Detecting Privileged Side-Channel Attacks in Shielded Execution with Déjà Vu

Sanchuan Chen
The Ohio State University
Columbus, OH, USA
chen.4825@osu.edu

Xiaokuan Zhang
The Ohio State University
Columbus, OH, USA
zhang.5840@osu.edu

Michael K. Reiter
University of North Carolina
Chapel Hill, NC, USA
reiter@cs.unc.edu

Yinqian Zhang
The Ohio State University
Columbus, OH, USA
yinqian@cse.ohio-state.edu

ABSTRACT

Intel Software Guard Extension (SGX) protects the confidentiality and integrity of an unprivileged program running inside a secure enclave from a privileged attacker who has full control of the entire operating system (OS). Program execution inside this enclave is therefore referred to as *shielded*. Unfortunately, shielded execution does not protect programs from side-channel attacks by a privileged attacker. For instance, it has been shown that by changing page table entries of memory pages used by shielded execution, a malicious OS kernel could observe memory page accesses from the execution and hence infer a wide range of sensitive information about it. In fact, this page-fault side channel is only an instance of a category of side-channel attacks, here called *privileged side-channel attacks*, in which privileged attackers frequently preempt the shielded execution to obtain fine-grained side-channel observations. In this paper, we present DÉJÀ VU, a software framework that enables a shielded execution to detect such privileged side-channel attacks. Specifically, we build into shielded execution the ability to check program execution time at the granularity of paths in its control-flow graph. To provide a trustworthy source of time measurement, DÉJÀ VU implements a novel software reference clock that is protected by Intel Transactional Synchronization Extensions (TSX), a hardware implementation of transactional memory. Evaluations show that DÉJÀ VU effectively detects side-channel attacks against shielded execution and against the reference clock itself.

CCS Concepts

•Security and privacy → Information flow control;

ASIA CCS '17 April 02-06, 2017, Abu Dhabi, United Arab Emirates

ACM ISBN 978-1-4503-4944-4/17/04.

DOI: http://dx.doi.org/10.1145/3052973.3053007

Keywords

side-channel detection; Software Guard Extension; Transactional Synchronization Extensions

1. INTRODUCTION

To reduce the trusted computing base of sensitive applications, numerous efforts have proposed systems to support *shielded execution*—i.e., application program execution for which the confidentiality and integrity is protected from an untrusted operating system (*e.g.*, [37, 10, 22, 7]). The advent of Intel Software Guard eXtension (SGX) [3], which is now commercially available, offers an opportunity for shielded execution to become mainstream. Enforced by the hardware memory management unit (MMU), shielded memory regions of userspace programs (*i.e.*, enclaves) are isolated from other software, including privileged system software—no memory read or write can be performed inside the enclave by external software, regardless of its privilege level.

A significant challenge in accomplishing shielded execution *effectively* is that the operating system (OS), though untrusted, must still be relied upon for some system services (*e.g.*, managing devices and physical memory). Previous work has shown, for example, that the untrusted OS might be able to compromise a shielded execution by manipulating the return values to system calls that it invokes [8]. Side-channel attacks from an untrusted OS have also been demonstrated against SGX-based shielded execution: *e.g.*, it has been shown that a malicious OS can force page faults on all but one or two virtual page owned by a victim SGX enclave so that memory accesses by the victim will leave a trace of page faults that could leak sensitive information [53, 45]. Unlike the vulnerabilities exploitable by malicious return values from system calls, these side-channel attacks cannot be avoided merely through defensive programming of the shielded program.

Moreover, on untrusted OSes, this page-fault side channel is only the tip of the iceberg. Page faults are simply one type of exception that can trap the execution of a software program into the OS kernel, allowing the malicious OS to trace the control flow or data flow of a shielded execution. Similar side channels can be constructed by a malicious OS using other types of exceptions or interrupts. For example, by interrupting the shielded execution using frequent hard-

ware interrupts, the malicious OS may conduct fine-grained cache side-channel attacks on private caches (*e.g.*, L1 caches) that are used by the shielded execution. This type of attack has been demonstrated in scenarios where the attackers are unprivileged—cross-process attacks [39, 20] or cross-VM attacks [58]—by exploiting design flaws in OS schedulers. On untrusted OSes, such attacks, which can be performed without scheduler vulnerabilities, are even more powerful as the privileged attacker can program hardware interrupt controllers directly to take control of the CPU at high frequency, *e.g.*, after every memory access of the shielded execution.

In this paper, we systematize a category of side-channel attacks on untrusted OSes that we call *privileged side-channel attacks*, in which privileged attackers in control of the OS frequently intervene, via software exceptions or hardware interrupts, on the shielded execution to obtain fine-grained side-channel observations. To defeat such privileged side-channel attacks, we devise an approach to allow the shielded execution to detect such attacks as it incurs them during its execution. The shielded execution can then implement an application-specific policy in response to these detections. So, for example, if the shielded execution detects unusually frequent page faults during its execution of a cryptographic routine, then the execution could abort or else refresh its keys. Our focus here is not on the policy—again, the policy will generally be application-specific—but rather on constructing a way for a shielded execution to reliably detect privileged side-channel attacks when they occur.

A key obstacle to building such a detection capability is that current SGX-enabled hardware provides no support for a shielded execution to directly observe the page faults it induces or interrupts issued by hardware. An alternative, therefore, is for the shielded execution to time its own activity to observe timings that indicate that a page fault or an interrupt occurred, as both exceptions and interrupts occurred inside enclaves will result in Asynchronous Enclave Exits (AEXs) and context switches from userspace to kernel space, which grows the execution time of the shielded execution. Unfortunately, current SGX-enabled hardware allows access to timers only through the untrusted OS, rendering these timers untrustworthy, as well. As such, the central challenge is for the shielded execution to itself implement a *reference clock* against which to time steps of its own execution, to detect when exceptions or interrupts occurred during one of those steps.

Following Wray [52], one approach to constructing such a reference clock is by using execution itself as a clock—*i.e.*, to measure the time between events by the distance that an execution progresses between those events. While Wray envisioned this capability as a means to *implement* timing side channels, here we use this idea defensively, to detect when steps of the shielded execution take too long in comparison to a reference execution. That is, our goal is to construct a shielded execution consisting of threads that execute concurrently, one serving as the "reference clock" to measure the time taken for each step of the other thread.

Of course, this design begs the question of how to detect if the reference-clock thread incurs a delay due to a page fault or interrupt (again, possibly induced by the untrusted OS). For this purpose, we leverage another capability of modern Intel platforms, namely Transactional Synchronization Extensions (TSX), a hardware implementation of transactional memory. Specifically, our design advances its reference clock using transactional memory, in a way that an interrupt or page fault will cause a transaction to abort with high probability. This transaction abort will be visible to the reference-clock thread, allowing it to determine when it was "paused" by the OS. Of course, this is not the only threat that the OS poses to the reference clock—*e.g.*, the OS might change the execution speed of the processor core on which it executes. However, we show that with careful engineering, we can implement an execution-based reference clock within SGX-based shielded execution, for detecting the interruption of (and thus potential leakage from) critical routines within the shielding.

We have realized our design in an implementation for Linux called DÉJÀ VU.[1] Specially, DÉJÀ VU is implemented as an extension of the LLVM framework [28], which takes the source code of the shielded program as the input and outputs an instrumented binary to be loaded into SGX enclaves at runtime. The execution time of the shielded execution is measured by the instrumented code at selected basic blocks, which employ threshold-based classifiers to determine whether the measured execution time exceeds the expected values—longer execution between basic blocks suggests AEXs have occurred during the execution.

We have integrated DÉJÀ VU with the Intel Linux SGX SDK. The security evaluation shows DÉJÀ VU achieves at least 0.94 precision and recall in detecting AEXs of the application thread, and 0.95 or better recall and 0.78 or better precision in detecting AEXs on the reference-clock thread. The runtime performance overhead of DÉJÀ VU when applied to `nbench` applications [4] is typically less than 4%.

Contributions. In summary, this paper contributes to the field of study in the following aspects:

- A software framework, DÉJÀ VU, for automated program transformation and detection of privileged side-channel attacks against shielded execution with good accuracy.
- A software-based reference clock that is protected by Intel's hardware transactional memory, which cannot be manipulated by the malicious OS without being noticed.
- Implementation and evaluation of DÉJÀ VU on Intel Skylake processors with both SGX and TSX features.

2. BACKGROUND AND RELATED WORK

2.1 Intel SGX and Shielded Execution

There is a long history of research on shielded execution on untrusted operating systems [31, 46, 9, 10, 37, 54, 42, 56, 22, 11, 14, 30]. Most rely on a trusted software component, usually a hypervisor running at the highest privilege, to protect the memory of an isolated program against both confidentiality and integrity attacks from an untrusted guest operating system.

In contrast to these software-based approaches, Intel Software Guard eXtensions (SGX) is a new hardware feature provided in the most recent Intel processors (*e.g.*, Skylake processor families) that protects a *shielded* memory region from reading and writing by external software regardless of its privilege levels [3]. This security mechanism provides

[1] DÉJÀ VU is a reference to the movie *The Matrix*, in which the world as we know it is a simulation. The simulation has glitches, which are revealed when one experiences déjà vu. A transaction abort in our reference-clock thread is an analogy for this déjà vu, or evidence of a "glitch" in the virtual world.

software developers with an unprecedented capability to develop security-critical software programs that can achieve strong security guarantees (in terms of confidentiality and integrity) even under the assumption of a malicious operating system.

2.2 Transactional Synchronization Extensions

Intel Transactional Synchronization eXtensions (TSX) is a hardware implementation of transactional memory that is available in recent Intel processors (including Skylake models that have SGX enabled). Transactional memory (TM) [21] enables atomic execution of a set of memory read and write instructions on shared memory regions by concurrent threads, without the overhead of software locks.

Although not designed as a security enhancement, Intel TSX has been leveraged to improve security in several works. Most directly related to our work, concurrent research by Shih *et al.* [44] employs TSX to address controlled-channel attacks on SGX by leveraging, as we do here, the fact that an exception during the execution of a transaction in an enclave will abort the transaction, notifying the enclave program without interference by the system software. We will compare to their solution, called T-SGX, in Sec. 3.4. In more distant security-related research, Liu *et al.* [34] demonstrated the use of Intel TSX to facilitate virtual machine introspection. Guan *et al.* [19] explored the use of TSX to protect cryptographic keys in CPU caches to prevent memory disclosure attacks.

In terms of usability of Intel TSX, it has been shown in these prior studies [34, 19] that enclosing large code regions inside TSX transactions may induce numerous transaction aborts, degrading the performance of their applications and even making it unusable. Our use of Intel TSX avoids such issues by only enclosing a small loop inside the transaction, which significantly reduces the likelihood of transaction aborts due to regular system operations. Therefore, our design is highly practical.

2.3 Side-Channel Attacks and Defenses

Besides exception-based side-channel attacks, cache-based side-channel attacks have also been demonstrated to be capable of tracing the victim program's code execution and data accesses. A cache-based side channel can be constructed using data caches (including per-core L2 unified caches) on shared processors [39, 47, 20, 23], instruction caches [5, 58], shared last-level caches (LLC) [55, 59, 33, 18, 40, 25]. Similar to page fault side-channel attacks, cache side-channel attacks conducted on per-core caches also require frequent preemption of the victim program's execution [39, 20, 58]. However, the page fault side-channel attacks enforce CPU preemption by inducing page faults, which leads to better synchronization with the victim program and less noise.

Defenses against these side-channel attacks are not easy. Modifying the processor architecture [50, 36, 15, 32] and relying on system-level isolation enhancement [49, 27, 29, 60, 48, 57] are not possible in the settings we consider. Software defenses against cache-based side-channel attacks may be applicable to our settings, however. Ideally, if the shielded execution can be transformed so that it does not have secret-dependent side-effects on memory pages or caches, both cache-based and exception-based side channels can be eliminated. However, doing so in practice is extremely labor-intensive. Automated program transformation to eliminate the secret-dependent control flows or data flows have been attempted in several prior works [38, 12, 35, 13, 43], but due to the high performance overhead (*e.g.*, several times higher runtime), these compiler-based approaches have not yet been widely adopted. Static analysis of software programs to automatically detect side-channel vulnerabilities have been studied by Doychev *et al.* [16]. To enable such analysis for exception-based side-channel attacks, new models need to be constructed.

3. PRIVILEGED SIDE-CHANNEL ATTACKS ON SGX ENCLAVES

Prior studies on side channels by an attacker with a foothold on the same machine as the victim usually consider unprivileged attackers, *e.g.*, virtual machines confined by hypervisors (*e.g.*, [58, 55, 24, 33]), non-root userspace processes (*e.g.*, [41, 39, 5, 6, 47, 59]), or sandboxed Javascript code (*e.g.*, [40]). In contrast, much less is known about side-channel attacks from privileged attackers. This is primarily because privileged attackers, *i.e.*, attackers who control privileged software components, are capable of conducting more direct attacks than side-channel attacks.

As described in Sec. 2, Intel SGX provides hardware-level memory isolation of userspace programs running inside enclaves with an enhanced memory-management unit (MMU), as well as hardware-facilitated encryption of memory when it is not protected by the MMU. As such, even the most privileged software attacker that controls the entire operating system cannot inspect the memory space inside the enclave. Side-channel threats thus become primary attack vectors against SGX-protected programs.

Although the malicious OS cannot access memory inside the enclaves, it still controls the scheduling of CPU resources, mediates accesses to I/O devices, handles interrupts and exceptions, maintains process control blocks and page tables, *etc.* Of particular concern here is the OS' responsibilities in handling exceptions and interrupts, which allows the malicious OS to intercept the control flows of the shielded execution in ways that can be controlled by the attacker.

3.1 Exception-based Attacks

Exceptions triggered during the execution inside an enclave will be captured first by the untrusted OS before being forwarded to the enclave program. As such, these exceptions can be exploited by the adversary to keep track of the activities of the victim program's execution. Even worse, in some cases, the untrusted OS can *cause* the enclave execution to induce exceptions that give it considerable insight into the enclave's activities. For instance, Xu et al. [53] showed that by modifying page table entries of pages inside an enclave, the untrusted OS can cause enclave execution to incur an exception upon each new page access, permitting the OS to trace the enclave execution (at the granularity of pages as SGX masks the lowest 12 bits of the page-fault address before passing it to the OS kernel) and gain considerable information about secret data it holds. For instance, it was shown that the code and data page access patterns of the `freetype` font rendering engine, the Hunspell spell checker, and the image processing library `libjpeg` can be monitored through the page-fault side channel, which may be exploited to infer the content of the documents or images. Shinde *et al.* [45] demonstrated similar page-fault at-

tacks against cryptographic code, *e.g.*, the EdDSA signing process in `libgcrypt`. By tracing the execution sequence of the three functions `ec_mul`, `add_point` and `test_bit`, which are located in three separate pages, the adversary can learn the signing key.

3.2 Interrupt-based Attacks

It has been shown in prior studies [20, 58] that an unprivileged attacker may generate frequent interrupts to preempt the victim program's execution, by exploiting design flaws of the underlying OS or hypervisor schedulers. The frequent preemption enables fine-grained cache side-channel attacks, *e.g.*, PRIME-PROBE attacks, on the L1 data cache and instruction cache to extract encryption keys. A privileged attacker who controls the entire OS may also program hardware interrupt controllers to trigger even more frequent interrupts (*e.g.*, one interrupt per instruction) without exploiting any vulnerability of the schedulers. For example, the high-precision Advanced Configuration and Power Interface (ACPI) Power Management Timer, the High Precision Event Timer (HPET), the Local Advanced Programmable Interrupt Controller (LAPIC), and the hardware performance monitoring units (PMUs) can all be programmed to trigger frequent interrupts to preempt the shielded execution with high frequency.

Not only can privileged side-channel attackers trigger more frequent PRIMEs and PROBEs by instructing hardware interrupt controllers to trigger high-frequency interrupts, they can also reduce the background noise that are major obstacles in unprivileged cache side-channel attacks by, *e.g.*, pinning all other processes on the system to other CPU cores to make sure no other processes are scheduled on the core where the attack is performed. Moreover, as the adversary also has full control over the hardware configuration, he might also disable hardware features that make his cache attacks challenging. For instance, by disabling hardware cache prefetching, the attacker can considerably reduce noise in cache-based side-channel attacks.

Whereas page-fault side channels trace the execution of the shielded code at the page level and cache side channels trace shielded execution at cache-set granularity, it is also conceivable that a malicious OS may combine page-level side channels and cache-set-level side channels to trace the execution of the shielded code at the granularity of 64-byte cache lines, enabling much more fine-grained observations than any previously known attacks.

3.3 Threat Model

To summarize, we consider privileged side-channel attacks against shielded execution in SGX enclaves. The operating system is untrusted and possibly malicious, but its direct inspection of enclave memory is prohibited by SGX. We specifically consider a privileged side-channel attacker who may induce page faults or trigger interrupts to preempt the shielded execution and trace its control flow or data flow at the page level or cache-line level. We assume the attacker's side-channel observations are noise-free and deterministic.

We also assume that the untrusted OS is willing to provide an execution environment to shielded execution that is free of excessive interrupts, *e.g.*, by pinning the shielded execution to dedicated CPU cores. We consider this requirement a contract between the operating system and the shielded execution. Violation of the contract, detected using the methods we lay out in this paper, will result in self-termination of the shielded execution (assuming the shielded program is configured with policy to do so), which leads to denial-of-service (DOS). DOS attacks are beyond the scope of the paper, as a malicious OS can easily do so by not scheduling the shielded execution.

3.4 Known Defenses

Closely related to DÉJÀ VU is concurrent research by Shih *et al.* [44]. Their solution, called T-SGX, works by compiling the enclave application into a collection of TSX transactions, so that page faults are handled by the transaction abort handler first, before trapping into the kernel, effectively hiding faulting addresses from the kernel. A side effect is that exceptions and interrupts are detectable (by causing these transactions to abort). Doing so, however, requires detailed static analysis and a number of program transformations to dissect the enclave program into short transactions subject to hardware-specific constraints (*e.g.*, that its write and read sets fit into the L1 or L3 caches, respectively). These larger write and read sets will tend to cause more transaction aborts than our simple reference-clock code. And, perhaps most importantly, T-SGX precludes the use of TSX for its intended purpose: eliding software locks in multithreaded programs. Our approach is based on similar principles, but neither requires detailed static analysis nor imposes on enclave programs' other uses of TSX. Aside from T-SGX, the only other software-based defense known to us for page-fault side channels involves hiding the pattern of page accesses from the untrusted OS, which addresses *only* the page-fault side channel (versus also cache-based side-channels leveraging interrupts), has been demonstrated only on cryptographic libraries, and required manual program annotation to achieve reasonable overheads [45].

4. ATTACK DETECTION THROUGH TIMED EXECUTION

As discussed in Sec. 3, a significant vantage point that a privileged attacker has is the ability to frequently preempt the shielded execution in enclaves, resulting in unexpected enclave exits—*i.e.*, Asynchronous Enclave Exits (AEXs) in SGX—that the untrusted OS can observe. While not every AEX is necessarily indicative of an attack, a higher frequency of AEXs than normal, particularly at certain points in an execution, can serve as a signature for such an attack. Therefore, we aim to detect AEXs as an indicator of side-channel attacks.

This seemingly simple task in practice is very challenging: Shielded execution is not notified by CPU hardware when AEXs occur. Because the shielded program cannot rely on the untrusted OS for such information, there is no reliable source that allows the shielded program to detect AEXs. Therefore, the only viable solution is to detect them itself. DÉJÀ VU does so by implementing a trustworthy reference clock to measure the execution time of its application thread in the enclave and compare the execution time of its steps with their normal execution times to detect whether enclave exits occurred and, if so, during which steps they occurred.

We design a mechanism that embeds time measurement into the program's execution—the shielded program will periodically check a reliable clock to measure its own execution time with fine granularity. If during any period of time, the

execution time deviates from the expectation by a threshold, the shielded application program obtains evidence of AEXs during its execution. To make the scheme secure, the time measurement code must be executed inside the enclave. For the moment, assume that there is a trustworthy clock inside the enclave for time measurement. (As we will discuss in the next section, such an assumption is not easy to satisfy.)

DÉJÀ VU statically instruments enclave programs at compile time to incorporate time-measurement code that is used to detect variations in the execution times of execution paths in the instrumented enclave programs due to both page faults and interrupts (and, indeed, any AEXs). The key insight of DÉJÀ VU is that by injecting the time-measurement code at the basic-block level in the control-flow graph (CFG), the final compiled binary code, when run inside the enclave, will repeatedly reference a real-time clock during its execution along paths of the CFG.

The execution time of each path in the CFG is then measured to detect unexpected enclave exits; *i.e.*, if the execution time is too long, then it is quite likely that an AEX occurred during the execution. Although the design seems straightforward, in practice it is challenging to make the method both effective and efficient. In the rest of this section, we gradually explain the technical difficulties we faced and incrementally describe our solutions to these challenges.

4.1 Sources of Time

To measure the execution time with fine granularity, DÉJÀ VU constructs a reference clock that satisfies the following requirements:

- The clock provides an interface to a monotonically non-decreasing counter.
- The counter value of the clock cannot be read or altered by the untrusted OS.
- The clock cannot be silently stopped by the untrusted OS without being noticed; referencing a clock that has been stopped will return a failure indicator.

As we will show in Sec. 5, such a reference clock cannot be achieved easily inside enclaves. We will elaborate on our design and implementation shortly. For the sake of discussion in this section, we simply assume the existence of such a reference clock.

4.2 Selective Basic Block Instrumentation

Instrumenting all basic blocks for time measurements may induce high performance overhead. So, DÉJÀ VU only selectively instruments a subset of all basic blocks. Specifically, DÉJÀ VU instruments only two types of basic blocks:

- *multi-sinks:* A multi-sink is a basic block with indegree larger than 1 in the CFG, or that is an entry block or exit block.
- *multi-sink predecessors:* A multi-sink predecessor is a basic block that has an edge to a multi-sink in the CFG.

By definition, a basic block can be both a multi-sink and a multi-sink predecessor. DÉJÀ VU instruments multi-sinks with the time measurement logic; specifically, the clock is referenced to get the current value of the clock counter, and the time increment since the last time measurement (*i.e.*, the last time a multi-sink was visited in the CFG) is computed by subtracting the last clock reading from the current clock reading. A multi-sink predecessor is not instrumented to measure time (unless it is also a multi-sink), but instead each is instrumented to record the fact that it was just traversed, by logging its basic-block identifier for that purpose.

Let an *execution pathlet* between two multi-sinks in a CFG be a path of basic blocks between the two multi-sinks that does not contain another multi-sink. Note that there is a one-to-one correspondence between each multi-sink predecessor in the CFG and the execution pathlet that traverses it and reaches its successor. This is due to the fact that only two basic blocks on an execution pathlet have indegrees larger than 1. As such, in the instrumentation of a multi-sink, it suffices to only check the multi-sink predecessor last traversed to determine which pathlet was just followed. Therefore, a multi-sink can use the identifier of this predecessor to look up the threshold to which to compare the time taken to traverse this pathlet to detect an AEX during its execution.

We illustrate these basic blocks in Fig. 1. In this figure, four types of basic blocks are shown: basic block 1 and 6 are multi-sinks, basic block 3 and 4 are multi-sink predecessors, and basic block 5 is both a multi-sink and a multi-sink predecessor. Basic block 2 is not instrumented. Four execution pathlets are shown in the figure: two pathlets from basic block 1 to 5, one from 1 to 6, and one from 5 to 6. The two pathlets between block 1 and 5 can be distinguished at basic block 5 by checking the predecessor.

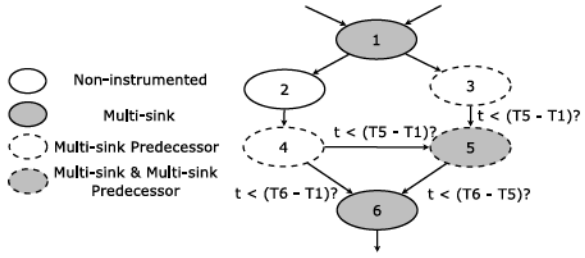

Figure 1: Examples of multi-sinks, multi-sink predecessors and execution pathlets.

4.3 Detecting AEXs

To detect AEXs at runtime, DÉJÀ VU first trains a classifier offline. The classifier is trained using measured execution times (using the reference clock of Sec. 5) of each execution pathlet, each labeled according to whether AEXs were taken during the execution of that pathlet or not. Provided that these times are easily separable by a threshold (and we will show that they are), the classifier will apply that threshold to each execution time of an execution pathlet to determine whether an AEX was taken during its execution.

As the execution time is measured per pathlet, training of the threshold-based classifier is independent of the input. This is because indirect control flow transfers (*e.g.*, loops) do not exist on any pathlet. The execution time without AEXs is only dependent on the set of instructions on the pathlet and the runtime interaction with CPU caches. In practice, we find the variation of execution time due to cache misses and hits is negligible compared to the time consumed by an AEX. When the training data is not sufficient to cover all execution pathlets in the CFG, which is a limitation of any dynamic training scheme, a default threshold is used for the baseline classifier. The default threshold is conservatively estimated by the minimum time it needs to take for one

AEX. We will detail our implementation of the detection mechanisms in Sec. 6.

4.4 Responding to AEX Detections

Given the wide variety of applications, it is unlikely that there is a single best policy for responding to AEX detections across all of them. However, in the context of specific applications, some effective response policies are evident. For example, in the page-fault attack of Shinde *et al.* [45] against `libgcrypt`, the attacker leveraged page faults to determine the path taken through the `ec_mul` routine; since the path taken is determined by an exponent bit, determining the path determines the exponent bit (*i.e.*, the signing key bit). Under normal operation, a high rate of AEXs during a modular exponentiation routine would be very unlikely. As such, even only a couple of detections of AEXs during the execution of this routine would already be indicative of an attack occurring. In response, the application could simply halt the exponentiation to protect the key, or even change its cryptographic key. However, our goal here is not to provide response policies for many different applications, but rather to provide the underlying capability to reliably detect AEXs during the execution of an application thread in an enclave.

5. CREATING A REFERENCE CLOCK

As discussed in Sec. 4, the central innovation in DÉJÀ VU is in how it constructs a reference clock with which enclave logic can measure execution of pathlets of the application thread in the enclave, in order to determine whether the application thread incurs page faults. While numerous facilities exist on a computer platform to support measuring execution time, unfortunately none of them work in our threat model:

- *Operating system clocks:* Operating systems provide fine-grained software clocks, such as `clock_gettime()` and `gettimeofday()`, but as they are all maintained by the untrusted OS kernel, none of them can be trusted to provide faithful measurement of the passage of time.
- *Hardware clocks and timers:* Fine-grained hardware time stamp counters can be accessed from userspace through the `rdtsc` instruction. However, this instruction can be emulated by the operating system so that the returned value is controlled by the untrusted OS [2]. In addition, the current SGX implementation, SGX v1.0, does not support running `rdtsc` instructions inside enclaves. Similarly, APIC timers and performance counters may also provide fine-grained time measurement by delivering interrupts at specific time intervals. However, they can be controlled by the malicious OS.
- *Remote clocks:* The enclave could communicate through a secure channel (*e.g.*, over TLS/SSL) to a remote timing facility. However, since the IP stack is still maintained by the untrusted OS, requests to remote clocks can be arbitrarily delayed, and without a local clock reference, the enclave program has no way to validate whether the returned value from the remote clock is recent.

As discussed in Sec. 1, we therefore implement our reference clock using a separate thread in the SGX enclave, *i.e.*, in which progress of the application thread is measured by progress of the reference-clock thread. We first refine our goals for our clock implementation in Sec. 5.1 and then describe its design in Sec. 5.2–5.4.

5.1 Design Goals

Denote the value of our reference clock at real time t by $C(t)$. Our goal is to implement a clock that behaves as follows for any $t_1 \geq t_0$:

$$\left\lfloor \frac{f_{\min} \times (t_1 - t_0)}{v} \right\rfloor \leq C(t_1) - C(t_0) \leq \left\lfloor \frac{f_{\max} \times (t_1 - t_0)}{v} \right\rfloor \quad (1)$$

Here, v is a parameter to our algorithm that is the number of CPU cycles that defines a "clock tick". $f_{\min}$ and $f_{\max}$ are the minimum and maximum CPU core frequencies (GHz), respectively, to which the untrusted OS can set the core running the reference-clock thread. $f_{\min}$ and $f_{\max}$ can be obtained from processor specifications. The fact that the untrusted OS can manipulate the frequency of the CPU core on which the reference-clock thread runs, accounted for by the inclusion of $f_{\min}$ and $f_{\max}$ in (1), means that the untrusted OS has some latitude to manipulate our clock to its advantage. As we will show on our test platform, however, for intervals $[t_0, t_1]$ that are characteristic of how we use the clock, this latitude is still inadequate for the attacker to avoid the enclave detecting a page fault incurred by the application thread during $[t_0, t_1]$.

One exceptional case in which our clock will not meet condition (1) is if the untrusted OS interrupts the reference clock thread in $[t_0, t_1]$. Fortunately, it suffices for our purposes to ensure condition (1) if the reference-clock thread is not interrupted during $[t_0, t_1]$ and, if it is interrupted, to detect that interruption with high probability and set a flag indicating the interruption.

5.2 Intel TSX

To detect the interruption of the reference-clock thread by the untrusted OS, DÉJÀ VU needs to leverage hardware contracts that even system software cannot break. Our choice for such a hardware contract is hardware transactional memory, specifically Intel TSX.

Intel TSX implements extensions to support atomic operation of critical regions of software. Once a program enters a critical region enclosed within a hardware transaction, modifications to data read by the critical region or fetching of data modified within the critical region will cause the transaction to abort, after which the memory will roll back to a state before entering the transaction. A feature that is particularly of interest to the design of DÉJÀ VU is that a transaction will abort if it is interrupted by the operating system. This abort is enforced in hardware; even the most privileged system software cannot avoid it [1]. Therefore, by enclosing the execution of the reference-clock thread inside a transaction, DÉJÀ VU can guarantee that interruption of the thread will not go undetected: If the malicious OS attempts to preempt the reference-clock thread, either by delivering hardware interrupts or by generating a system-level exception, the interrupted transaction will abort immediately, which will be detected by the reference-clock thread (by it following a fallback execution path).

5.3 Detailed Design

Our reference-clock thread continuously updates an enclave variable `timer` that represents the current clock time; the application thread consults this `timer` to get the current clock time $C(t)$ when needed. Because `timer` is protected by SGX in the enclave, the untrusted OS cannot read or

alter it, and so its value cannot be directly manipulated by the untrusted OS.

One subtlety in this design is that because the `timer` will be frequently read by the application thread, it cannot be included in the write-set of a hardware transaction; otherwise the transaction will abort whenever `timer` is read. Therefore, `timer` can be updated only outside transactions, implying that detecting interruption of the reference-clock thread will be (only) probabilistic. Moreover, there is a tension between the frequency with which the reference-clock thread updates `timer` and the probability of it detecting its interruption: On the one hand, updating the counter frequently implies a lower ability to detect an interruption, since the update requires executing outside the protection of hardware transactions. On the other hand, executing within a hardware transaction longer increases the likelihood of detecting an interruption, but prevents updating `timer` frequently, meaning its granularity will suffer.

To balance this tension, we randomize the number of cycles that the reference-clock thread executes within a transaction prior to updating `timer`; see Fig. 2. More specifically, the reference-clock thread runs in an endless loop: it first obtains a pseudo-random number from the hardware by issuing `rdrand` instruction at the beginning of each loop, keeps its least significant bits and adds one (line 10), yielding a pseudo-random value `rand` $\in [1, N]$, where N is a power of 2. In the example shown in Fig. 2, N is 8. At the end of each loop, the reference-clock thread increments `timer` by `rand`. The thread executes an inner loop of v cycles in duration to make sure the execution of each outer loop is `rand` $\times\, v$ cycles. To leverage the desired property of hardware transactional memory, the reference-clock thread encloses the inner loop inside a transaction (lines 7-14), which guarantees that the execution of the inner loop cannot be disrupted without being detected. Otherwise, a counter is incremented to indicate the interruption (line 16).

One consequence of this randomization is that this clock implementation can *lag* by up to N ticks; *i.e.*, the left-hand inequality in (1) must be weakened to

$$\left\lfloor \frac{f_{\min} \times (t_1 - t_0)}{v} \right\rfloor - N \leq C(t_1) - C(t_0)$$

However, the probability of a substantial lag is small, assuming `rand` is uniformly distributed in $[1, N]$, *i.e.*,

$$\begin{aligned} &\mathbb{P}\left(C(t_1) - C(t_0) + x < \left\lfloor \frac{f_{\min} \times (t_1 - t_0)}{v} \right\rfloor \right) \\ &\leq \mathbb{P}(\texttt{rand} > x) \\ &= 1 - \frac{x}{N} \end{aligned}$$

5.4 Side-Channel Inferences on Clock References

Because `timer` cannot be written inside the transaction of lines 7–14 of Fig. 2, the untrusted OS could induce a page fault on the page containing `timer` to detect whenever `timer` is accessed. In this way, the untrusted OS could measure the real time between accesses to `timer` and use them to infer what pathlets the application thread in the enclave executes.

To address this threat, DÉJÀ VU accesses a variable in the same virtual page as `timer` within the transaction (in line 12).[2] In this way, if the untrusted OS induces a page fault when that page is accessed (*e.g.*, in an effort to perform the timing attack described above), then it aborts the transaction with high probability, causing the interruption to be detected.

```
1  unsigned int timer; // global variable
2  unsigned int interrupted; // global variable
3  void timer_thread()
4  {
5    unsigned int rand;
6    while (1) {
7      if (_xbegin() == _XBEGIN_STARTED) {
8        __asm volatile ("rdrand %0\n\t"
9          :"=r"(rand));
10       rand = (rand & 0x7) + 1;
11       for(int i = 0; i < rand; i++) {
12         // tasks comprising v cycles
13       }
14       _xend();
15     } else {
16       interrupted += 1;
17       continue;
18     }
19     timer += rand
20   }
21 }
```

Figure 2: Code snippet for the reference-clock thread.

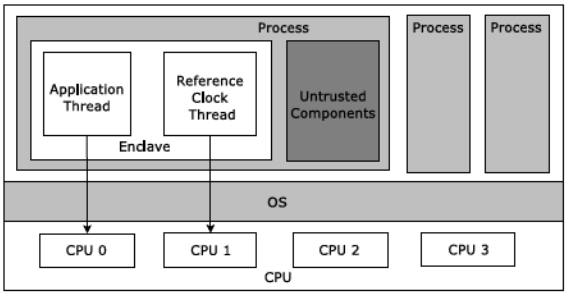

Figure 3: System architecture of DÉJÀ VU. Blocks in gray are untrusted, which include the untrusted components of the processes and the entire OS kernel.

6. IMPLEMENTATION

The runtime architecture of DÉJÀ VU is illustrated in Fig. 3. The application logic that runs inside the SGX enclave is contained inside one or more application threads, which are accompanied by the reference-clock thread within the same enclave. The threads are bound to dedicated CPU cores by the untrusted OS. Failure of the untrusted OS to do so will cause these threads to suffer more interruptions, which will be detected by DÉJÀ VU and might result in the shielded execution terminating itself (if its policy is to do so). If the application itself is multi-threaded, only one reference-clock per enclave is needed to protect all threads.

We implemented DÉJÀ VU by extending the LLVM framework (v3.5.2). The workflow of DÉJÀ VU is illustrated in Fig. 4. Specifically, a DÉJÀ VU library implements the reference-clock thread and the code specifying the security policy that reacts to AEX detections. The DÉJÀ VU library code is compiled using the `gcc` compilerto an object file. The

[2]To confirm that another variable at virtual address *vaddr* lies in the same virtual page as `timer`, it suffices to check that $\lfloor vaddr/pageSize \rfloor = \lfloor \&\texttt{timer}/pageSize \rfloor$, where *pageSize* is the smallest page size allowed by the processor in bytes.

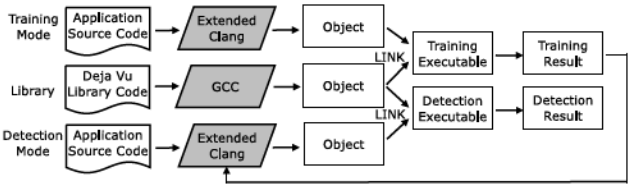

Figure 4: Workflow of DÉJÀ VU.

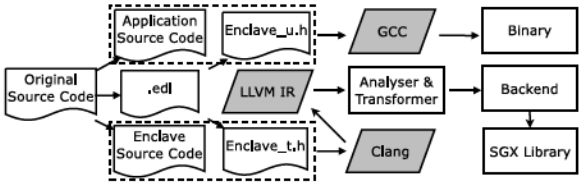

Figure 5: Integrating DÉJÀ VU with Linux SGX SDK.

application source code is compiled using `Clang`[3], a front-end to LLVM that compiles C code into LLVM intermediate representation (IR). Our static analysis tool is implemented as an LLVM IR optimization pass that instruments selected basic blocks according to our criteria. The instrumentation conducted by our extended `Clang` is different in the *training mode* and *detection mode*, which will be explained further in Sec. 6.1 and Sec. 6.2. The output of the training mode contains classifier thresholds, which will be used in monitoring the shielded execution in the detection mode.

6.1 Training Mode

Our extended `Clang` in the training mode instruments every multi-sink by adding instructions to read the reference clock, so that the execution time of each execution pathlet will be measured by the training executables. It also instruments every multi-sink predecessor to record the basic-block identifier, which is used to indicate which pathlet is taken. The training executables are given a set of input values and are run for multiple times. During the execution, the instrumented code records the time measurement of running every pathlet in a large array and prints the records into files at the very end of the training. The training file is post-processed to obtain the mean and standard deviation of the time measurement of each pathlet.

To measure the time spent in AEXs, we conducted the following experiments. Recall that the time spent in an AEX includes the time for an enclave exit and a context switch into the kernel. So, to conservatively estimate the minimum time needed by a malicious AEX, we measured only the time for enclave exits, which can be approximated by the time spent in empty OCalls. As such, we tried to measure the time (in the v-cycle time units of our own reference clock, where v is about 30) taken in an empty OCall by calling it 1000 times. On our test platform, the average time taken was 78.14 time units and the standard deviation was 3.27 time units.

The threshold of the execution time measurement of each execution pathlet is determined as the mean of the execution time of the underlying pathlet minus its standard deviation plus the mean of the time for one AEX minus its standard deviation. The resulting configuration file is then used by the extended `Clang` in the detection mode.

6.2 Detection Mode

The instrumented basic blocks in the detection mode work in the following way. At each multi-sink predecessor, the basic-block identifier is recorded to keep track of the current execution pathlet. At each multi-sink, a reference to the clock is made. If the clock was interrupted in the time since the last clock reference, a call to the DÉJÀ VU library is made to indicate a *clock-AEX alarm*. If not, then the current clock reading is compared with the previous clock reading to calculate the difference, which is the time spent to execute the just-finished pathlet. If the execution time is larger than the instrumented threshold, then a call is made to the DÉJÀ VU library to indicate an *app-AEX alarm*. Both clock-AEX alarms and app-AEX alarms can be used by the security policy engine to make informed decisions about whether a privileged side-channel attack is ongoing.

6.3 Integration with SGX Software Development Environments

The design of DÉJÀ VU is independent of the software development environment. For demonstration purposes, we have integrated DÉJÀ VU with the official Linux SGX SDK released by Intel[4]. The workflow for integrating DÉJÀ VU with the official Linux SGX SDK is shown in Fig. 5. Following the standard use case of Intel SGX described by the SDK, the source code of the project to be protected by DÉJÀ VU is separated into two components: an application component and an enclave component. The separation is enabled by the SGX SDK with an `edl` file, which is manually created by the developer and specifies which files and functions are to be compiled into which components. With the help of an SDK-provided tool called `edger8r`, two header files are generated that help the two components to interact with each other: `Enclave_u.h` and `Enclave_t.h`. The standard SDK compiles the application source code using the `gcc` compiler. DÉJÀ VU leaves this part unchanged. To enable program analysis and instrumentation, we replace the compiler for the enclave source code (*i.e.*, `gcc`) with `Clang`. The compiled binary is the SGX library that will be loaded into the enclave.

7. EVALUATION

7.1 Experiment Setup

Our experiments were conducted on a Dell Latitude E5470 laptop, which is equipped with an Intel CORE i5-6440HQ Skylake processor that supports both SGX and TSX extensions. The processor had four cores, whose maximum frequency is 2.6GHz. The laptop was equipped with 8GB DRAM. The size of EPC was the default, 128MB. The operating system was a Ubuntu 14.04 with Linux kernel version 3.19.0. To perform security and performance evaluations, we ported the `nbench` performance benchmark [4] to support SGX and applied DÉJÀ VU to compile them with the SGX Linux SDK.

7.2 Security Evaluation

We consider three types of attacks against DÉJÀ VU in our empirical security evaluation: (1) stopping the clock by

[3] http://clang.llvm.org/

[4] https://github.com/01org/linux-sgx

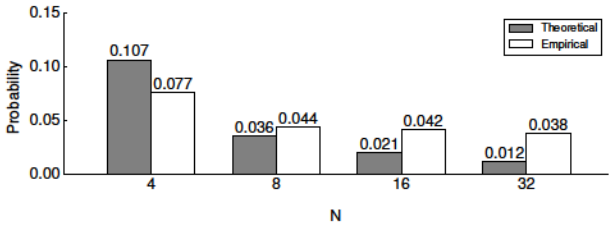

Figure 6: Probability of clock-thread AEX not causing a TSX abort. Grey bars show theoretical probability estimates and white bars represent the empirical probabilities.

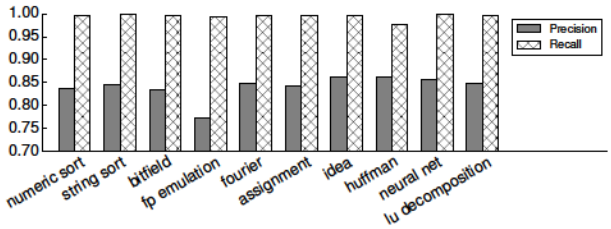

Figure 7: Precision and recall of clock-AEX alarms during ten `nbench` applications.

interrupting the execution of the reference-clock thread; (2) tracing the shielded execution by triggering page faults; and (3) slowing the reference-clock thread by scaling down the operating frequency of the underlying CPU core.

7.2.1 *Detecting Reference-Clock AEXs*

We designed the reference clock so that the probability for the adversary to preempt the clock without inducing a TSX abort is only about 5%, by tuning the value of v, the fixed CPU cycles taken in the inner loop, and N, the upper-bound of the variable `rand`. That is, the reference-clock thread will spend about 5% of the time outside the hardware transactions, to execute the instruction `timer+=rand`.

We validate that our theoretical estimates are consistent with empirical results in the following experiments. In each of the experiments we report in Fig. 6, we instrumented the OS kernel to trigger interrupts to preempt the reference-clock thread, inducing roughly 5000 AEXs on that thread. The reference-clock thread was implemented with a fixed unit execution time, $v \approx 30$ CPU cycles, but we varied the value of N in each of the experiments. The gray bars in Fig. 6 represent our theoretical estimates of the probability that an AEX will *not* induce a TSX abort on the clock thread (calculated for the choice of v and N in the experiment), and the white bars represent the empirical probabilities that an AEX did not induce a TSX abort in the experiments. The empirical probability was calculated by comparing the time stamps of transaction aborts (read in an OCall from the transaction abort handler) and AEX time stamps read in the Asynchronous Exit handler Pointer (AEP), which is a piece of code that is called when an AEX resumes, right before re-entering the enclave. If the transaction abort took place shortly (less than 5000 cycles) after the AEX, then it is considered to be caused by the AEX. The reported empirical probability is the ratio of the number of AEXs that did not cause an abort (*i.e.*, were not followed by a TSX abort in the next 5000 cycles) to the total number of AEXs.

By comparing these values, we can see that our theoretical estimation is close to the empirical probability when N is small (*e.g.*, 4 or 8), but when N increases, the theoretical value of the vulnerable window in which the reference-clock thread can incur an AEX without causing a TSX abort drops faster than the empirical results suggest. We conjecture that the result has to do with our inability to accurately measure the execution time of one single statement outside the transaction due to the CPU's speculative execution, which is critical in our calculation of the theoretical probability. Nevertheless, the experiments suggest the theoretical values are close to empirical ones, especially when $N = 8$, which is the value we used in the rest of our experiments.

A TSX abort does not itself raise a clock-AEX alarm; rather, the application thread raises a clock-AEX alarm upon noticing that at least one TSX abort occurred on the clock thread since the application thread last consulted the clock. Moreover, a TSX abort can occur for other reasons than an AEX occurring on the clock thread. As such, a clock-AEX alarm can reflect zero, one, or multiple AEXs on the clock thread. To measure the accuracy of clock-AEX alarms, then, we adapt definitions of *precision* and *recall* to our case as follows. First, we estimate a clock-AEX alarm to be *accurate* if at least one AEX of the clock thread preceded it by at most δ clock cycles, where δ is empirically determined as the longest execution time of any execution pathlet in the CFG of the program (and so is the maximum duration between clock accesses by the application thread for measuring the time to execute a pathlet). The *precision* of clock-AEX alarms is then the ratio of the number of accurate clock-AEX alarms to the total number of clock-AEX alarms. The *recall* of clock-AEX alarms is the ratio of the number of clock-thread AEXs that were followed within δ cycles by a clock-AEX alarm (*i.e.*, that were accurately detected) to the total number of clock-thread AEXs.

The precision and recall of clock-AEX alarms are shown in Fig. 7, when the shielded program (running on the application thread) was one of ten `nbench` applications [4]. Each bar represents an average calculated over ten runs. In each run, about 5000 AEXs were triggered on the clock thread. As can be seen there, the recall of clock-AEX alarms was often close to 1.0 and is above 0.95 in all cases. Precision was at least 0.83 in all cases but one (where it was 0.78). The somewhat lower precision of clock-AEX alarms reflects the fact that the clock thread's TSX transactions can abort for reasons other than AEXs, mainly due to effects of other applications running alongside it. It remains future work to mitigate these effects.

7.2.2 *Detecting AEXs on the Application Thread*

To show the shielded program cannot be preempted without being detected, we conducted the following experiments. The OS started the shielded application in the enclave and then periodically injected an interrupt to induce AEXs on the application thread. After each interrupt, the OS waited for long enough time to make sure the shielded program consulted its reference clock at least once (so that it had an opportunity to detect the AEX) before triggering the next interrupt.

Similarly to clock-AEX alarms, we measure the accuracy of app-AEX alarms using *precision* and *recall*. We estimate

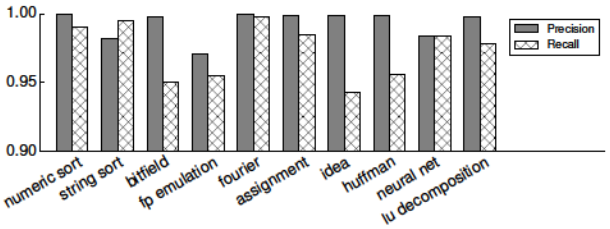

Figure 8: Precision and recall of app-AEX alarms during ten **nbench** applications.

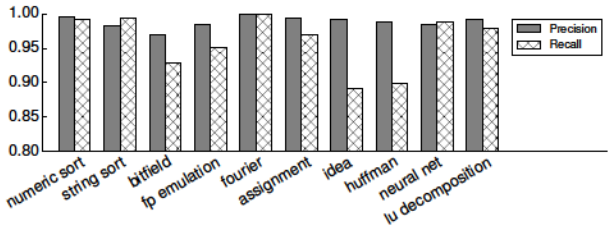

Figure 9: Precision and recall of app-AEX alarms during ten **nbench** applications when the application core and reference-clock core were set to run at their maximum and minimum frequencies, respectively.

an app-AEX alarm to be *accurate* if it was raised within δ cycles following an AEX of the application thread. Then, the app-AEX alarm precision was computed as the ratio of the accurate app-AEX alarms to the total number of app-AEX alarms. To define app-AEX alarm recall, we estimate an application-thread AEX to have been *undetectable* if it was followed within δ cycles by a clock-AEX alarm (and no intervening app-AEX alarm), as the clock-AEX alarm suggests that the clock thread was interrupted while the pathlet interrupted by the AEX was executing. The app-AEX alarm recall was computed as the ratio of the accurately detected application AEXs (i.e., followed by an app-AEX alarm within δ cycles) to the total number of detectable application AEXs.

In Fig. 8, we show the precision and recall of app-AEX alarms for ten programs in the **nbench** benchmark suite. The reported results are averages of ten runs. In each run, about 5000 to 6000 AEXs were triggered on the application thread. We can see from the figure that both precision and recall were at least 0.95 for all applications.

7.2.3 *Manipulated CPU Speeds*

A more advanced adversary may slow down the reference-clock CPU core to its minimum frequency $f_{\min}$ and speed up the application core to its maximum frequency $f_{\max}$, which on our platform are $f_{\min} = 0.8$GHz and $f_{\max} = 2.6$GHz, respectively. In this way, the attacker will minimize the likelihood that its application-thread AEXs result in app-AEX alarms. We made these adjustments by editing a file in **procfs**[5] and then re-ran the experiments in Sec. 7.2.2 again under this new setup. The results are shown in Fig. 9, which suggest modest changes in app-AEX alarm precision and recall. Even despite this manipulation, precision and recall remained above 0.96 and 0.91, respectively. As such, we conclude that the adversary manipulating core speeds is not a significant threat to DÉJÀ VU.

7.3 Performance Evaluation

We evaluated the performance overhead of DÉJÀ VU by measuring the normalized execution time of each of the ten **nbench** applications. Specifically, we compiled the **nbench** suite with LLVM and **Clang**. The baseline execution time was measured with the benchmark compiled without any instrumentation, and another version was compiled with DÉJÀ VU's instrumentation code for AEX detection. As both versions were compiled with LLVM, the increase in the runtime of the instrumented version over that of the baseline version, divided by the runtime of the baseline version, is the (relative) overhead of the instrumentation itself. The average

[5] /sys/devices/system/cpu/cpu0/cpufreq/scaling_setspeed

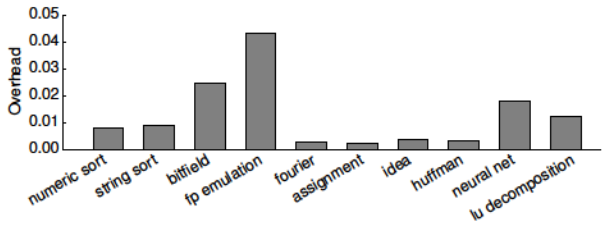

Figure 10: Performance overhead of DÉJÀ VU on **nbench** applications.

performance overheads are shown in Fig. 10. We can see from the figure that the runtime overhead was less than 5% for these applications. The instrumentation increases the size of the enclave binary by roughly 64%, mostly due to the DÉJÀ VU helper function library.

8. DISCUSSION

Core utilization. As discussed in Sec. 6, our design asks that the OS pin enclave threads (both the application threads and our reference-clock thread) to dedicated cores, so as to minimize the frequency of interrupts (and so AEXs) incurred by the enclave threads. This does not increase our trust of the OS; the OS' failure to abide by this contract will increase the interrupts (and so AEXs) detected by DÉJÀ VU, resulting in self-termination of the enclave (if that is what the security policy specifies). A malicious OS could accomplish the same ends by simply never scheduling the enclave threads in the first place.

Despite not requiring trust of the OS, this contract does come with costs, specifically interfering with the normal scheduling policy of the OS. Coupled with the fact that our design adds an additional thread to the enclave (the reference-clock thread), DÉJÀ VU does somewhat interfere with optimally making use of all available cores on the computer. That said, this contract also provides distinct benefits. In particular, this contract isolates the enclave workload from the non-enclave workload. So, for example, a busy web server running on (and incurring frequent interrupts on) other cores will not interfere with DÉJÀ VU.

Training coverage. DÉJÀ VU requires dynamic training of the execution time of each execution pathlet in the CFG of the programs. Although the execution times of execution pathlets are not very sensitive to inputs, coverage of training data is a limitation of our training-based detection method. However, as discussed in Sec. 6, a default timing threshold

is provided for each pathlet that is not covered in the training data. This threshold is basically the minimum time for an AEX, which is typically much larger than the execution time of the pathlets. Therefore, this threshold should lead to few superfluous app-AEX alarms but should still ensure detection of AEXs on the application thread, causing the detection system to work reasonable well even when the training data does not completely cover every execution pathlet. Nevertheless, future work will explore approaches to maximize the training coverage by exploiting techniques such as concolic execution [17].

Security policy upon AEX detection. Similar to any intrusion detector, DÉJÀ VU requires the users to specify a policy that dictates how AEXs detections should be addressed. We anticipate two general categories of applications that may need different treatment. The first category is applications that contain short secrets, such as cryptographic keys, that may be leaked through privileged side channels with relatively few side-channel observations. For these applications, it would be warranted to set AEX-detection thresholds more conservatively to increase app-AEX alarm precision, and to take even a low number of AEX detections as a serious indication of a threat (e.g., stopping the computation or changing keys). The second category includes applications like `libjpeg` and `freetype` that contain much larger secrets that might eventually leak over the course of a longer execution [53]. These applications may be tolerant of more AEXs, and so leaving the thresholds as, say, set in Sec. 6 and alerting a remote administrator in case of excessive AEXs might suffice.

Especially for the first category above, however, it is important to note that some AEXs will occur even when no attacks are underway, owing to the normal interrupts and page faults that occur on computers (even if the enclave threads are pinned to their own cores). So, a zero-tolerance policy is unlikely to be viable, even in conjunction with a very high precision. For this reason, particularly fragile applications might need to incorporate additional defenses (e.g., frequent, proactive key updates) to compensate for the fundamentally ambiguous situation (with respect to side-channel attacks) that the enclave is in.

Other side-channel threats. While DÉJÀ VU is an effective defense against side-channel attacks that induce AEXs (*e.g.*, controlled-channel attacks, and side channels in per-core caches), there remains the possibility of other side-channel attacks against an enclave. For example, last-level cache side-channel attacks [25, 33], which do not require the attacker to preempt the victim to conduct, should be equally potent against an enclave and, since existing system-level defenses (e.g., [27, 61]) presume a trusted OS, these attacks require additional research to address in this context (or hardware support, *e.g.*, [51, 26]).

9. CONCLUSION

In this paper we detailed the design and implementation of DÉJÀ VU, a system for detecting privileged side-channel attacks mounted by an untrusted OS on an SGX enclave. DÉJÀ VU detects AEXs that could give rise to such attacks, by timing each execution pathlet of the enclave application and detecting when its execution timing suggests that it was interrupted. The key challenge that DÉJÀ VU addresses is the lack of any reliable time source accessible to the enclave to measure pathlet execution times. To fill this need, DÉJÀ VU builds a novel reference clock leveraging hardware transactional support now available on Intel platforms. This transactional support allows us to construct a reference clock that will incur a transaction abort with high probability when the reference-clock thread is interrupted. Moreover, when it is not interrupted, the reference clock can be used to effectively delineate between when a pathlet suffered an AEX and when it did not. Our evaluations showed DÉJÀ VU reliably detects AEXs during pathlet executions. While the best policy for reacting to detections is application-specific, the detections supported by DÉJÀ VU are an important ingredient in defending SGX enclaves against privileged side-channel attacks.

Acknowledgements. This research was supported in part by NSF grants 1330599 and 1566444.

10. REFERENCES

[1] Intel 64 and IA-32 architectures software developer's manual, combined volumes:1,2A,2B,2C,3A,3B and 3C. http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html. version 052, retrieved on Dec 25, 2014.

[2] Intel 64 and IA-32 architectures software developer's manual volumes 3d: System programming guide, part 4. http://www.intel.eu/content/www/eu/en/architecture-and-technology/64-ia-32-architectures-software-developer-vol-3d-part-4-manual.html. Order Number: 332831-059US, June 2016.

[3] Intel Software Guard Extensions programming reference. https://software.intel.com/sites/default/files/managed/48/88/329298-002.pdf. October 2014.

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

[5] O. Aciiçmez. Yet another microarchitectural attack: exploiting I-Cache. In *2007 ACM Workshop on Computer Security Architecture*, 2007.

[6] 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.

[7] A. Baumann, M. Peinado, and G. Hunt. Shielding applications from an untrusted cloud with Haven. *ACM Transactions on Computer Systems*, 33(3), Aug. 2015.

[8] S. Checkoway and H. Shacham. Iago attacks: Why the system call API is a bad untrusted RPC interface. In *18th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2013.

[9] H. Chen, F. Zhang, C. Chen, Z. Yang, R. Chen, B. Zang, and W. Mao. Tamper-resistant execution in an untrusted operating system using a virtual machine monitor. Technical report, Fudan University, Aug. 2007.

[10] X. Chen, T. Garfinkel, E. C. Lewis, P. Subrahmanyam, C. A. Waldspurger, D. Bohen, J. Dwoskin, and D. R. K. Ports. Overshadow: A virtualization-based approach to retrofitting protection in commodity operating systmes. In *13th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 2–13.

[11] Y. Cheng, X. Ding, and R. H. Deng. AppShield: Protecting applications against untrusted operating system. Technical report, Singapore Management University, October 2013.

[12] J. V. Cleemput, B. Coppens, and B. De Sutter. Compiler mitigations for time attacks on modern x86 processors. *ACM Trans. Archit. Code Optim.*, 8(4), Jan. 2012.

[13] S. Crane, A. Homescu, S. Brunthaler, P. Larsen, and M. Franz. Thwarting cache side-channel attacks through dynamic software diversity. In *2015 Network and Distributed System Security (NDSS) Symposium*, 2015.

[14] J. Criswell, N. Dautenhahn, and V. Adve. Virtual ghost: Protecting applications from hostile operating systems. In *19th International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 2014.

[15] L. Domnitser, A. Jaleel, J. Loew, N. Abu-Ghazaleh, and D. Ponomarev. Non-monopolizable caches: Low-complexity mitigation of cache side channel attacks. *ACM Trans. Archit. Code Optim.*, 8(4), Jan. 2012.

[16] G. Doychev, D. Feld, B. Kopf, L. Mauborgne, and J. Reineke. Cacheaudit: A tool for the static analysis of cache side channels. In *22nd USENIX Security Symposium*, 2013.

[17] P. Godefroid, N. Klarlund, and K. Sen. Dart: Directed automated random testing. *SIGPLAN Not.*, 40(6):213–223, June 2005.

[18] D. Gruss, R. Spreitzer, and S. Mangard. Cache template attacks: Automating attacks on inclusive last-level caches. In *24th USENIX Security Symposium*, 2015.

[19] L. Guan, J. Lin, B. Luo, J. Jing, and J. Wang. Protecting private keys against memory disclosure attacks using hardware transactional memory. In *36th IEEE Symposium on Security and Privacy*, 2015.

[20] D. Gullasch, E. Bangerter, and S. Krenn. Cache games – bringing access-based cache attacks on AES to practice. In *32nd IEEE Symposium on Security and Privacy*, 2011.

[21] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. *SIGARCH Comput. Archit. News*, 21(2):289–300, May 1993.

[22] O. S. Hofmann, S. Kim, A. M. Dunn, M. Z. Lee, and E. Witchel. InkTag: Secure applications on an untrusted operating system. In *18th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2013.

[23] R. Hund, C. Willems, and T. Holz. Practical timing side channel attacks against kernel space ASLR. In *34th IEEE Symposium on Security and Privacy*, 2013.

[24] M. S. Inci, B. Gulmezoglu, G. Irazoqui, T. Eisenbarth, and B. Sunar. Seriously, get off my cloud! Cross-VM RSA key recovery in a public cloud. Cryptology ePrint Archive, Report 2015/898, 2015. http://eprint.iacr.org/.

[25] 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 *36th IEEE Symposium on Security and Privacy*, 2015.

[26] G. Keramidas, A. Antonopoulos, D. N. Serpanos, and S. Kaxiras. Non deterministic caches: A simple and effective defense against side channel attacks. *Design Automation for Embedded Systems*, 12(3), 2008.

[27] T. Kim, M. Peinado, and G. Mainar-Ruiz. STEALTHMEM: system-level protection against cache-based side channel attacks in the cloud. In *21st USENIX Security Symposium*, 2012.

[28] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis and transformation. In *International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*. IEEE Computer Society, 2004.

[29] P. Li, D. Gao, and M. K. Reiter. Mitigating access-driven timing channels in clouds using StopWatch. In *43rd IEEE/IFIP International Conference on Dependable Systems and Networks*, 2013.

[30] Y. Li, J. McCune, J. Newsome, A. Perrig, B. Baker, and W. Drewry. Minibox: A two-way sandbox for x86 native code. In *2014 USENIX Annual Technical Conference*, 2014.

[31] D. Lie, C. A. Thekkath, and M. Horowitz. Implementing an untrusted operating system on trusted hardware. In *19th ACM Symposium on Operating Systems Principles*. ACM, 2003.

[32] F. Liu and R. B. Lee. Random fill cache architecture. In *47th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE, 2014.

[33] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee. Last-level cache side-channel attacks are practical. In *36th IEEE Symposium on Security and Privacy*, 2015.

[34] Y. Liu, Y. Xia, H. Guan, B. Zang, and H. Chen. Concurrent and consistent virtual machine introspection with hardware transactional memory. In *20th International Symposium on High Performance Computer Architecture*, 2014.

[35] H. Mantel and A. Starostin. *Transforming Out Timing Leaks, More or Less*. Springer International Publishing, 2015.

[36] R. Martin, J. Demme, and S. Sethumadhavan. Timewarp: rethinking timekeeping and performance monitoring mechanisms to mitigate side-channel attacks. In *39th Annual International Symposium on Computer Architecture*, 2012.

[37] J. M. McCune, B. J. Parno, A. Perrig, M. K. Reiter, and H. Isozaki. Flicker: An execution infrastructure for TCB minimization. In *3rd ACM European Conference on Computer Systems*, 2008.

[38] D. Molnar, M. Piotrowski, D. Schultz, and D. Wagner. The program counter security model: Automatic detection and removal of control-flow side channel attacks. In *8th International Conference on Information Security and Cryptology*. Springer-Verlag, 2006.

[39] M. Neve and J.-P. Seifert. Advances on access-driven cache attacks on AES. In *13th International Conference on Selected Areas in Cryptography*, 2007.

[40] Y. Oren, V. P. Kemerlis, S. Sethumadhavan, and A. D. Keromytis. The spy in the sandbox: Practical cache attacks in javascript and their implications. In *22nd ACM Conference on Computer and Communications Security*. ACM, 2015.

[41] C. Percival. Cache missing for fun and profit. In *2005 BSDCan*, 2005.

[42] D. R. K. Ports and T. Garfinkel. Towards application security on untrusted operating systems. In *3rd Workshop on Hot Topics in Security*, 2008.

[43] A. Rane, C. Lin, and M. Tiwari. Raccoon: Closing digital side-channels through obfuscated execution. In *24th USENIX Security Symposium*, 2015.

[44] M.-W. Shih, S. Lee, T. Kim, and M. Peinado. T-SGX: Eradicating controlled-channel attacks against enclave programs. In *ISOC Network and Distributed System Security Symposium*, 2017.

[45] S. Shinde, Z. L. Chua, V. Narayanan, and P. Saxena. Preventing page faults from telling your secrets. In *11th ACM Asia Conference on Computer and Communications Security*, 2016.

[46] R. Ta-Min, L. Litty, and D. Lie. Splitting interfaces: Making trust between applications and operating systems configurable. In *7th USENIX Symposium on Operating Systems Design and Implementation*, 2006.

[47] E. Tromer, D. A. Osvik, and A. Shamir. Efficient cache attacks on AES, and countermeasures. *J. Cryptol.*, 23(2):37–71, Jan. 2010.

[48] V. Varadarajan, T. Ristenpart, and M. Swift. Scheduler-based defenses against cross-VM side-channels. In *23th USENIX Security Symposium*, 2014.

[49] B. C. Vattikonda, S. Das, and H. Shacham. Eliminating fine grained timers in Xen. In *3rd ACM Workshop on Cloud Computing Security*, 2011.

[50] Z. Wang and R. B. Lee. New cache designs for thwarting software cache-based side channel attacks. In *34th annual international symposium on Computer architecture*, 2007.

[51] Z. Wang and R. B. Lee. A novel cache architecture with enhanced performance and security. In *41st IEEE/ACM International Symposium on Microarchitecture*, 2008.

[52] J. C. Wray. An analysis of covert timing channels. In *IEEE Symposium on Security and Privacy*, 1991.

[53] Y. Xu, W. Cui, and M. Peinado. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *36th IEEE Symposium on Security and Privacy*. IEEE, 2015.

[54] J. Yang and K. G. Shin. Using hypervisor to provide data secrecy for user applications on a per-page basis. In *4th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE)*. ACM, 2008.

[55] Y. Yarom and K. E. Falkner. FLUSH+RELOAD: A high resolution, low noise, L3 cache side-channel attack. In *23rd USENIX Security Symposium*, 2014.

[56] F. Zhang, J. Chen, H. Chen, and B. Zang. Cloudvisor: Retrofitting protection of virtual machines in multi-tenant cloud with nested virtualization. In *23rd ACM Symposium on Operating Systems Principles*. ACM, 2011.

[57] T. Zhang, Y. Zhang, and R. Lee. Cloudradar: A real-time side-channel attack detection system in clouds. In *19th International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, 2016.

[58] 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*, 2012.

[59] Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart. Cross-tenant side-channel attacks in PaaS clouds. In *ACM Conference on Computer and Communications Security*, 2014.

[60] Y. Zhang and M. K. Reiter. Düppel: Retrofitting commodity operating systems to mitigate cache side channels in the cloud. In *20th ACM Conference on Computer and Communications Security*, 2013.

[61] Z. Zhou, M. K. Reiter, and Y. Zhang. A software approach to defeating side channels in last-level caches. In *23rd ACM Conference on Computer and Communications Security*, 2016.