altering the execution speed of the enclave program by (1) causing cache contention, (2) altering CPU frequency, and (3) disabling caching.

Design goals. Our design targets same-core side-channel attacks that are conducted from the same physical core where the enclave program runs:

- Hyper-Threading side-channel attacks from the other logical core of the same physical core, by exploiting one or more attack vectors listed in Table II.
- AEX side-channel attacks, such as exception-based attacks (*e.g.*, page-fault attacks [1], [2]), through manipulating the page tables of the enclave programs, and other interrupt-based side-channel attacks (*e.g.*, those exploiting cache [7] or branch prediction units [9]), by frequently interrupting the execution of the enclave program using *Inter-processor interrupts* or *APIC* timer interrupts.

IV. Physical-core Co-Location Tests

In this section, we first present a number of straw-man solutions for physical-core co-location tests and discuss their limitations, and then describe a novel co-location test using contrived data races.

A. Straw-man Solutions

A simple straw-man solution to testing physical-core co-location is to establish a covert channel between the two enclave threads that only works when the two threads are scheduled on the same physical core.

Timing-channel solutions. One such solution is to establish a covert timing channel using the L1 cache that is shared by the two threads. For instance, a simple timing channel can be constructed by measuring the PROBE time of a specific cache set in the L1 cache set in a PRIME-PROBE protocol [20], or the RELOAD time of a specific cache line in a FLUSH-RELOAD protocol [10]. One major challenge of establishing a reliable timing channel in SGX is to construct a trustworthy timing source inside SGX, as SGX version 1 does not have `rdtsc/rdtscp` supports and SGX version 2 provides `rdtsc/rdtscp` instructions to enclave but allows the OS to manipulate the returned values. Although previous work has demonstrated that software clocks can be built inside SGX [17], [5], [4], manipulating the speed of such clocks by tuning CPU core frequency is possible [17]. Fine-grained timing channels for measuring subtle micro-architectural events, such as cache hits/misses, in a strong adversary model is fragile. Besides, timing-channel solutions are also vulnerable to man-in-the-middle attacks, which will be described shortly.

Timing-less solutions. A timing-less scheme has been briefly mentioned by Gruss *et al.* [30]: First, the receiver of the covert channel initiates a transaction using hardware transactional memory (*i.e.*, Intel TSX) and places several memory blocks into the write-set of the transaction (by writing to them). These memory blocks are carefully selected so that all of them are mapped to the same cache set in the L1 cache. When the sender of the covert channel wishes to transmit 1 to the receiver, it accesses another memory blocks also mapped to the same cache set in the L1 cache; this memory access will evict the receiver's cache line from the L1 cache. Because Intel TSX is a cache-based transactional memory implementation, which means the write-set is maintained in the L1 cache, evicting a cache-line in the write-set from the L1 cache will abort the transaction, thus notifying the receiver. As suggested by Gruss *et al.*, whether or not two threads are scheduled on the same physical core can be tested using error rate of the covert channel: 1.6% when they are on the same core vs. 50% when they are not on the same core.

Man-in-the-middle attacks. As acknowledged in Gruss *et al.* [30], the aforementioned timing-less solution may be vulnerable to man-in-the-middle attacks. In such attacks, the adversary can place another thread to co-locate with both the sender thread and the receiver thread, and then establish covert channels with each of them separately. On the sender side, the adversary monitors the memory accesses of the sender using side channels (*e.g.*, the exact one that is used by the receiver), and once memory accesses from the sender is detected, the signal will be forwarded to the receiver thread by simulating the sender on the physical core where the receiver runs. The timing-channel solutions discussed in this section are also vulnerable to such attacks.

Covert-channel (both timing and timing-less) based co-location tests are vulnerable to man-in-the-middle attacks because these channels can be used by any software components in the system, *e.g.*, the adversary outside SGX enclaves can mimic the sender's behavior. Therefore, in our research, we aim to derive a new solution to physical-core co-location tests that do not suffer from such drawbacks—by observing memory writes inside enclaves that cannot be performed by the adversary. We will detail our design in the next subsection.

B. Co-Location Test via Data Race Probability

Instead of building micro-architectural covert channels between the two threads that are supposed to occupy the two logic cores of the same physical core, which are particularly vulnerable to man-in-the-middle attacks, we propose a novel co-location test that verifies the two threads' co-location status by measuring their probability of observing data races on a shared variable inside the enclave.

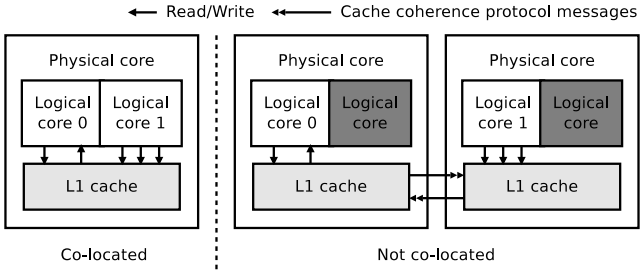

Fig. 1. Data races when threads are co-located/not co-located.

In this section, we first illustrate the idea using a simplified example, and then refine the design to meet the security requirements. A hypothesis testing scheme is then described to explain how co-location is detected by comparing the observed data race probability with the expected one.

An illustrating example. To demonstrate how data race could be utilized for co-location tests, consider the following example:

1. An integer variable, $\mathcal{V}$, shared by two threads is allocated inside the enclave.
2. Thread T_0 repeatedly performs the following three operations in a loop: writing 0 to $\mathcal{V}$ (using a `store` instruction), waiting N (*e.g.*, $N = 10$) CPU cycles, and then reading $\mathcal{V}$ (using a `load` instruction).
3. Thread T_1 repeatedly writes 1 to $\mathcal{V}$ (using a `store` instruction).

There is a clear data race between these two threads, as they write different values to the same variable concurrently. When these two threads are co-located on the same physical core, thread T_0 will read 1, the value written by thread T_1, from the shared variable $\mathcal{V}$ with a high probability (close to 100%). In contrast, when these two threads are located on different physical cores, thread T_0 will observe value 1 with very low probability (*i.e.*, close to zero).

Such a drastic difference in the probability of observing data races is caused by the location in which the data races take place. As shown in Fig. 1, when the two threads are co-located, data races happen in the L1 cache. Specifically, both thread T_0 and T_1 update the copy of $\mathcal{V}$ in the L1 data cache. However, the frequency of thread T_0's updates to the shared variable $\mathcal{V}$ is much lower than that of T_1, because the additional read and N-cycle waiting in thread T_0 slow down its execution. Therefore, even though the `load` instruction in thread T_0 can be fulfilled by a store-to-load forwarding from the same logical core, when the `load` instruction retires, almost always the copy of $\mathcal{V}$ in the L1 cache is the value stored by thread T_1, invalidating the value obtained from store-to-load forwarding [31]. As such, the `load` instruction in thread T_0 will read value 1 from $\mathcal{V}$ with a very high probability.

However, when the two threads are not co-located—*e.g.*, thread T_0 runs on physical core C_0 and thread T_1 runs on physical core C_1—the data races happen in the L1 cache of physical core C_0. According to the cache coherence protocol, after thread T_0 writes to $\mathcal{V}$, the corresponding cache line in C_0's L1 cache, denoted by $\texttt{CL}_0$, transitions to the *Modified* state. If T_0's `load` instruction is executed while $\texttt{CL}_0$ is still in the same state, thread T_0 will read its own value from $\texttt{CL}_0$. In order for thread T_0 to read the value written by thread T_1, one necessary condition is that $\texttt{CL}_0$ is invalided before the `load` instruction of thread T_0 starts to execute. However, this condition is difficult to meet. When thread T_1 writes to $\mathcal{V}$, the corresponding cache line in C_1's L1 cache, denoted by $\texttt{CL}_1$, is in the *Invalidate* state due to T_0's previous *store*. T_1's update will send an invalidation message to $\texttt{CL}_0$ and transition $\texttt{CL}_1$ to the *Modified* state. However, because the time needed to complete the cache coherence protocol is much longer than the time interval between thread T_0's write and the following read, $\texttt{CL}_0$ is very likely still in the *Modified* state when the following read is executed. Hence, thread T_0 will read its own value from variable $\mathcal{V}$ with a high probability.

A refined data-race design. The above example illustrates the basic idea of our physical-core co-location tests. However, to securely utilize data races for co-location tests under a strong adversarial model (*e.g.*, adjusting CPU frequency, disabling caching), the design needs to be further refined. Specifically, the refined design aims to satisfy the following requirements:

- Both threads, T_0 and T_1, observe data races on the same shared variable, $\mathcal{V}$, with high probabilities when they are co-located.
- When T_0 and T_1 are not co-located, at least one of them observes data races with low probabilities, even if the attacker is capable of causing cache contention, adjusting CPU frequency, or disabling caching.

To meet the first requirement, T_0 and T_1 must both write and read the shared variable. In order to read the value written by the other thread with high probabilities, the interval between the *store* instruction and the *load* instruction must be long enough to give the other thread a large window to *overwrite* the shared variable. Moreover, when the two threads are co-located, their execution time in one iteration must be roughly the same and remain constant. If a thread runs much faster than the other, it will have a low probability of observing data races, as its *load* instructions are executed more frequently than the *store* instructions of the slower thread. To satisfy the second requirement, instructions that have a non-linear slowdown when under interference (*e.g.*, cache contention) or execution distortion (*e.g.*, CPU frequency change or cache manipulation) should be included.

The code snippets of refined thread T_0 and T_1 are listed in Fig. 2. Specifically, each co-location test consists of n rounds, with k data race tests per round. What follows is the common routine of T_0 and T_1:

1. Initialize the round index `%rdx` to n (running the test for n rounds); and reset counter `%rcx`, which is used to count the number of data races (the number of times observing the other thread's data).
2. Synchronize T_0 and T_1. Both threads write their round index `%rdx` to the other thread's `sync_addr` and read

Thread T_0

```
1  <initialization>:
2     mov $colocation_count, %rdx
3     xor %rcx, %rcx
4     ; co-location test counter
5  <synchronization>:
6     ··· ; acquire lock 0
7   .sync0:
8     mov %rdx, (sync_addr1)
9     cmp %rdx, (sync_addr0)
10    je .sync1
11    jmp .sync0
12  .sync1:
13    mfence
14    mov $0, (sync_addr0)
15 <initialize a round>:
16    mov $begin0, %rsi
17    mov $1, %rbx
18    mfence
19    mov $addr_v, %r8
20 <co-location test>:
21  .L0:
22 <load>:
23    mov (%r8), %rax
24 <store>:
25    mov %rsi, (%r8)
26 <update counter>:
27    mov $0, %r10
28    mov $0, %r11
29    cmp $end0, %rax
30    ; a data race happens?
31    cmovl %rbx, %r10
32    sub %rax, %r9
33    cmp $1, %r9
34    ; continuous number?
35    cmova %r11, %r10
36    add %r10, %rcx
37    shl $b_count, %rbx
38    ; bit length of $count
39    mov %rax, %r9
40    ; record the last number
41 <padding instructions 0>:
42    nop
43    nop
44     ⋮
45    nop
46    mov (%r8), %rax
47    mov (%r8), %rax
48     ⋮
49    mov (%r8), %rax
50    dec %rsi
51    cmp $end0, %rsi
52    jne .L0
53    ; finish 1 co-location test
54 <all rounds finished?>:
55    ··· ; release lock 1
56    dec %rdx
57    cmp $0, %rdx
58    jne .sync0
```

Thread T_1

```
1  <initialization>:
2     mov $colocation_count, %rdx
3     xor %rcx, %rcx
4     ; co-location test counter
5  <synchronization>:
6     ··· ; release lock 0
7   .sync2:
8     mov %rdx, (sync_addr0)
9     cmp %rdx, (sync_addr1)
10    je .sync3
11    jmp .sync2
12  .sync3:
13    mfence
14    mov $0, (sync_addr1)
15 <initialize a round>:
16    mov $begin1, %rsi
17    mov $1, %rbx
18    mfence
19    mov $addr_v, %r8
20 <co-location test>:
21  .L2:
22 <load>:
23    mov (%r8), %rax
24 <update counter>:
25    mov $0, %r10
26    mov $0, %r11
27    cmp $end0, %rax
28    ; a data race happens?
29    cmovg %rbx, %r10
30    sub %rax, %r9
31    cmp $1, %r9
32    ; continuous number?
33    cmova %r11, %r10
34    add %r10, %rcx
35    shl $b_count, %rbx
36    ; bit length of $count
37     mov %rax, %r9
38    ; record the last number
39 <store>:
40    mov %rsi, (%r8)
41 <padding instructions 1>:
42    mov (%r8), %rax
43    lfence
44    mov (%r8), %rax
45    lfence
46    mov (%r8), %rax
47    lfence
48    mov (%r8), %rax
49    lfence
50    mov (%r8), %rax
51    lfence
52    dec %rsi
53    cmp $end1, %rsi
54    jne .L2
55    ; finish 1 co-location test
56 <all rounds finished?>:
57    ··· ; acquire lock 1
58    dec %rdx
59    cmp $0, %rdx
60    jne .sync2
```

Fig. 2. Co-location detection code.

from each others' `sync_addr`. If the values match (*i.e.*, they are in the same round), T_0 and T_1 begin the current round of co-location test.

3. At the beginning of each round, set the test index `%rsi` to $b_0 + k$ for T_0 and to $b_1 + k$ for T_1. Therefore, T_0 will write b_0+k, b_0+k-1, b_0+k-2, $\cdots$, b_0+1 to the shared variable; T_1 will write $b_1 + k$, $b_1 + k - 1$, $b_1 + k - 2$, $\cdots$, b_1+1. $[b_0, b_0+k]$ does not overlap with $[b_1, b_1+k]$ so either thread, when writes its `%rsi` to $\mathcal{V}$ and reads from it, knows whether it receives the input from the other thread. After that, initialize the address of shared variable $\mathcal{V}$ in `%r8`.
4. For T_0, store the content of `%rsi` to $\mathcal{V}$, determine whether a data race happens, and update `%rcx` if so. For T_1, determine whether a data race happens, update `%rcx` if so, and then store `%rsi` to $\mathcal{V}$. A data race is counted if and only if contiguous values written by the other thread are read from $\mathcal{V}$, which indicates that the two threads run at the same pace.
5. Record the data race in a counter using the conditional move (*i.e.*, `CMOV`) instruction. This avoids fluctuations in the execution time due to conditional branches.
6. Execute the padding instructions to (1) make the execution time of T_0 and T_1 roughly the same; (2) increase the interval between the *store* instruction and the *load* instruction; (3) create non-linear distortion in the execution time when being manipulated (see discussions in Sec. V).
7. Decrease `%rsi` by 1 and check whether it hits b_0 (for T_0) or b_1 (for T_1), which indicates the end of the current round. If so, go to step 8. Otherwise, go to step 4.
8. Decrease `%rdx` by 1 and check whether it becomes 0. If so, all rounds of tests finish; Otherwise, go to step 2.

The time for one data race test for thread T_0 and T_1 is roughly the same when both threads are running on the same physical core. As shown in Fig. 3, when the two threads are co-located, since the interval from `load` to `store` (line 22 to 24 for T_0, line 22 to 39 for T_1) is much shorter than the interval between `store` and `load` (line 24 to 52 then jump to 21 for T_0, line 39 to 54, including the serializing instruction `lfence`, then jump to 21 for T_1), there is a high probability that the `store` operation from the other thread will fall into the interval between the `store` and `load`. As a result, each thread becomes much more likely to see the other's data than its own. In contrast, when the two threads are not co-located, the communication time between the two physical cores is longer than the interval between `store` and `load`: that is, even when one thread's `store` is performed in the other's `store` to `load` interval, the data of the `store` will not be seen by the other due to the delay caused by the cache coherence protocol. Therefore, data races will not happen.

Fig. 3. The basic idea of the data race design. Monitoring the memory operations of the two threads on $\mathcal{V}$. LD: `load`; ST: `store`.

Testing co-location via statistical hypothesis testing. To determine whether two threads are co-located on the same physical core, we perform the following hypothesis test.

During each round of a co-location test, k samples are collected by each thread. We consider the k samples as $k-1$ unit tests; each unit test consists of two consecutive samples: if both samples observe data races (and the observed counter values are also consecutive), the unit test passes; otherwise it fails. We take the i-th ($i = 1, 2, \ldots, k-1$) unit test from each round (of the n rounds), and then consider this n unit

tests as n independent Bernoulli trials. Then, we have $k-1$ groups of Bernoulli trials. We will conduct $k-1$ hypothesis tests for each of the two threads as follows, and consider the co-location test as passed if any of the $k-1$ hypothesis tests accepts its null hypothesis:

We denote the j-th unit test as a binary random variable X_j, where $j = 1, 2, \ldots, n$; $X_j = 1$ indicates the unit test passes, and $X_j = 0$ otherwise. We assume when the two threads are co-located, the probability of each unit test passing is p. Therefore, when they are co-located, $P(X_j = 1) = p$. We denote the actual ratio of passed unit tests in the n tests as $\hat{p}$. The null and alternative hypotheses are as follows:

H_0: $\hat{p} \geq p$; the two threads are co-located.

H_1: $\hat{p} < p$; the two threads are not co-located.

Because X_j is a test during round j and threads T_0 and T_1 are synchronized before each round, we can consider $X_1, X_2, \cdots, X_n$ independent random variables. Therefore, the sum of n random variables, *i.e.*, $X = \sum_{j=1}^{n} X_j$, follows a Binomial distribution with parameters n and p. The mean of the Binomial distribution is $E(X) = np$ and the variance is $D(X) = np(1-p)$. When n is large, the distribution of X can be approximated by a normal distribution $N(np, np(1-p))$. Let the significance level be α. Then

$$Pr\left[\frac{X - np}{\sqrt{np(1-p)}} < -u_\alpha\right] = \alpha.$$

We will reject H_0 and decide that the two threads are not co-located, if

$$X < np - u_\alpha\sqrt{np(1-p)}.$$

In our prototype implementation, we parameterized n, p, and α. For example, when $n = 256$ and $\alpha = 0.01$, $u_\alpha = 2.33$. From the measurement results given in Table V (Sec. V), the probabilities for T_0 and T_1 to see data races with co-location are $p_0 = 0.969$ and $p_1 = 0.968$, respectively. So we have for both T_0 and T_1

$$Pr\left[X < 242\right] = 0.01.$$

In other words, in the hypothesis test, we reject the null hypothesis if less than 242 unit tests (out of the 256 tests) pass in T_0 (or T_1).

Here the probability of a type I error (*i.e.*, falsely rejecting the null hypothesis) is about 1%. The probability of a type II error is the probability of falsely accepting H_0 when the alternative hypothesis H_1 is true. For example, when X follows a normal distribution of $N(np, np(1-p))$ and $p = 0.80$, the probability of a type II error in T_0 and T_1 will be (let $Z = \frac{X-np}{\sqrt{np(1-p)}} \sim N(0,1)$):

$$\begin{aligned} & Pr\left[X \geq 242 \;\middle|\; X \sim N(np, np(1-p))\right] \\ = \; & Pr\left[Z \geq \frac{242 - 256 \cdot 0.80}{\sqrt{256 \cdot 0.80 \cdot (1-0.80)}} \;\middle|\; Z \sim N(0,1)\right] < 0.01\%. \end{aligned}$$

Practical considerations. The above calculation only provides us with theoretic estimates of the type I and type II errors of the hypothesis tests. In practice, because system events cannot be truly random and independent, approximation has to be made. Particularly, the two threads are only synchronized between rounds, and the k samples in each round are collected without re-synchronization. Therefore, although samples in different rounds can be considered independent, the k samples within the same round may be dependent. Second, within each round, a truly random variable X requires T_0 and T_1 to start to monitor data races uniformly at random, which is difficult to achieve in such fine-grained data race measurements. We approximate the true randomness using the pseudo-randomness introduced in the micro-architecture events (*e.g.*, data updates in the L1 cache reflected in memory reads) during the synchronization. To account for the dependence between unit tests in the same round and the lack of true randomness of each unit test, we select the i-th unit test from each round to form the i-th n-sample hypothesis test, and consider the co-location test as passed if any of the $k-1$ hypothesis tests accepts its null hypothesis. We will empirically evaluate how this design choice impacts the type I errors and type II errors in Sec. V-C.

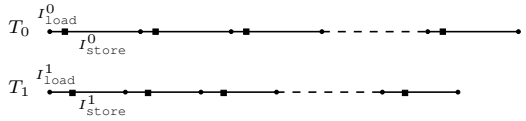

Fig. 4. The model of thread T_0 and thread T_1. •: `load`; ■: `store`.

V. Security Analysis of Co-location Tests

In this section, we provide an analysis on the security of the co-location tests. To do so, we first establish the relationship between the execution time of the communicating threads and the data race probability. We next empirically estimate the execution time of the threads under a variety of execution conditions that the adversary may create (*e.g.*, Priming caches, disabling caching, adjusting CPU frequencies, *etc*.) and then apply the measurement results to analytically proof that, under all attacker-created conditions we have considered, the data race probability cannot reach the same level as that when the two threads are co-located. Finally, we empirically performed attacks against our proposed scheme and demonstrated that none of the attacks could pass the co-location tests.

A. Security Model

To establish the relationship between the execution time of the communicating threads and the probability of data races, we first construct execution models of thread T_0 and thread T_1 (see their code snippets in Fig. 2). Particularly, we abstract the execution of T_0 and T_1 as sequences of alternating load and store operations on the shared variable $\mathcal{V}$. After each load or store operation, some delays are introduced by the padding instructions. We use I^i_w, where $w \in \{\texttt{store}, \texttt{load}\}$ and $i \in \{0, 1\}$ to denote a code segment between two instructions for thread T_i: when w is `load`, the segment is from `load` to `store` (line 22 to 24 for T_0, line 22 to 39 for T_1; see

Fig. 2); when w is `store`, the segment begins with the `store` instruction and ends with the first `load` encountered (line 24 to 52 then jump to 21 for T_0, line 39 to 54, then jump to 21 for T_1).

The execution time of these code segments depends on their instructions and the memory hierarchy v on which the data access (to variable $\mathcal{V}$) operation w is performed (*i.e.*, memory access latency). Therefore, the execution time of the code segment I^i_w is denoted by $\mathcal{T}(I^i_w, v)$, where $i \in \{0,1\}$ and $v \in \{\texttt{L1}, \texttt{L2}, \texttt{LLC}, \texttt{Memory}\}$. We further denote $\mathcal{T}^i_{w,v} = \mathcal{T}(I^i_w, v)$ for short. As such, the period of thread T_i's one iteration of the store and load sequence (line 22 to 52, then jump to 21 for T_0, line 22 to 54, jump to 21 for T_1) is $\mathcal{R}^i_v = \mathcal{T}^i_{\texttt{load},v} + \mathcal{T}^i_{\texttt{store},v}$, *i.e.*, the time between two adjacent `load` instructions' retirements of thread T_i when the data accesses take place in memory hierarchy v.

We use variable $\mathcal{G}_{v,u}$, where $u \in \{\texttt{c}, \texttt{nc}\}$, to denote the communication time, *i.e.*, the time that the updated state of $\mathcal{V}$ appears in the other thread's memory hierarchy v, after one thread modifies the shared variable $\mathcal{V}$, if two threads are co-located ($u = \texttt{c}$) or not co-located ($u = \texttt{nc}$).

Consider the data race happens in memory hierarchy v. If $\mathcal{T}^i_{\texttt{store},v} < \mathcal{G}_{v,u}$, $i \in \{0,1\}$, during the time thread $T_{i\oplus 1}$'s updated state of $\mathcal{V}$ is propagated to thread T_i's memory hierarchy v, T_i has updated $\mathcal{V}$ and fetched data from v at least once. As a result, data races will not happen. In contrast, if $\mathcal{T}^i_{\texttt{store},v} \geq \mathcal{G}_{v,u}$, a data race will happen if the data value of $\mathcal{V}$ is propagated from thread $T_{i\oplus 1}$ to T_i's memory hierarchy v during $\mathcal{T}^i_{\texttt{store},v}$.

Further, if $\mathcal{T}^i_{\texttt{store},v} \geq \mathcal{R}^{i\oplus 1}_v$, at least one store from thread $T_{i\oplus 1}$ will appear in v during $\mathcal{T}^i_{\texttt{store},v}$. Then data races will be observed by thread T_i. If $\mathcal{T}^i_{\texttt{store},v} < \mathcal{R}^{i\oplus 1}_v$, the data race probability of thread T_i will be $\mathcal{T}^i_{\texttt{store},v}/\mathcal{R}^{i\oplus 1}_v$, since the faster the store-load operations of T_i compared with the other thread's iteration, the less likely T_i will see the other's data. Hence, we have the data race probability of thread T_i ($i \in \{0,1\}$):

$$p_i \begin{cases} = 0 & \text{if } \mathcal{T}^i_{\texttt{store},v} < \mathcal{G}_{v,u} \\ \approx \min(\mathcal{T}^i_{\texttt{store},v}/\mathcal{R}^{i\oplus 1}_v, 1) & \text{if } \mathcal{T}^i_{\texttt{store},v} \geq \mathcal{G}_{v,u} \end{cases}. \quad (1)$$

It is worth noting that when the two threads run at drastically different paces, the faster thread will have a low probability to observe data races, as its `load` instructions are executed more frequently than the `store` instructions of the slower thread. Therefore, we implicitly assume that $\mathcal{R}^0_v$ is close to $\mathcal{R}^1_v$. This implicit requirement has been encoded in our design of the unit tests: the way we count data race requires two consecutive data races to read consecutive counter values from the other thread.

Necessary conditions to pass the co-location tests: To summarize, in order to pass the co-location tests, an adversary would have to force the two threads to execute in manners that satisfy the following necessary conditions: (1) They run at similar paces. That is, $\mathcal{R}^0_v/\mathcal{R}^1_v$ is close to 1. (2) The communication speed must be faster than the execution speed of the threads. That is, $\mathcal{T}^i_{\texttt{store},v} \geq \mathcal{G}_{v,u}$, where $i \in \{0,1\}$. (3) $\mathcal{T}^i_{\texttt{store},v}/\mathcal{R}^{i\oplus 1}_v$ must be close to 1, where $i \in \{0,1\}$, to ensure high probabilities of observing data races.

TABLE III
TIME INTERVALS (IN CYCLES) OF T_0 AND T_1.

	T_0		T_1	
	$\mathcal{T}^0_{\texttt{store},v}$	$\mathcal{R}^0_v$	$\mathcal{T}^1_{\texttt{store},v}$	$\mathcal{R}^1_v$
Caching Enabled	95.90	96.30	88.70	98.69
Caching Disabled	1.32e+5	1.35e+5	1.34e+4	2.57e+4

B. Security Analysis

In this section, we systematically analyze the security of the co-location tests by investigating empirically whether the above necessary conditions can be met when the two threads are *not co-located*. Our empirical analysis is primarily based on a Dell Optiplex 7040 machine equipped with a Core i7-6700 processor. We also conducted experiments on machines with a few other processors, such as E3-1280 V5, i7-7700HQ, i5-6200U (see Table V).

We consider the scenarios in which the two threads T_0 and T_1 are placed on different CPU cores by the adversary and the data races are forced to take place on the memory hierarchy v, where $v = \{L1/L2, LLC, memory\}$. We discuss these scenarios respectively.

1) L1/L2 Cache Data Races: We first consider the cases where $v = \{L1, L2\}$. This may happen when the adversary simply schedule T_0 and T_1 on two cores without cache intervention (*e.g.*, cache Priming or caching disabling). However, the adversary is capable of altering the CPU frequency on which T_0 or T_1 runs to manipulate $\mathcal{T}^i_{\texttt{store},v}$ and $\mathcal{G}_{v,nc}$.

Latency of cache accesses. We use the pointer-chasing technique [24], [13] to measure cache access latencies. Memory load operations are chained as a linked list so that the address of the next pointer depends on the data of previous one. Thus the memory accesses are completely serialized. In each measurement, we access the same number of cache-line sized and aligned memory blocks as the number of ways of the cache at the specific cache level, so that every memory access induces cache hits on the target cache level and cache misses on all lower-level caches. According to the result averaged over 10,000,000 measurements, the average value of cache access latencies for the L1/L2/L3 caches were 4, 10 and 40 cycles, respectively.

Cross-core communication time. We developed a test program with two threads: Thread T_a repeatedly writes to a shared variable in an infinite loop, without any additional delays between the two consecutive writes. Thread T_b runs on a different physical core, which after writing to the shared variable executes a few dummy instructions to inject a delay, and then reads from the variable to check for data race. The execution time of the dummy instructions can be used to measure the communication time: When dummy instructions are short, T_b will observe no data race; but when the execution time of the dummy instructions increases to certain threshold,

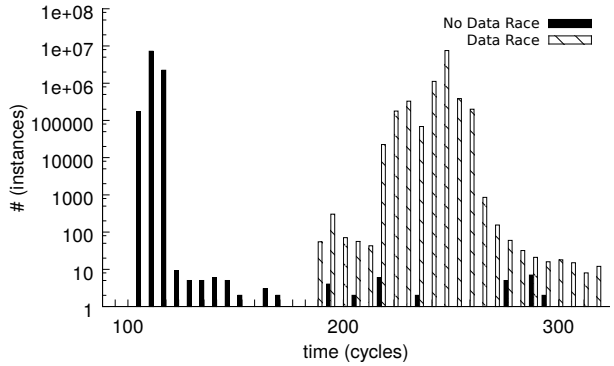

Fig. 5. Demonstration of the cross-core communication time. There is no data race if the dummy instructions take time shorter than 190 cycles.

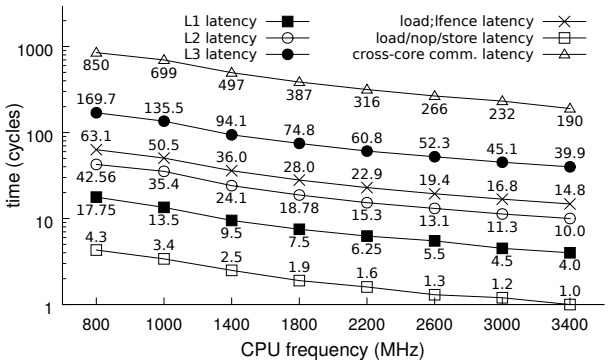

Fig. 6. The effects of frequency changing on execution speed, cache latencies, and cross-core communication time.

T_b will start to observe data race. We draw the histogram of 10,000,000 measurements (Fig. 5). The solid bars represent measurements in which data races were not observed (*i.e.*, T_b reads its own data) and the shaded bars represent measurements where data races were observed (*i.e.*, T_b reads T_a's data). From the experiment, we see when the execution time of the dummy functions is less than 190 cycles, data races were hardly observed. Therefore, we believe the latency of cross-core communication is about 190 cycles.

Effects of frequency changes. In our experiments, we managed the CPU frequency with the support of Hardware-Controlled Performance states (HWP). Specifically we first enabled HWP by the writing to the `IA32_PM_ENABLE` MSR, then configured the frequency range by the writing to the `IA32_PM_REQUEST` MSR. To understand the relation between instructions latencies and the CPU frequency, we evaluated the latency of L1/L2/L3 cache accesses, the latency of executing `nop`,`load`, and `store` instructions, respectively, and the latency of executing the `store; lfence` instruction sequence, under different CPU frequencies. We also measured the cross-core communication speed under these frequencies. The measurements were conducted in a tight loop, averaged over 10,000,000 tests. The results are plotted in Fig. 6. The results suggest that when the CPU frequency changes from 3.40 Ghz to 800 Mhz the instruction execution speed ($4.3\times$), cache access latencies ($4.25\times$–$4.44\times$), and cross-core communication time ($4.47\times$) are affected in the similar order of magnitude.

Discussion. For $v \in \{\texttt{L1}, \texttt{L2}\}$, we have $\mathcal{G}_{v,\texttt{c}} \leq 12$ cycles (the latency for a L2 access) and $\mathcal{G}_{v,\texttt{nc}} > 190$ cycles (the latency of cross-core communication). According to Table III, $\mathcal{T}^0_{\texttt{store},v} = 95.90$ and $\mathcal{T}^1_{\texttt{store},v} = 88.70$. Therefore, $\mathcal{G}_{v,\texttt{c}} < \mathcal{T}^i_{\texttt{store},v} < \mathcal{G}_{v,\texttt{nc}}$, $i \in \{0, 1\}$. As such, data races will happen only if the two threads are co-located. Altering the CPU frequency will not change the analysis. According to Fig. 6, frequency changes have similar effects on $\mathcal{T}^i_{\texttt{store},v}$ and $\mathcal{G}_{v,nc}$. That is, when the CPU frequency is reduced, both $\mathcal{T}^i_{\texttt{store},v}$ and $\mathcal{G}_{v,nc}$ will increase, with similar derivatives. As a result, when the adversary places T_0 and T_1 on different cores, and reduces the frequency of these two cores, their communication speed will be slowed down at the same pace as the slowdown of the execution.

2) LLC Data Races: We next consider the cases where $v = \{LLC\}$. This may happen when the adversary PRIMEs the private caches used by T_0 and T_1 (from co-located logical cores) to evict the shared variable $\mathcal{V}$ to the LLC.

Effects of cache PRIMEs. The data races can occur on the shared LLC when the copies of $\mathcal{V}$ in the private L1 and L2 caches are invalidated, which can only be achieved by having an attacking thread frequently PRIMEing the shared L1/L2 caches from the co-located logical core. To counter such attacks, thread T_0 and T_1 both include in their padding instructions redundant `load` instructions (*i.e.*, line 46 to 49 of T_0 and line 42 to 50 of T_1 in Fig. 2). These `load` instructions precede the `load` instruction that measures data races, thus they effectively pre-load $\mathcal{V}$ into the L1/L2 caches to prevent the adversary's PRIMEs of related cache lines. This mechanism not only defends against attempts to PRIME local L1/L2 caches, but TLBs and paging structure caches.

Discussion. According to our measurement study, the time needed to PRIME one cache set in L1 and one cache set in L2 (to ensure that $\mathcal{V}$ is not in L1 and L2 cache) is at least $10 \times (w_{L2} - 1) + 40 \times 1$ cycles (w_{L2} is the number of cache lines in one L2 cache set), which is significantly larger than the interval between the pre-load instructions and the actual `load` instruction (*i.e.*, 1 cycle). Moreover, because CPU frequency changes are effective on both logical cores of the same physical core, altering CPU frequency will not help the adversary. Therefore, we conclude that data race cannot happen on LLC.

3) Data Races in Main Memory: We next consider the cases where $v = \{Memory\}$. This may happen when the adversary (1) PRIMEs the caches, (2) invalidates the caches, or (3) disables the caching.

Latency of cache invalidation instructions. According to Intel software developers manual [26, Chapter 8.7.13.1], the `wbinvd` instruction executed on one logical core can invalidate the cached data of the other logical core of the same physical core. Directly measuring the latency of cache

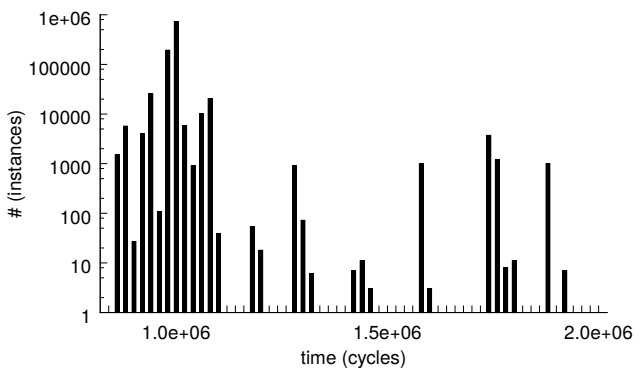

Fig. 7. The histogram of `wbinvd` execution time over 1,000,000 measurements.

TABLE IV
INSTRUCTION LATENCIES (IN CYCLES) CAUSED BY DISABLING CACHING.

Instructions	Caching enabled	Caching disabled	Slowdown
nop	1.00	901	901×
load	1.01	1266	1253×
store	1.01	978	968×
load; lfence	14.82	2265	153×

invalidation using the `wbinvd` instruction is difficult. Instead, we measure the execution time of `wbinvd` to approximate the latency of cache invalidation. This is reasonable because `wbinvd` is a serialized instruction. Specifically, we conducted the following experiments: We run `wbinvd` in a tight loop for 1,000,000 times and measure the execution time of each loop, which is shown in Fig. 7. We observe that in some cases the latency is as high as 2×10^6 cycles, which typically happens early in the experiments, while most of the times the latency is only 1×10^6 cycles. We believe this is because dirty cache lines need to be written back to the memory in the first few tests, but later tests usually encounter already-empty caches.

Effects of disabling caching. The attacker can disable caching on a logical core by setting the `CD` bit of control registers. According to Intel Software Developer's Manual [26, Chapter 8.7.13.1], "the `CD` flags for the two logical processors are *ORed* together, such that when any logical processor sets its `CD` flag, the entire cache is nominally disabled." This allows the adversary to force an enclave thread to enter the no-fill caching mode. According to Intel's manual [26, Sec. 11.5.3 and Table 11-5], after setting the `CD` bit, the caches need to be flushed with `wbinvd` instruction to insure system memory coherency. Otherwise, cache hits on reads will still occur and data will be read from valid cache lines. The adversary can also disable caching of the entire PRM by setting the `PRMRR` [32, Chapter 6.11.1], as "all enclave accesses to the `PRMRR` region always use the memory type specified by the `PRMRR`, unless the `CR0.CD` bit on one of the logical processors on the core running the enclave is set." It is worth noting that the `PRMRR_BASE` and `PRMRR_MASK` MSRs are set in an early booting stage, and cannot be updated after the system boots.

We measured the latency of the `nop`, `load`, `store` instructions, and the `load;lfence` instruction sequence, respectively, in tight loops (averaged over 10,000,000 measurements) with the caching enabled and disabled. The results are shown in Table IV. The slowdowns were calculated by comparing the latency with caching disabled and enabled. It can be seen that the slowdowns of `nop`, `load`, and `store` instructions are around $1000\times$. But the slowdown of `load;lfence` instruction sequence is only two orders of magnitude. This result leads to the non-linear distortion of T_1 when caching are disabled (see Fig. 2), which is also shown in Table III: $\mathcal{T}^0_{\texttt{store},v}$ and $\mathcal{T}^1_{\texttt{store},v}$ are on the same order of magnitude when caching is enabled but become drastically different when caching is disabled (*i.e.*, $1.32\texttt{e}{+5}$ vs. $1.34\texttt{e}{+4}$).

Discussion. A prerequisite of observing data races in the memory is that the load operations miss L1/L2/LLC caches. This may be achieved using one of the following mechanisms:

- *Evicting the shared variable to memory on-the-fly.* The adversary could leverage two approaches to evict the shared variable to memory: (1) Flushing cache content using the `wbinvd` instruction. However, as the latency of the instruction (on the order of 10^6 cycles) is too large (see Fig. 7), it cannot effectively evict the shared variable to memory. In fact, during the execution of the `wbinvd` instruction, caches can still be filled normally. We have empirically confirmed that co-location tests that happen during the execution of the `wbinvd` instruction are not affected. (2) Evicting the cache content using PRIME-PROBE techniques. However, according to our measurement study, the time needed to PRIME one cache set in LLC is at least $40 \times w_{LLC}$ cycles (w_{LLC} is the number of cache lines in one LLC slides), which is significantly larger than the interval between the pre-load instructions and the actual `load` instruction (*i.e.*, 1 cycle). Even if the adversary could distribute the task of cache PRIMEs to multiple threads running on different CPU cores, which is by itself challenging due to cache conflicts among these threads, the gap of speed should be huge enough to prevent such attacks. We will empirically verify this artifact in Sec. V-C.
- *Disabling caching.* We have examined several approaches to disable caching: First, the adversary can disable caching by editing `PRMRR`, which will be effective after system reboots. Second, the adversary can interrupt the co-location tests before the `load` instructions and flush the cache content using the `wbinvd` instruction or PRIME-PROBE operations (though interruption of the co-location tests will be detected and thus restart the co-location tests). Third, the adversary can disable the caching of the two physical cores on which T_0 and T_1 executes by setting the `CD` bits of the control registers. However, none of this methods can pass the co-location tests. This is because we use `load` instructions as paddings in thread T_0, and use `load` followed by `lfence` instructions as paddings in thread T_1. If caching is disabled, the slowdown of "`load; lfence`" is much smaller than the other instructions, since the former

already serializes the `load` operations (see Table IV). As a result, the relative speed of the two threads changes significantly (see Table III). Particularly, as $\mathcal{R}_v^0/\mathcal{R}_v^1$ is no longer close to 1, the co-location tests will not pass.

- *Altering CPU frequency when caching is disabled.* We further consider the cases of changing CPU frequency after disabling caching by setting the `CD` bits. Suppose the frequency change slows down thread T_0 and T_1 by a factor of c_0 and c_1, respectively, which are constant. Then $\mathcal{T}^0_{\texttt{store},v} = c_0 \cdot 1.32 \times 10^5$, $\mathcal{R}_v^0 = c_0 \cdot 1.35 \times 10^5$, $\mathcal{T}^1_{\texttt{store},v} = c_1 \cdot 1.34 \times 10^4$, $\mathcal{R}_v^1 = c_1 \cdot 2.57 \times 10^4$, according to Table III. Then, based upon Equa. (1), the data race probabilities of T_0 and T_1 are $\hat{p}_0 = \min(\frac{c_0 \cdot 1.32 \times 10^5}{c_1 \cdot 2.57 \times 10^4}, 1)$ and $\hat{p}_1 = \min(\frac{c_1 \cdot 1.34 \times 10^4}{c_0 \cdot 1.35 \times 10^5}, 1)$ respectively. Since $\hat{p}_0 \cdot \hat{p}_1 \leq \frac{c_0 \cdot 1.32 \times 10^5}{c_1 \cdot 2.57 \times 10^4} \cdot \frac{c_1 \cdot 1.34 \times 10^4}{c_0 \cdot 1.35 \times 10^5} \approx 0.51$, we can see that the probability for a thread to observe the data race will not exceed $\sqrt{0.51} \approx 71.4\%$, which has a near zero probability to pass our co-location test.
- *Nonlinear CPU frequency changes.* The only remaining possibility for the adversary to fool the co-location test is to change the CPU frequency nonlinearly so that $\mathcal{T}^0_{\texttt{store},v}$, $\mathcal{T}^0_{\texttt{load},v}$, $\mathcal{T}^1_{\texttt{store},v}$, $\mathcal{T}^1_{\texttt{load},v}$ change independently. However, the CPU frequency transition latency we could achieve on our testbed is between $20\mu s$ and $70\mu s$ (measured using the method proposed by Mazouz *et al.* [33]), which is on the same order of magnitude as $\mathcal{R}_v^1$ when caching is disabled (and thus much larger than $\mathcal{R}_v^1$ when caching is enabled), making it very difficult, if not impossible, to introduce desired nonlinear frequency change during the co-location tests.

In summary, when the data races take place in the memory through any of the methods we discussed above, the attacker cannot achieve high probability of observing data races in both T_0 and T_1. The hypothesis tests will fail in all cases.

C. Empirical Security Evaluation

We empirically evaluated the accuracy of the co-location tests. As the primary goal of the co-location test is to raise alarms when the two threads are not co-located, we define a false positive as a false alarm (*i.e.*, the co-location test fails) when the two threads are indeed scheduled on the same physical core, and a false negative as a missed detection (*i.e.*, the co-location test passes) of the threads' separation.

False positive rates. A False positive of the co-location tests is approximately the combined type I error of two hypothesis tests (from T_0 and T_1, respectively). We run the same code shown in Fig. 2 on four different processors (*i.e.*, i7-6700, E3-1280 v5, i7-7700HQ, and i5-6200U) without modification. The empirical probabilities of passing unit tests by T_0 and T_1 on these processors are listed in Table V. These values are estimated by conducting $25,600,000$ unit tests. Then with parameter $n = 256$ and the corresponding values of p_0 and p_1, we run co-location tests with $\alpha = 0.01$, $\alpha = 0.001$, $\alpha = 0.0001$, respectively. The false positive rates are reported in Table V. Although the empirical values are close to the theoretical values of α, there are cases where the empirical values are $3\times$ the theoretical ones (*i.e.*, on i5-6200U with $\alpha = 0.0001$). This is probably because of the lack of true randomness and independence in our statistical tests (explained in Sec. IV-B). However, these values are on the same order of magnitude. We believe it is reasonable to select a desired α value to approximate false positives in practice.

TABLE V
EVALUATION OF FALSE POSITIVE RATES.

CPU	p_0	p_1	false positive rates ($\alpha =$)		
			0.01	0.001	1e−4
i7-6700	0.969	0.968	0.005	5e−4	4e−5
E3-1280 V5	0.963	0.948	0.004	4e−4	5e−5
i7-7700HQ	0.965	0.950	0.005	5e−4	2e−4
i5-6200U	0.968	0.967	0.006	0.001	3e−4

False negative rates. A false negative of the co-location test is approximately the type II error of the hypothesis test. We particularly evaluated the following four scenarios:

1. The adversary simply places the two threads on two physical cores without interfering with their execution.
2. The adversary simply places the two threads on two physical cores, and further reduces the frequency of the two physical cores to 800 Mhz.
3. The adversary simply places the two threads on two physical cores, and further disabling caching on the cores on which the two threads run, by setting the `CD` flag.
4. The adversary simply places the two threads on two physical cores, and creates 6 threads that concurrently PRIME the same LLC cache set to which the shared variable $\mathcal{V}$ is mapped.

We run $100,000$ co-location tests for every scenarios. The tests were conducted on the i7-6700 processor, with parameter $n = 256$, $p_0 = 0.969$, $p_1 = 0.968$, $\alpha = 0.0001$. Results are shown in Table VI. Column 2 and 3 of the table show $\hat{p}_0$ and $\hat{p}_1$, the probability of passing unit tests under the considered scenarios, respectively. We can see that in all cases, the probabilities of observing data races from T_0 and T_1 are very low (*e.g.*., 0.03% to 2.2%). In all cases, the co-location tests fail, which suggests we have successfully detected that the two threads are not co-located. We only show results with $\alpha = 0.0001$ because larger α values (*e.g.*, 0.01 and 0.001) will lead to even lower false negative rates. In fact, with the data collected in our experiments, we could not achieve any false negatives even with a much smaller α value (*e.g.*, 1e−100). This result suggests it is reasonable to select a rather small α value to reduce false positives while preserving security guarantees. We leave the decision to the user of HYPERRACE.

VI. PROTECTING ENCLAVE PROGRAMS WITH HYPERRACE

In this section, we introduce the overall design and implementation of HYPERRACE that leverages the physical core co-location test presented in the previous sections.

TABLE VI
EVALUATION OF FALSE NEGATIVE RATES.

Scenario	$\hat{p_0}$	$\hat{p_1}$	false negative rates ($\alpha = 1\mathtt{e}{-4}$)
1	0.0004	0.0007	0.000
2	0.0003	0.0008	0.000
3	0.0153	0.0220	0.000
4	0.0013	0.0026	0.000

A. Safeguarding Enclave Programs

HYPERRACE is a compiler-assisted tool that compiles a program from source code into a binary that runs inside enclaves and protects itself from Hyper-Threading side-channel attacks (as well as other same-core side-channel attacks).

At the high-level, HYPERRACE first inserts instructions to create a new thread (*i.e.*, the shadow thread) at runtime, which shares the same enclave with the original enclave code (dubbed the protected thread). If the enclave program itself is already multi-threaded, one shadow thread needs to be created for each protected thread.

HYPERRACE then statically instruments the protected thread to insert two types of detection subroutines at proper program locations, so the subroutines will be triggered periodically and frequently at runtime. The first type of subroutines is designed to let the enclave program detect AEXs that take place during its execution. The second type of subroutines performs the aforementioned physical-core co-location tests. The shadow thread is essentially a loop that spend most of its time waiting to perform the co-location test.

At runtime, the co-location test is executed first when the protected thread and the shadow thread enter the enclave, so as to ensure the OS indeed has scheduled the shadow thread to occupy the same physical core. Once the test passes, while the shadow thread runs in a busy loop, the protected thread continues the execution and frequently checks whether an AEX has happened. Once an AEX has been detected, which may be caused by either a malicious preemption or a regular timer interrupt, the protected thread will instruct the shadow thread to conduct another co-location test and, if passes, continue execution.

AEX detection. HYPERRACE adopts the technique introduced by Gruss *et al.* [30] to detect AEX at runtime, through monitoring the State Save Area (SSA) of each thread in the enclave. Specifically, each thread sets up a marker in its SSA, for example, writing 0 to the address within SSA that is reserved for the instruction pointer register `RIP`. Whenever an AEX occurs, the current value of `RIP` overrides the marker, which will be detected by inspecting the marker periodically.

When an AEX is detected, the markers will be reset to value 0. A co-location test will be performed to check co-location of the two threads, because AEX may indicate a privilege-level switch—an opportunity for the OS kernel to reschedule one thread to a different logical core. By the end of the co-location test, AEX detection will be performed again to make sure no AEX happened during the test.

Co-location test. To check the co-location status, HYPERRACE conducts the physical-core co-location test described in Sec. IV between two threads. Since the shared variable in the test is now in the enclave memory, the adversary has no means to inspect or modify its value. Once the co-location status has been verified, subsequent co-location tests are only needed when an AEX is detected.

B. Implementation of HYPERRACE

HYPERRACE is implemented by extending the LLVM framework. Specifically, the enclave code is complied using Clang [34], a front-end of LLVM [35] that translates C code into LLVM intermediate representation (IR). We developed an LLVM IR optimization pass that inserts the AEX detection code (including a conditional jump to the co-location test routine if an AEX is detected) into every basic block. Further, we insert one additional AEX detection code every q instructions within a basic block, where q is a parameter we could tune. Checking AEX in every basic block guarantees that secret-dependent control flows are not leaked due to side-channel attacks; adding additional checks prevents data-flow leakage. We will evaluate the effects of tuning q in Sec. VII.

The shadow thread is created outside the enclave and system calls are made to set the CPU affinity of the protected thread and the shadow thread prior to entering the enclave. We use spin locks to synchronize the co-location test routines for the protected thread and the shadow thread. Specifically, the shadow thread waits at the spin lock until the protected thread requests a co-location test. If the co-location test fails, the enclave program reacts according to a pre-defined policy, *e.g.*, retries r times and, if all fail, terminates.

VII. PERFORMANCE EVALUATION

In this section, we evaluate the performance overhead of HYPERRACE. All experiments were conducted on a Dell Optiplex 7040 machine with an Intel Core i7-6700 processor and 32GB memory. The processor has four physical cores (8 logical cores). The parameter α of the co-location tests was set to $1\mathtt{e}{-6}$; p_0, p_1, and n were the same as in Sec. V.

A. nbench

We ported *nbench* [36], a lightweight benchmark application for CPU and memory performance testing, to run inside SGX and applied HYPERRACE to defend it against Hyper-Threading side-channel attacks.

Contention due to Hyper-Threading itself. Before evaluating the performance overhead of HYPERRACE, we measured the execution slowdown of *nbench* due to contention from the co-located logical core. This slowdown is not regarded as an overhead of HYPERRACE, because the performance of an enclave program is expected to be affected by resource contention from other programs; a co-located thread running a completely unrelated program is normal.

We set up two experiments: In the first experiment, we run *nbench* applications with a shadow thread (busy looping) executing on a co-located logical core; in the other experiment,

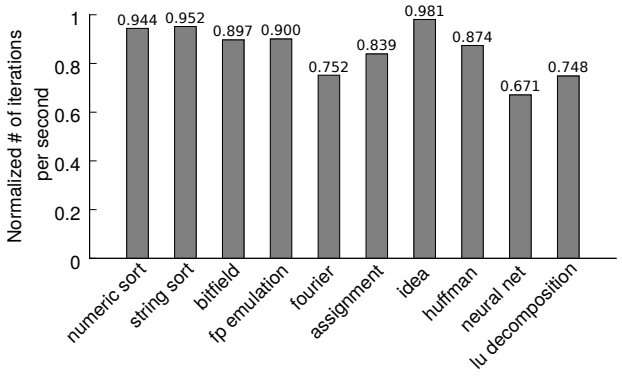

Fig. 8. Normalized number of iterations of *nbench* applications when running with a busy looping program on the co-located logical core.

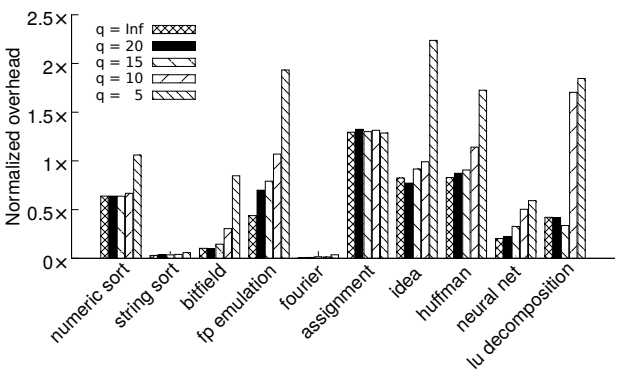

Fig. 9. Runtime overhead due to AEX detection; $q = \texttt{Inf}$ means one AEX detection per basic block; $q = 20/15/10/5$ means one additional AEX detection every q instructions within a basic block.

we run *nbench* with the co-located logical core unused. In both cases, the *nbench* applications were complied *without* HYPERRACE instrumentation. In Fig. 8, we show the normalized number of iterations per second for each benchmark application when a shadow thread causes resource contention; the normalization was performed by dividing the number of iterations per second when the benchmark occupies the physical core by itself.

As shown in Fig. 8, the normalized number of iterations ranges from 67% to 98%. For instance, the benchmark `numeric sort` runs 1544.1 iterations per second with a shadow thread while 1635.2 iterations per second without it, which leads to a normalized value of $1544.1/1635.2 = 0.944$. The following evaluations do not include the performance degradation due to the Hyper-Threading contention.

Overhead due to frequent AEX detection. The performance overhead of the HYPERRACE consists of two parts: AEX detection and co-location tests. We evaluated these two parts separately because the frequency of AEX detection depends on the program structure (*e.g.*, control-flow graph) while the frequency of the co-location tests depends on the number of AEXs detected. We use the execution time of non-instrumented *nbench* applications (still compiled using LLVM) with a shadow thread running on the co-located logical core as the baseline in this evaluation.

To evaluate the overhead of AEX detection, we short-circuited the co-location tests even when AEXs were detected in HYPERRACE. Hence no co-location tests were performed. Fig. 9 shows the overhead of AEX detection. Note that $q = \texttt{Inf}$ means that there is only one AEX detection at the beginning of every basic block; $q = 5$ suggests that if there are more than 5 instructions per basic block, a second AEX detection is inserted; $q = 20$, $q = 15$, and $q = 10$ are defined similarly. Since each instrumentation for AEX detection (by checking SSA) consists of two memory loads (one SSA marker for each thread) and two comparisons, when the basic blocks are small, the overhead tends to be large. For example, the basic blocks in the main loop of `assignment` benchmark application containing only 3 or 4 instructions per basic block, the overhead of HYPERRACE on `assignment` is large (*i.e.*, $1.29\times$) even with $q = \texttt{Inf}$. Generally, the overhead increases as more instrumentations are added. With $q = \texttt{Inf}$, the overhead ranges from 0.8% to 129.3%, with a geometric mean of 42.8%; when $q = 5$, the overhead ranges from 3.5% to 223.7%, with geometric mean of 101.8%.

TABLE VII
MEMORY OVERHEAD (NBENCH).

	Original	$q = 20$	$q = 15$	$q = 10$	$q = 5$
Bytes	207,904	242,464	246,048	257,320	286,448
Overhead	-	16.6%	18.3%	23.7%	37.7%

Overhead due to co-location tests. The overhead of co-location tests must be evaluated when the number of AEX is known. HYPERRACE triggers a co-location test when an AEX happens in one of the two threads or both. By default, the operating system generates timer interrupts and other types interrupts to each logical core. As such, we observe around 250 AEXs on either of these two threads per second. To evaluate the overhead with increased numbers of AEXs, we used a High-Resolution Timers in the kernel (*i.e.*, `hrtimer`) to induce interrupts to cause more AEXs. The overhead is calculated by measuring the overall execution time of one iteration of the *nbench* applications, which includes the time to perform co-location tests when AEXs are detected.

We fixed the instrumentation parameters as $q = 20$ in the tests. The evaluation results are shown in Fig. 10. The overhead of AEX detection has been subtracted from the results. From the figure, we can tell that the overhead of co-location tests is small compared to that of AEX detection. With 250 AEXs per second, the geometric mean of the overhead is only 3.5%; with 1000 AEXs per second, the geometric mean of the overhead is 16.6%. The overhead grows almost linear in the number of AEXs.

Memory overhead. The memory overhead of the enclave code is shown in Table VII. We compared the code size without instrumentation and that with instrumentation under different q values. The memory overhead ranges from 16.6% to 37.7%.

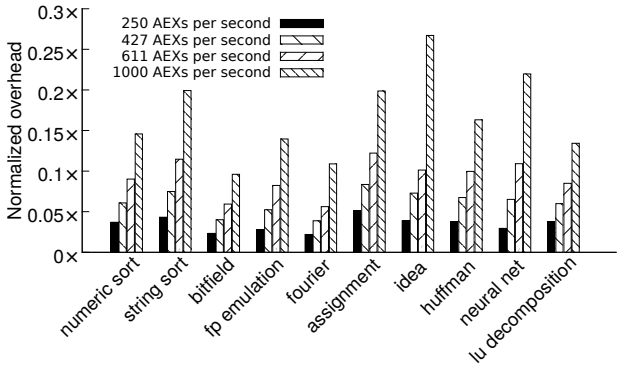

Fig. 10. Runtime overhead of performing co-location tests when $q = 20$.

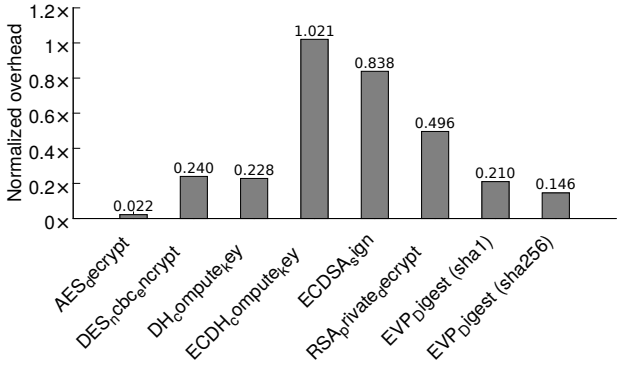

Fig. 11. Overhead of crypto algorithms.

B. Cryptographic Libraries

We also applied HYPERRACE to the Intel SGX SSL cryptographic library [37] and measured the performance overhead of eight popular cryptographic algorithms. We run each algorithm repeatedly for 10 seconds and calculated the average execution time for one iteration. Fig. 11 gives the overhead (for both AEX detection and co-location test) when instrumented every $q = 20$ instructions per basic block, and no extra AEXs introduced (the default 250 AEXs per second).

The overhead for `AES_decrypt` algorithm is small (around 2%) compared to other algorithms since its dominating basic blocks are relative large. In contrast, the overhead for `ECDH_compute_key` and `ECDSA_sign` are relatively large (*i.e.*, 102.1% and 83.8%) because elliptic curve algorithms consist of many small basic blocks. The overhead for other evaluated algorithms ranges from 14.6% to 49.6%. The geometric mean is 36.4%.The size of the complied static trusted library `libsgx_tsgxssl_crypto.a` grew from 4.4 MB to 6.6 MB, resulting in an memory overhead of 50%.

VIII. RELATED WORK

Related to our work is a large volume of literature on micro-architectural side-channel attacks. Many of these attacks leverage various shared resources on Hyper-Threading, such as the L1 D-cache [20], [21], the L1 I-cache [22], [23], [27], branch target buffers [18] and floating-point unit [19], to perform same-core attacks against co-located victim processes. These attacks also work on SGX-enabled processors.

Countermeasures to Hyper-Threading side-channel attacks are less explored. The only known solution is to disable Hyper-Threading. However, because the OS is not trusted by the enclave programs, it cannot be trusted to disable Hyper-Threading. Gruss *et al.* [30] briefly touched upon this problem in their exploration of using TSX to mitigate cache side channels. As the TSX-based solutions do not address Hyper-Threading enabled attacks, they proposed to launch two threads to occupy both logical cores of the physical core, and construct a timing-less covert channel using TSX transactions to verify that the two threads are indeed scheduled on the same core. However, as discussed in Sec. IV-A, covert-channel solutions are vulnerable to man-in-the-middle attacks. As a countermeasure, Gruss *et al.* proposed to randomly choose "a different L1 cache set (out of the 64 available) for each bit to transmit". However, because the adversary can perform a PRIME-PROBE analysis on the entire L1 cache to learn which cache set is used for the covert channel (and at the same time extract the signals), man-in-the-middle attacks are still feasible. In contrast, our scheme does not rely on cache-contention based covert channels; even with the system capability, the adversary cannot simulate the data races that take place inside the enclave, fundamentally addressing the man-in-the-middle threats.

HYPERRACE has been inspired by HomeAlone [38], which utilizes cache side-channel analysis techniques to identify unknown VMs in public clouds. HYPERRACE is different in that it faces a stronger adversary who controls the entire system software. The idea of using covert channels for co-location detection has been applied in prior works to achieve VM co-location in public clouds [39], [40], [41], [29]. Our method of detecting AEX follows Gruss *et al.* [30]. A very similar technique (*i.e.*, placing markers in control data structures) has been explored by Zhang *et al.* for detecting hypervisor context switches from guest VMs [42].

IX. CONCLUSION

In conclusion, HYPERRACE is a tool for protecting SGX enclaves from Hyper-Threading side-channel attacks. The main contribution of our work is the proposal of a novel physical-core co-location test using contrived data races between two threads running in the same enclave. Our design guarantees that when the two threads run on co-located logical cores of the same physical core, they will both observe data races on a shared variable with a close-to-one probability. Our security analysis and empirical evaluation suggest that the adversary is not able to schedule the two threads on different physical cores while keeping the same probability of data races that are observed by the enclave threads. Performance evaluation with *nbench* and the Intel SGX SSL library shows that the performance overhead due to program instrumentation and runtime co-location tests is modest.

X. Acknowledgments

We would like to express our sincere thanks to our shepherd Jay Lorch and the anonymous reviewers for their valuable feedback to help us improve the paper. The work was supported in part by the NSF grants 1566444, 1750809, 1527141, 1408874, 1618493, 1718084, an NIH grant 1U01EB023685 and an ARO grant W911NF1610127.

References

[1] Y. Xu, W. Cui, and M. Peinado, "Controlled-channel attacks: Deterministic side channels for untrusted operating systems," in *IEEE Symposium on Security and Privacy*. IEEE, 2015, pp. 640–656.

[2] S. Shinde, Z. L. Chua, V. Narayanan, and P. Saxena, "Preventing page faults from telling your secrets," in *11th ACM on Asia Conference on Computer and Communications Security*. ACM, 2016, pp. 317–328.

[3] J. Van Bulck, N. Weichbrodt, R. Kapitza, F. Piessens, and R. Strackx, "Telling your secrets without page faults: Stealthy page table-based attacks on enclaved execution," in *USENIX Security Symposium*, 2017.

[4] W. Wang, G. Chen, X. Pan, Y. Zhang, X. Wang, V. Bindschaedler, H. Tang, and C. A. Gunter, "Leaky cauldron on the dark land: Understanding memory side-channel hazards in SGX," in *ACM SIGSAC Conference on Computer and Communications Security*, 2017.

[5] M. Schwarz, S. Weiser, D. Gruss, C. Maurice, and S. Mangard, *Malware guard extension: Using SGX to conceal cache attacks*. Springer International Publishing, 2017.

[6] F. Brasser, U. Müller, A. Dmitrienko, K. Kostiainen, S. Capkun, and A.-R. Sadeghi, "Software grand exposure: SGX cache attacks are practical," in *USENIX Workshop on Offensive Technologies*, 2017.

[7] M. Hähnel, W. Cui, and M. Peinado, "High-resolution side channels for untrusted operating systems," in *USENIX Annual Technical Conference*, 2017, pp. 299–312.

[8] J. Götzfried, M. Eckert, S. Schinzel, and T. Müller, "Cache attacks on Intel SGX," in *EUROSEC*, 2017, pp. 2–1.

[9] S. Lee, M.-W. Shih, P. Gera, T. Kim, H. Kim, and M. Peinado, "Inferring fine-grained control flow inside SGX enclaves with branch shadowing," in *USENIX Security Symposium*, 2017, pp. 557–574.

[10] Y. Yarom and K. E. Falkner, "FLUSH+RELOAD: A high resolution, low noise, L3 cache side-channel attack," in *USENIX Security Symposium*, 2014, pp. 719–732.

[11] D. Gruss, R. Spreitzer, and S. Mangard, "Cache template attacks: Automating attacks on inclusive last-level caches," in *USENIX Security Symposium*, 2015, pp. 897–912.

[12] G. Irazoqui, T. Eisenbarth, and B. Sunar, "S$A: A shared cache attack that works across cores and defies VM sandboxing—and its application to AES," in *IEEE Symposium on Security and Privacy*, May 2015.

[13] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee, "Last-level cache side-channel attacks are practical," in *IEEE Symposium on Security and Privacy*. IEEE, 2015, pp. 605–622.

[14] Y. Yarom and N. Benger, "Recovering OpenSSL ECDSA nonces using the FLUSH+RELOAD cache side-channel attack," in *Cryptology ePrint Archive*, 2014.

[15] M. Hähnel, W. Cui, and M. Peinado, "High-resolution side channels for untrusted operating systems," in *USENIX Annual Technical Conference*, 2017, pp. 299–312.

[16] M.-W. Shih, S. Lee, T. Kim, and M. Peinado, "T-SGX: Eradicating controlled-channel attacks against enclave programs," in *Network and Distributed Systems Security (NDSS) Symposium*, 2017.

[17] S. Chen, X. Zhang, M. K. Reiter, and Y. Zhang, "Detecting privileged side-channel attacks in shielded execution with Déjá Vu," in *12th ACM on Asia Conference on Computer and Communications Security*. ACM, 2017, pp. 7–18.

[18] O. Aciiçmez, c. K. Koç, and J.-P. Seifert, "Predicting secret keys via branch prediction," in *7th Cryptographers' track at the RSA conference on Topics in Cryptology*, 2007, pp. 225–242.

[19] O. Aciicmez and J.-P. Seifert, "Cheap hardware parallelism implies cheap security," in *Workshop on Fault Diagnosis and Tolerance in Cryptography*, 2007, pp. 80–91.

[20] C. Percival, "Cache missing for fun and profit," in *2005 BSDCan*, 2005.

[21] D. A. Osvik, A. Shamir, and E. Tromer, "Cache attacks and countermeasures: the case of AES," in *6th Cryptographers' track at the RSA conference on Topics in Cryptology*, 2006, pp. 1–20.

[22] O. Aciiçmez, "Yet another microarchitectural attack: exploiting I-Cache," in *2007 ACM workshop on Computer security architecture*, 2007, pp. 11–18.

[23] O. Aciiçmez, B. B. Brumley, and P. Grabher, "New results on instruction cache attacks," in *12th international conference on Cryptographic hardware and embedded systems*, 2010, pp. 110–124.

[24] E. Tromer, D. A. Osvik, and A. Shamir, "Efficient cache attacks on AES, and countermeasures," *Journal of Cryptology*, vol. 23, no. 1, pp. 37–71, 2010.

[25] Y. Yarom, D. Genkin, and N. Heninger, "CacheBleed: a timing attack on OpenSSL constant-time RSA," *Journal of Cryptographic Engineering*, vol. 7, no. 2, pp. 99–112, 2017.

[26] "Intel 64 and IA-32 architectures software developer's manual, combined volumes:1,2A,2B,2C,3A,3B,3C and 3D," https://software.intel.com/sites/default/files/managed/39/c5/325462-sdm-vol-1-2abcd-3abcd.pdf, 2017, order Number: 325462-063US, July 2017.

[27] Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart, "Cross-VM side channels and their use to extract private keys," in *ACM conference on Computer and communications security*. ACM, 2012, pp. 305–316.

[28] B. S. Ahmad Moghimi, Thomas Eisenbarth, "MemJam: A false dependency attack against constant-time crypto implementations," arXiv:1711.08002, 2017, https://arxiv.org/abs/1711.08002.

[29] D. Sullivan, O. Arias, T. Meade, and Y. Jin, "Microarchitectural minefields: 4k-aliasing covert channel and multi-tenant detection in IaaS clouds," in *Network and Distributed Systems Security (NDSS) Symposium*, 2018.

[30] D. Gruss, J. Lettner, F. Schuster, O. Ohrimenko, I. Haller, and M. Costa, "Strong and efficient cache side-channel protection using hardware transactional memory," in *USENIX Security Symposium*, 2017, pp. 217–233.

[31] M. F. Chowdhury and D. M. Carmean, "Method, apparatus, and system for maintaining processor ordering by checking load addresses of unretired load instructions against snooping store addresses," Nov. 19 2002, US Patent 6,484,254.

[32] "Intel software guard extensions programming reference," https://software.intel.com/sites/default/files/managed/48/88/329298-002.pdf/, 2014, order Number: 329298-002, October 2014.

[33] A. Mazouz, A. Laurent, B. Pradelle, and W. Jalby, "Evaluation of CPU frequency transition latency," *Computer Science - Research and Development*, vol. 29, no. 3, pp. 187–195, Aug 2014. [Online]. Available: https://doi.org/10.1007/s00450-013-0240-x

[34] Clang: a C language family frontend for LLVM. http://clang.llvm.org/.

[35] The LLVM compiler infrastructure. https://llvm.org/.

[36] Nbench-byte benchmarks. http://www.math.cmu.edu/~florin/bench-32-64/nbench/.

[37] Intel software guard extensions SSL. https://github.com/intel/intel-sgx-ssl.

[38] Y. Zhang, A. Juels, A. Oprea, and M. K. Reiter, "HomeAlone: Co-residency detection in the cloud via side-channel analysis," in *IEEE Symposium on Security and Privacy*, 2011.

[39] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage, "Hey, you, get off of my cloud: Exploring information leakage in third-party compute clouds," in *ACM Conference on Computer and Communications Security*, 2009.

[40] V. Varadarajan, Y. Zhang, T. Ristenpart, and M. Swift, "A placement vulnerability study in multi-tenant public clouds," in *USENIX Security Symposium*, 2015.

[41] T. Zhang, Y. Zhang, and R. B. Lee, "Dos attacks on your memory in cloud," in *12th ACM on Asia Conference on Computer and Communications Security*. ACM, 2017.

[42] Y. Zhang and M. K. Reiter, "Düppel: retrofitting commodity operating systems to mitigate cache side channels in the cloud," in *ACM Conference on Computer and Communications Security*, 2013, pp. 827–838.

APPENDIX A
INTEL MICROPROCESSORS WITH SGX SUPPORT

SGX support as of Oct 2017. Note that although some of the processors support SGX, the feature may not be enabled by default by system manufacturer in UEFI. Processors marked with ⋆ have Hyper-Threading (HT) support.

TABLE VIII: Intel CPU with SGX support

Generation	Family	Model					
Skylake	Xeon	E3-1575M V5 ⋆	E3-1545M V5 ⋆	E3-1515M V5 ⋆	E3-1280 V5 ⋆	E3-1275 V5 ⋆	E3-1270 V5 ⋆
		E3-1268L V5 ⋆	E3-1260L V5 ⋆	E3-1245 V5 ⋆	E3-1240L V5 ⋆	E3-1240 V5 ⋆	E3-1235L V5
		E3-1230 V5 ⋆	E3-1225 V5	E3-1220 V5	E3-1505L V5 ⋆	E3-1535M V5 ⋆	E3-1505M V5 ⋆
	Pentium	G4400TE	4405Y ⋆	4405U ⋆	G4500	G4500T	G4520
		G4400	G4400T				
	Celeron	G3902E	G3900E	G3920	G3900TE	G3900T	G3900
		3955U	3855U				
	Core	i3-6006U ⋆	i3-6157U ⋆	i7-6785R ⋆	i5-6685R	i5-6585R	i7-6660U ⋆
		i7-6970HQ ⋆	i7-6870HQ ⋆	i7-6770HQ ⋆	i5-6350HQ ⋆	i5-6402P	i3-6098P ⋆
		i7-6822EQ ⋆	i7-6820EQ ⋆	i7-6700TE ⋆	i5-6500TE	i5-6440EQ	i5-6442EQ
		i3-6100E ⋆	i3-6102E ⋆	i3-6100TE ⋆	i7-6920HQ ⋆	i7-6820HQ ⋆	i7-6820HK ⋆
		i7-6700HQ ⋆	i7-6650U ⋆	i7-6600U ⋆	i7-6560U ⋆	i7-6500U ⋆	i5-6440HQ
		i5-6360U ⋆	i5-6300HQ	i5-6300U ⋆	i5-6200U ⋆	i5-6260U ⋆	i5-6267U ⋆
		i5-6287U ⋆	i3-6100H ⋆	i3-6167U ⋆	m7-6Y75 ⋆	m5-6Y57 ⋆	m5-6Y54 ⋆
		m3-6Y30 ⋆	i7-6700T ⋆	i7-6700 ⋆	i5-6600	i5-6600T	i5-6500
		i5-6500T	i5-6400	i5-6400T	i3-6300 ⋆	i3-6300T ⋆	i3-6320 ⋆
		i3-6100 ⋆	i3-6100T ⋆	i7-6700K ⋆	i5-6600K	i7-6567U ⋆	
Kabylake	Pentium	4415Y ⋆	G4600T ⋆	G4600 ⋆	G4620 ⋆	G4560T ⋆	G4560 ⋆
		4415U ⋆	4410Y ⋆				
	Celeron	3965Y	G3930TE	G3930E	G3950	G3930T	G3930
		3965U	3865U				
	Core	i3-7130U ⋆	m3-7Y32 ⋆	i7-7920HQ ⋆	i7-7820HQ ⋆	i7-7820HK ⋆	i7-7820EQ ⋆
		i7-7700HQ ⋆	i7-7700 ⋆	i7-7700K ⋆	i7-7700T ⋆	i7-7660U ⋆	i7-7600U ⋆
		i7-7567U ⋆	i7-7560U ⋆	i5-7600K	i5-7600T	i5-7600	i5-7500
		i5-7500T	i5-7442EQ	i5-7440HQ	i5-7440EQ	i5-7400T	i5-7400
		i5-7360U ⋆	i5-7300U ⋆	i5-7300HQ	i5-7287U ⋆	i5-7267U ⋆	i5-7260U ⋆
		i5-7Y57 ⋆	i3-7350K ⋆	i3-7320 ⋆	i3-7300 ⋆	i3-7300T ⋆	i3-7102E ⋆
		i3-7101E ⋆	i3-7101TE ⋆	i3-7100T ⋆	i3-7100E ⋆	i3-7100 ⋆	i3-7167U ⋆
		i3-7100H ⋆	i7-7500U ⋆	i7-7Y75 ⋆	i5-7200U ⋆	i3-7100U ⋆	m3-7Y30 ⋆
		i5-7Y54 ⋆					
	Xeon	E3-1285 V6 ⋆	E3-1501L V6	E3-1501M V6	E3-1280 V6 ⋆	E3-1275 V6 ⋆	E3-1270 V6 ⋆
		E3-1245 V6 ⋆	E3-1240 V6 ⋆	E3-1230 V6 ⋆	E3-1225 V6	E3-1220 V6	E3-1535M V6 ⋆
		E3-1505M V6 ⋆	E3-1505L V6 ⋆				
Coffee Lake	Core	i7-8700 ⋆	i7-8700K ⋆	i5-8600K	i5-8400	i3-8350K	i3-8100
		i7-8650U ⋆	i7-8550U ⋆	i5-8350U ⋆	i5-8250U ⋆		