T-SGX: Eradicating Controlled-Channel Attacks Against Enclave Programs

Ming-Wei Shih[†,⋆], Sangho Lee[†], and Taesoo Kim
Georgia Institute of Technology
{mingwei.shih, sangho, taesoo}@gatech.edu

Marcus Peinado
Microsoft Research
marcuspe@microsoft.com

***Abstract*—Intel Software Guard Extensions (SGX) is a hardware-based trusted execution environment (TEE) that enables secure execution of a program in an isolated environment, an *enclave*. SGX hardware protects the running enclave against malicious software, including an operating system (OS), a hypervisor, and even low-level firmwares. This strong security property allows the trustworthy execution of programs in a hostile environment, such as a public cloud, without trusting anyone (*e.g.*, a cloud provider) between the enclave and the SGX hardware. However, recent studies have demonstrated that enclave programs are vulnerable to an accurate controlled-channel attack: Since enclaves rely on the underlying OS, a curious or potentially malicious OS can observe a sequence of accessed addresses by intentionally triggering page faults.**

In this paper, we propose T-SGX, a complete mitigation solution to the controlled-channel attack in terms of compatibility, performance, and ease of use. T-SGX relies on a commodity component of the Intel processor (since Haswell), Transactional Synchronization Extensions (TSX), which implements a restricted form of hardware transactional memory. As TSX is implemented as an extension (*i.e.*, snooping the cache protocol), any unusual event, such as an exception or interrupt, that should be handled in its core component, results in an abort of the ongoing transaction. One interesting property is that the TSX abort suppresses the notification of errors to the underlying OS, which means that the OS cannot know whether a page fault has occurred during the transaction. T-SGX, by utilizing such property, can carefully isolate effects of attempts to tap running enclaves, thereby completely eradicating the known controlled-channel attack.

We have implemented T-SGX as a compiler-level scheme that automatically transforms a normal enclave program into a secured one. We not only evaluate the security properties of T-SGX, but also demonstrate that it applies to all the previously demonstrated attack targets including libjpeg, Hunspell, and FreeType. In addition, we evaluate the performance of T-SGX by porting ten benchmark programs of nbench to the SGX environment. The results are promising; that is, T-SGX incurs on average 50% runtime overhead, which is an order of magnitude faster than state-of-the-art mitigation schemes.

[†] The two lead authors contributed equally to this work.
[⋆] The author did part of this work during an intership at Microsoft Research.

NDSS '17, 26 February - 1 March 2017, San Diego, CA, USA

http://dx.doi.org/10.14722/ndss.2017.23193

I. Introduction

Hardware-based trusted execution environments (TEEs) have become one of the most promising solutions against various security threats, including malware, remote exploits, kernel exploits, hardware Trojans, and even malicious cloud operators [27]. ARM's TrustZone [1] and Samsung's KNOX [50] are now widely deployed on mobile phones and tablets. To secure traditional computing devices, such as laptops, desktops, and servers, the trusted platform module (TPM) [60], Intel's Trusted Execution Technology (TXT) [16], and Software Guard Extensions (SGX) [24] have been developed and are being adopted into mainstream products. Among these hardware-based TEEs, Intel SGX is getting considerable attention because it can be the basis for practical solutions in an important security domain: the trustworthy public cloud, which provides strong guarantees of both confidentiality and integrity, which are known to be the biggest obstacle to wider cloud adoption [27, 59]. Homomorphic encryption [14] has been proposed as a software-only solution to this problem, but, so far, it is too slow for practical uses. More critically, sensitive operations are often executed on potentially malicious clients [13, 36, 41], which significantly weakens the overall, end-to-end security of the system. In contrast, hardware-based Intel SGX provides strong security guarantees for running enclaves in combination with Intel's efforts on formal verification of the hardware specification and implementation of cryptographic operations [26]. The resulting security guarantees enable a variety of new applications, including data analytics [51], MapReduce [12], machine learning [46], Tor [31], network function virtualization (NFV) [53], and library OSs [4, 61, 62].

While Intel SGX draws significant attention to communities because of its strong security guarantees, researchers have recently demonstrated two critical side-channel attacks against SGX programs, namely, the page-fault- and cache-based side-channel attack [10, 54, 65]. The page-fault-based side-channel attack, also known as the controlled-channel attack, is particularly dangerous because it gives the malicious OS complete control over the execution of SGX programs. In contrast, the cache-based side-channel attack have to passively, thus non-interactively, monitor the execution from the outside. Specifically, to launch a controlled-channel attack, the malicious OS can stop an enclave program, unmap the target memory pages, and simply resume its execution. By using the leaked addresses, researchers [65] could reconstruct input text and image files from running enclave programs [4]. Similarly, the pigeonhole [54] attack could extract bits of encryption keys from cryptographic routines in OpenSSL and libgcrypt.

In response to the controlled-channel attack, two types of countermeasures have been proposed, namely, obfuscating memory accesses [9, 49, 54] and isolating page faults [19, 54], but both are limited in terms of performance or compatibility. First, memory access obfuscation suffers from huge performance degradation: up to 4000× overhead without significant developer effort [54]. Second, more efficient schemes, such as self-paging [19] and contractual execution [54], require new page-fault delivery mechanisms that do not exist in mainstream processors and are unlikely to be included in them in the foreseeable future. For example, Intel considers side-channel attacks as out of scope for SGX [26] and is unlikely to disrupt core processor components to accommodate such proposals.

In this paper, we propose a new, practical enclave design, T-SGX, that can protect any enclave program against controlled-channel attacks. At a high level, T-SGX transforms an enclave program such that any exception or interrupt that occurs during the execution is redirected to one specific page (see §V-B). We provide strong security guarantees against controlled-channel attacks under a conservative threat model (see §VII). T-SGX realizes this mechanism with a commodity hardware feature, Intel Transactional Synchronization Extensions (TSX), that was introduced with the Haswell processor. The key enabling property of TSX is the way it aborts an ongoing transaction when encountering an erroneous situation, such as a page fault or interrupt. In particular, when a page fault occurs, TSX immediately invokes a user-space fallback handler *without notifying the underlying OS*. The fallback handler recognizes whether the very recent attempt to execute a code page or access a data page has triggered a page fault. If it did, T-SGX carefully terminates the program. Further, TSX ensures that such traps and exceptions are *never exposed to system software including the OS and a hypervisor*, implying that the controlled-channel attack relying on page-fault monitoring is no longer possible with T-SGX because *even the OS cannot know whether a page fault has occurred*. However, obtaining a working, efficient TSX-secured enclave binary requires careful program analysis. First, TSX is very sensitive to cache usage; it treats cache conflicts and evictions as errors [25, §15.3.8.2]. Thus, we have to carefully compose transactional code regions based on their memory access patterns. Second, TSX treats any interrupts and exceptions as errors (*e.g.*, timer and I/O interrupts), so we cannot run a code region for a long time even if it makes no memory accesses. Third, setting up a TSX transaction is very expensive (around 200 cycles on our test machine with an Intel Core i7-6700K 4 GHz CPU), which implies a naïve solution, wrapping individual instructions with TSX, is impractical. Finally, we need to carefully arrange transactional code regions in memory to hide transitions between them from attackers (see §V-B).

T-SGX is based on a modified LLVM compiler satisfying the following three important design requirements. One is that T-SGX automatically transforms a normal enclave program into a secured version, all of whose code and data pages are wrapped with TSX. Another requirment is that T-SGX isolates the specific page for the fallback handler and other transaction control code, called *springboard*, from the original program's code and data pages to ensure that exceptions including page faults and timer interrupts can only be triggered on the springboard. The OS can still identify whether an exception has occurred at the springboard, but this does not reveal any meaningful information. Lastly, T-SGX ensures that no unexpected transaction aborts caused by benign errors (*e.g.*, transaction buffer overflow and timer interrupts), by carefully splitting a target enclave program into a number of *small execution blocks* satisfying the TSX cache constraints. A conservative splitting strategy (*e.g.*, secure individual basic blocks) significantly slows down T-SGX (§VIII). We develop compiler-level optimization techniques such as loop optimizations and cache usage analysis that maximize the size of execution blocks (§VI). Our evaluation results show the effectiveness of T-SGX in terms of security, compatibility, and performance. We applied T-SGX to three previous controlled-channel attack targets including libjpeg, Hunspell, and FreeType and demonstrated that the attack can no longer work. In addition, applying T-SGX to these programs require no source code modifications. We also checked the overall overhead of the programs. On average, the execution time increased by 40% and the memory consumption increased by 30%. Moreoever, we applied T-SGX to a popular benchmark suite, nbench, and confirmed that the performance overhead of T-SGX was 50% on average.

In summary, this paper makes the following contributions:

- **New security mechanism.** We develop a new security mechanism, T-SGX, that protects enclave programs from a serious threat: the controlled-channel attack. At compilation time, T-SGX transforms an enclave program into a secure version without requiring annotations or other manual developer efforts, and, most important, it does not require hardware modifications.
- **Novel usage of TSX.** To the best of our knowledge, T-SGX is the first attempt to use TSX to detect suspicious exceptions. Mimosa [18] was the first application of TSX to establish a confidential memory region, but it focuses on detecting read-write or write-write conflicts, which is the original use case of TSX. In contrast, we use TSX to isolate exceptions such as page faults and redirect them to a user space handler under our control.
- **Springboard and program analysis.** The properties of TSX (*e.g.*, cache- and interrupt-sensitivity) limit developers apply TSX to only a small protion of a program. Our springboard design and program analysis make a breakthrough: we can run any program in transactions without compatibility problems.
- **Evaluation and analysis.** We evaluated the security and performance of T-SGX by applying it to two representive groups of programs: three previous controlled-channel attack targets including libjpeg, Hunspell, and FreeType and a benchmark suite, nbench. The results show a 40% and a 50% performance degradation on the first and the second group, respectively. T-SGX is also easy to use; that is, it transforms all the programs with no source code modification.

The remainder of this paper is organized as follows. §II explains details about Intel SGX and TSX. §III describes controlled-channel attacks in depth. §IV provides the ideal system model. §V explains the design of T-SGX. §VI depicts how we implemented T-SGX. In §VII we conduct a security analysis of T-SGX. §VIII shows our evaluation results. §IX considers limitations of T-SGX. §X discusses related work. §XI concludes this paper.

II. Hardware Primitives and Motivation

In this section, we explain two hardware primitives supported by Intel CPUs: SGX for trusted computing and TSX for transactional memory. We also study how they handle CPU exceptions, since exceptions including page faults are the controlled channels a malicious OS can use to attack enclave programs [65].

A. Intel SGX

Intel Software Guard Extensions (SGX) is a set of extensions to the x86 instruction set architecture that aims to enable a hardware-based TEE, such that the Trusted Computing Base (TCB) consists of only the code and data that reside in a secure container (*enclave*) and the underlying hardware components. An SGX-enabled processor enforces new memory access semantics over an enclave such that the code and data within an enclave are inaccessible to all external software, including the operating system and the hypervisor. A dedicated physical memory region is allocated at boot time for enclave instantiation. To prevent known memory attacks such as memory snooping, SGX relies on a Memory Encryption Engine (MEE) to encrypt the enclave memory content. The enclave memory can only be decrypted when entering the CPU package during enclave execution. SGX provides a flexible programming model that allows an application to instantiate an enclave as part of its address space via the SGX instruction set. The enclave code and data are measured during the enclave initialization process. This measurement forms an enclave's identity, which a remote party can verify by means of remote attestation [23, 30]. In addition to the hardware-based protection mechanisms, SGX relies on the operating system to help with enclave initialization, exception handling, and resource management.

B. Intel TSX

In this section, we explain Intel Transactional Synchronization Extensions (TSX), which is Intel's implementation of hardware transactional memory (HTM) [20]. HTM was originally proposed to reduce the overhead of acquiring locks for mutual exclusion and to simplify concurrent programming. With HTM, a thread can transactionally execute in a critical section without any explicit software-based lock such as a spinlock or mutex. If a transaction completes without conflict, all of its read and write attempts are committed to memory. Otherwise, all of intermediate read and write attempts are rolled back (never exposed to the real memory) and a fallback (or abort) handler that was registered at the beginning of the transaction is invoked. The fallback handler decides whether to retry the transaction. Intel TSX supports two different interfaces, namely, hardware lock elision (HLE) and restricted transactional memory (RTM). For the discussion of this paper, we focus only on RTM.

Intel TSX provides four instructions: `XBEGIN`, `XEND`, `XABORT`, and `XTEST`. A thread can initiate a transactional execution using `XBEGIN` and terminate it using `XEND`. It can use `XABORT` to terminate a transaction and `XTEST` to test whether it is currently executing in a transaction.

Figure 1 shows a code snippet that uses TSX. It first executes `_xbegin()` (*i.e.*, `XBEGIN`) to begin a transaction. If it succeeds, `_xbegin()` returns `_XBEGIN_STARTED` and continues to execute

```
1  unsigned status;
2
3  // begin a transaction
4  if ((status = _xbegin()) == _XBEGIN_STARTED) {
5    // execute a transaction
6    [code]
7    // atomic commit
8    _xend();
9  } else {
10   // abort
11 }
```

Fig. 1: A basic example of Intel TSX. `_xbegin()` initiates a transaction region to execute `[code]` and `_xend()` closes the region. An exception at `[code]` makes the control flow go to the `else` block.

the code inside the `if` block (line 6). If there is no conflict, the program will eventually execute `_xend()` (*i.e.*, `XEND`) to atomically commit all the intermediate results. However, if there is a conflict or an exception (§II-C), the transaction is rolled back and the program executes the `else` block (line 10) to handle the error.

Technical details. Understanding the technical details of the TSX implementation [34] is important because such details can explain why TSX exhibits the described behavior (§II-C2). During a transaction, HTM needs a buffer to store intermediate data read or written, so it can commit them to memory at the end of the successful transaction. Instead of introducing a separate buffer, TSX uses the L1 cache as a buffer. This choice was made not only to avoid extra storage requirements, but also to simplify the implementation of TSX; it piggybacks on the existing cache coherence protocol to detect memory read or write conflicts without introducing complex new logic. The cache coherence protocol maintains data consistency between the caches of different cores such that TSX can detect data conflicts at the granularity of cache lines and roll back a transaction when a conflict occurs.

C. Exceptions inside SGX and TSX

During execution, a CPU can encounter various exceptions, such as page faults, general protection faults, and interrupts. When an exception occurs, the CPU calls the corresponding exception handler managed by the OS to resolve the problem or gracefully terminate execution. The CPU handles exceptions that occur during enclave and transactional execution differently from those that occur during normal execution.

1) SGX: Asynchronous Enclave Exit (AEX): Although SGX assumes the underlying OS could be malicious, it relies on the OS for exception handling. SGX takes special provisions to minimize information leakage during exception handling. Any exception or interrupt that arrives during enclave execution causes an *Asynchronous Enclave Exit (AEX)*. Figure 2 depicts how the AEX is conducted. The processor first stores the enclave's register context and the exit reason (exception code) in a region of enclave memory called the state save area (SSA) and loads synthetic values into the registers. In the case of page faults, the processor provides the OS only with the base address of the faulting page and not with the exact address. It then transfers control to the regular OS kernel exception handler. Eventually, the exception handler will return control to a user mode trampoline function outside the enclave, which can call the `ERESUME` instruction. `ERESUME` will restore the enclave's saved register context and resume enclave execution.

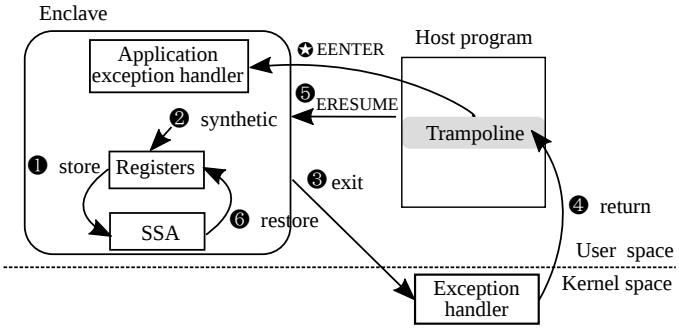

Fig. 2: Steps of an SGX asynchronous enclave exit (AEX). ❶ The processor stores register values and the exit reason into the state save area (SSA) inside the enclave. ❷ The processor loads synthetic data into registers. ❸ The enclave exits directly to the kernel space exception handler. ❹ The exception handler handles the interrupt and returns to the trampoline. ❺ The trampoline resumes the enclave. ❻ The processor restores the stored register values and resumes enclave execution. In addition, the trampoline can call an application exception handler inside the enclave to handle exceptions the OS cannot process.

Limitation. Although the AEX hides the register context and the exact address of an exception, information about the exception still leaks to the OS. For example, after each page fault, the OS learns which page the enclave attempted to access. This information is the basis for the controlled-channel attack [65].

2) TSX: Transaction Abort: When two transactions conflict with each other (*e.g.*, their read and write sets overlap), one will be canceled, and thus aborted (see §II-B). TSX also aborts a transaction when encountering an exception because a ring transition is not possible while executing a transaction. In the case of *synchronous exceptions* that occur during the execution of a specific instruction (*e.g.*, page fault, general protection fault, divide-by-zero), the exception is *not delivered* to the OS because TSX intentionally suppresses it ([25, §15.3.8.2]). On the other hand, an asynchronous exception (*e.g.*, timer interrupt and I/O interrupt) will be delivered right after a transaction is aborted and rolled back, because suppressing such interrupts would make user-space processes non-preemptable, which would interfere with OS scheduling.

D. Taking Control of Exception Handling

Using TSX inside SGX enclaves makes it possible to route exceptions such as page faults to TSX abort code inside the enclave and not to the ring-0 exception handler of the untrusted OS. This deprives attackers of all information about page faults and allows the enclave to identify potential attacks. We describe a design based on this observation in §V.

III. Controlled-channel Attack Revisited

In this section, we briefly explain the controlled-channel attack [65] (called pigeonhole attack in [54]) that allows a malicious OS to infer sensitive computation and data inside a TEE such as Haven [4] and InkTag [21]. We limit the discussion to attacks against SGX, which is the focus of our paper and which provides stronger security guarantees than a trusted hypervisor (*e.g.*, InkTag).

A. Threat Model

We explain the threat model of the controlled-channel attack. Note that our system, T-SGX, assumes the same threat model.

First, the attack assumes that an OS can manage (*e.g.*, map and unmap) enclave memory pages although it cannot see their contents. Whenever an enclave program attempts to access an unmapped page, the OS will receive a page fault to handle it; then the handler either remaps the page and resumes the program or generates an access violation error. However, this attack does not assume that the OS knows the exact offset of a page fault because TEEs can hide this information from the OS.

Second, the attack assumes that an attacker knows the detailed behavior of a target enclave program, especially its memory access patterns according to inputs. The attacker has already analyzed a target enclave program's source code and/or binary in detail to obtain the information. Also, this attack ignores programs with obfuscated memory access patterns (*e.g.*, Oblivious RAM (ORAM) [40, 49]) because they do not have visible behavior characteristics.

Third, the attack assumes that an attacker cannot arbitrarily run a target enclave program. Due to remote attestation, a user will know how many times his/her enclave program is executed in the public cloud such that it is difficult to run the target enclave program many times without the user's approval.

Fourth, the attack relies only on a noise-free side channel: page fault information. Other noisy side channels, including cache and memory bus, are out of the scope for this paper.

B. Controlled-channel Attack

The controlled-channel attack uses page faults as a controllable side channel. Since a malicious OS can manipulate the page table of an enclave program, it can know which memory pages the enclave program wants to access by setting a *reserved* bit in page table entries and monitoring page faults.

In contrast to a normal execution environment, the malicious OS cannot see the exact faulting address but only the page frame number because SGX masks the exact address, as explained in §II-C1. To overcome this limitation, the controlled-channel attack analyzes *sequences of page faults* rather than individual page faults.

The final step of the controlled-channel attack is correlating the page fault sequences with the results of offline, in-depth analysis of a target enclave program. This allows the attacker to infer the input to the enclave program if the memory access pattern of the program varies sufficiently with the input.

Effectiveness. The original controlled-channel attack was demonstrated against three popular libraries: FreeType, Hunspell, and libjpeg. The evaluation results show that the attack can accurately infer the input text and images to the libraries [65]. Shinde *et al.* [54] use a similar attack to extract bits of cryptographic keys from the OpenSSL and libgcrypt libraries.

C. Known Countermeasures

A few countermeasures against controlled-channel attacks have been discussed, but most of them are neither practical nor

secure. Intel has revised its SGX specification to support an option for recording page faults and general protection faults in the SSA [24]. However, this countermeasure is incomplete because a malicious OS can cause the SSA to be overwritten (details in §III-D). Second, Intel has suggested static and dynamic analysis to eliminate all feasible input-dependent code and data flows [26]. But, this requires significant developer effort and incurs non-negligible performance overhead. Third, Shinde *et al*. [54] have proposed deterministic multiplexing, a software-only solution against the controlled-channel attack. However, its performance overhead is tremendous without developer-assisted optimizations. Finally, Shinde *et al*. also have proposed a new execution model (contractual execution) that makes a contract between the enclave program and the OS to ensure that a specified number of memory pages reside in the enclave. Their proposal, however, requires modifications to core processors components. Such changes appear difficult and unrealistic.

D. Overwriting Exit Reason

As mentioned in §II-C1, the SSA stores the exit reason for each AEX. However, we found that a malicious OS can easily overwrite the exit reason by sending an arbitrary interrupt to an enclave program because the SSA stores only the last exit reason[1]. This makes an enclave program unaware of page faults, even if it uses an option `SECS.MISCSELECT.EXINFO=1` to record page faults and general protection faults.

We experimentally confirmed that a malicious OS can overwrite the exit reason of a page fault by using a fake general protection fault. When a page fault is generated, a corresponding address is stored in `SSA.MISC.EXINFO.MADDR` and `PFEC` is stored in `SSA.MISC.EXINFO.ERRCD` for later use [24]. However, a general protection fault could overwrite these fields: it stores 0 in `SSA.MISC.EXINFO.MADDR` and `GPEC` in `SSA.MISC.EXINFO.ERRCD`. We have found that a malformed Advanced Programmable Interrupt Controller (APIC) interrupt generates a general protection fault. The OS can program the APIC to generate such interrupts and thus general protection faults during enclave execution. Therefore, if a malicious OS generates a malformed APIC interrupt for an enclave program right after handling a page fault, the fields in the SSA are overwritten such that an enclave program cannot know whether or not a page fault has occurred. Further, the OS can generate another normal interrupt (*e.g*., a timer interrupt) later to even clear up the `GPEC` flag. Thus, we conclude that relying on the exit reason cannot protect an enclave program from the controlled-channel attack.

IV. System Model

In this section we explain our ideal system model. An ideal enclave (*uncontrollable enclave*) protects any enclave program from the security threats explained in §III. The basic requirement of the uncontrollable enclave is to enable an enclave program to know every interrupt and page table manipulation, and stop its normal execution when it detects that the OS has unmapped any of its sensitive memory pages.

[1]There is an SSA stack for handling nested exceptions. However, this overwriting attack is not about nested exceptions because it sends a new interrupt right after handling the previous interrupt.

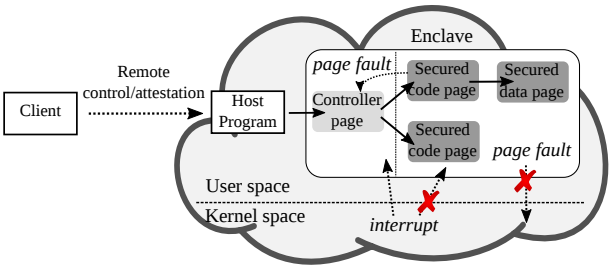

Fig. 3: The uncontrollable enclave model. It consists of secured and controller pages. The secured page is not interrupted by the OS and its page fault is delivered to the controller page instead of to the OS. The controller page manages control and data flows between secured pages and handles page faults generated by accessing secured pages.

To achieve these goals, the uncontrollable enclave allows an enclave program to have two kinds of memory pages: *secured pages* and *controller pages*, as shown in Figure 3. First, the secured pages are *unobservable* pages containing all code and data of an enclave program. The OS cannot interrupt the enclave program when it executes or accesses the secured pages, and it cannot monitor page faults generated due to the execution or access of these pages. Since secured pages are uninterruptible, the uncontrollable enclave needs to ensure that execution with secured pages is short (*e.g*., up to the interval of a timer interrupt) to prevent a malicious enclave program from fully occupying a CPU core. Also, when the uncontrollable enclave detects that any of the secured pages are unmapped, it treats the OS as malicious.

Second, the controller pages relay the control and data flow between the secured pages, check whether access to the secured pages is hindered by the OS (*i.e*., unmapped), and interact with the OS for scheduling and system calls. The OS can interrupt their execution and monitor page faults generated by accessing them. However, revealing their behavior does not leak much information because they are just trampoline pages and the actual execution of an enclave program is performed inside the secured pages.

The uncontrollable enclave ensures that no page fault sequence (*i.e*., inter-page accesses) is revealed to an OS. First, when the uncontrollable enclave identifies that a secured page is unmapped, it stops its execution. This could reveal up to a single page fault to the OS. Second, the enclave program can let its remote client know whether or not it has successfully terminated by sending an acknowledgment message. A lack of acknowledgment also means that there was a problem. Third, the uncontrollable enclave prevents the OS from running the enclave program arbitrarily. To achieve this, the enclave program checks whether its client allows the OS to run itself during a remote attestation process. The remote client would completely disallow any further execution if the program did not send acknowledgment messages before. Note that it is natural to assume that an enclave program runs in the cloud and its remote client controls its execution.

Based on these requirements, we implement a prototype scheme, T-SGX. T-SGX does not ensure perfect information leakage prevention, but we believe it is sufficient to make the known controlled-channel attacks impractical (see §VII for details).

```
1  // original code
2  void foo(char *msg, size_t len) {
3    const char *secret = "key";
4    ...
5  }
6
7  // protected code
8  void ecall_foo(char *msg, size_t len) {
9    if ((status = _xbegin()) == _XBEGIN_STARTED) {
10     foo(msg, len);
11     _xend();
12   } else {
13     // abort: e.g., page fault detected
14     abort_handler();
15   }
16 }
```

Fig. 4: A straw man example that wraps the entire enclave code in a TSX transaction to prevent controlled-channel attacks.

V. Design

In this section, we describe in detail the design of T-SGX, which is a practical realization of the uncontrollable enclave model (§IV). In particular, we explain how to realize the model's various components using Intel TSX.

A. Overview of the TSX-based Design

This section describes a working instantiation of our architecture that relies only on a widely deployed standard processor feature (TSX). This approach yields a practical and effective side-channel mitigation that can be used today.

Intuitively, the main value of TSX as a side-channel mitigation lies in its ability to suppress page faults and other synchronous exceptions. A page fault that occurs during a TSX transaction will not be delivered to the untrusted ring 0 page fault handler. Instead, the processor will abort the transaction and transfer control to the transaction's abort code. Thus, our strategy will be to run enclave code inside transactions and to place a trusted exception handler in the TSX abort code path.

Figure 4 shows a simple example of an enclave program and its TSX-based transformation. The code between `_xbegin` and `_xend` is executed as a transaction. The `else` branch contains the abort code path. TSX guarantees that any page fault that occurs while executing `foo(msg,len)` is suppressed and control is transferred directly to the `abort_handler` in the `else` branch.

A simple design idea could be to wrap the entire enclave program in a single TSX transaction. However, for typical programs, such transactions will never complete because (a) TSX will abort a transaction if its write or read set is too large to fit into the L1 or L3 cache, respectively, and (b) long-running transactions are highly likely to be aborted by interrupts. Thus, we have to partition the program into small execution blocks and wrap each execution block in a transaction.

This requires the ability to perform detailed static analysis as well as a number of program transformations. For this reason, we integrate T-SGX into the compiler. As the source code is compiled into an enclave binary, T-SGX computes an appropriate partitioning into execution blocks and makes sure each execution block is protected by TSX by conservatively placing `XBEGIN` and `XEND` instructions (see §V-C for details).

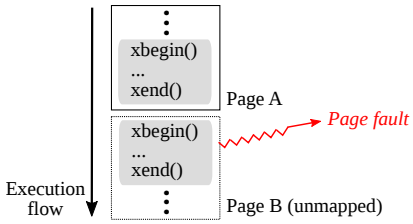

Fig. 5: Careless usage of TSX revealing a page fault. An attacker can monitor the page fault at Page B because a transition between Page A and Page B is not in a transaction.

B. The Springboard

Using many small transactions entails a new problem. As page faults are not suppressed across transactions, an attacker may still see all page faults he/she is interested in, unless transactions are carefully arranged in memory. Figure 5 shows an example in which the transition between two transactions is leaked because they are at a page boundary. An attempt to execute the first instruction on page B causes an observable page fault outside a transaction.

T-SGX solves this problem by placing all code that executes outside transactions on a single page. We call this page the *springboard*. Figure 6 displays the code that performs the transitions between consecutive transactions. An important property of this code is that *it does not access memory on any other page (e.g., stack, heap)*. Upon receiving an enclave function call from the host, the entry function begins by jumping to the `springboard.begin` block, which starts a transaction with an `XBEGIN` instruction followed by a jump to the start of first the block (`call` in this example). At the end of each block, the T-SGX compiler inserts two instructions that load the address of the next block into a register and jump to the `springboard.next` block. Code on the springboard then ends the current transaction (`XEND`), begins the next transaction (`XBEGIN`), and jumps to the start of the next block, as indicated by the register value provided by the previous block. Right before the end of execution, `springboard.end` ends the last transaction (`XEND`).

T-SGX also places the transaction abort code on the springboard page. Like the code that transitions between transactions, the abort code also executes outside a transaction. It is thus subject to page faults, and we ensure that it does not access memory outside the springboard. With this code layout, the only enclave page for which access could possibly result in a page fault is the springboard. This could happen (a) at the transaction transition (`springboard.next`, `springboard.begin`), and at `springboard.end` in Figure 6 and (b) in the transaction abort code (`springboard.abort`).

Example. Figure 7 shows how a host program and an OS interact with an enclave program secured by T-SGX.

1) The host program uses the SGX `EENTER` instruction to call a function inside the enclave.
2) `EENTER` transfers control to the enclave's *springboard*. The springboard starts the first transaction and jumps to the first execution block. As execution blocks complete and jump back to the springboard, the springboard completes and initiates transactions and jumps

```
springboard:
springboard.next:
  xend
springboard.begin:
  xbegin springboard.abort
  jmpq   *%r15

springboard.end:
  xend
  jmpq   *%r15

springboard.abort:
  # (abort handler code)
  ...
  # resume execution
  jmp    springboard.begin
```

```
# entry point to the function wrapper
entry_point:
  leaq  EB.start(%rip), %r15
  jmp   springboard.begin
EB.start:
  # (load parameters)
  call   _function
  # (save return value)
  leaq  EB.end(%rip), %r15
  jmp   springboard.end
EB.end:
  ...
  enclu[EEXIT]

# transformed function
_function:
  subq $40, %rsp
  ...
  leaq EB.1(%rip), %r15
  jmp  springboard.next
EB.1:
  ...
  leaq EB.2(%rip), %r15
  jmp  springboard.next
EB.2:
  ...
```

Fig. 6: Transaction transition code on the springboard and at the end of each execution block (denoted EB).

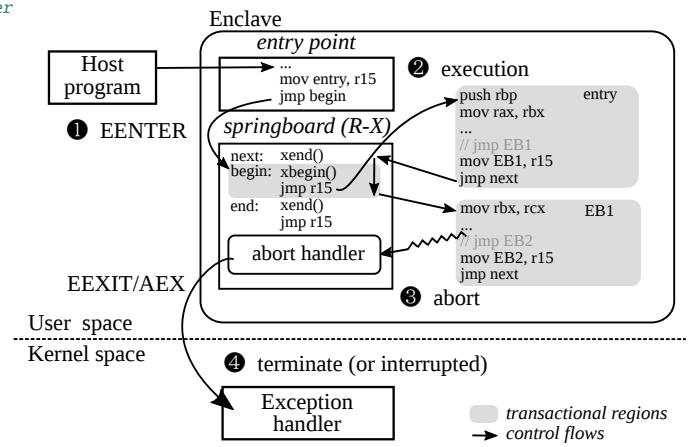

Fig. 7: Overall procedure of T-SGX: ❶ A host program calls an enclave program. ❷ Enclave execution is managed by the springboard that jumps into execution blocks scattered across multiple pages. The execution blocks jump back to the springboard when they are successfully executed. ❸ When an exception occurs in a execution block, control goes directly to the abort handler on the springboard. ❹ The enclave program either terminates or is interrupted. The OS can only identify the page containing the springboard.

to subsequent execution blocks. While these execution blocks may be distributed over many memory pages, only the springboard contains code that is not wrapped in a transaction.

3) If an exception occurs inside an execution block, the processor transfers control directly to the abort handler whose address is specified at `XBEGIN`.
4) The abort handler determines whether it has to restart the transaction or terminate the enclave program. The operating system will only see exceptions on the springboard page.

C. Execution Blocks

This section explains how the T-SGX compiler partitions a program into execution blocks that can be executed as transactions. We begin with a simple partitioning scheme that yields correct and functional programs. After that, we introduce various optimization techniques that drastically reduce the overhead of the simple scheme.

T-SGX computes the control flow graph of the program and tests for each basic block if it satisfies the transaction limits imposed by TSX and an execution time bound we establish. In particular, T-SGX makes a conservative estimate of the write and read sets of the basic blocks with respect to a cache model, as explained in §V-C1. We approximate execution time by counting the number of instructions in the basic block. Most basic blocks satisfy the two constraints. The remaining basic blocks are split by T-SGX into smaller blocks until all split blocks satisfy the transaction constraints. The resulting set of blocks is the partitioning into execution blocks under the basic scheme.

1) Transaction Constraints: TSX imposes strict bounds on the read and write sets of each transaction. The write set must fit into the L1 data cache. That is, the L1 data cache must be able to hold all memory writes of a transaction.

For example, on Skylake processors, the L1 data cache has a size of 32 kB. It is 8-way set associative with 64-byte

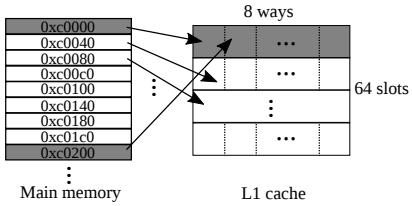

Fig. 8: Mapping of memory addresses to L1 cache slots.

cache lines. We can visualize this cache as eight copies (ways) of a 4 kB page partitioned into 64 slots of size 64 bytes (see Figure 8). The 64-byte cache line size is the granularity at which cache space is assigned. Multiple write operations within a single 64-byte aligned 64-byte address range occupy a single cache line. This 64-byte line is mapped to the slot in the L1 cache at the same page offset. A memory access within the 64-byte line causes it to be loaded into one of the eight ways at the corresponding slot in the L1 cache. This will cause the previous content of the of the way to be evicted from the L1 cache.

As the write set of a transaction must be kept in the L1 cache until the end of the transaction, a transaction will fail if its write set includes more than eight cache lines that map to a single slot. The T-SGX compiler uses this cache condition to determine if the write set of an execution block is too large. If it is, it will split the execution block into smaller units (as explained above).

Since the exact addresses of memory write operations may not be known at compile time, we have to use a conservative approximation. That is, given uncertainty about the addresses of memory accesses at run time, we assume the worst possibility. This may cause T-SGX to split the program into unnecessarily small execution blocks. However, it guarantees that the write set will fit into the L1 cache.

```
# TSX Basic
EB:
  ...
  leaq loop.header(%rip), %r15
  jmp  springboard.next
loop.body:
  ...
  incq 32(%rsp)
  leaq loop.header(%rip), %r15
  jmpq springboard.next
loop.header:
  ...
  cmpq $100, 32(%rsp)
  leaq loop.body(%rip), %r15
  jbe  springboard.next
  leaq loop.end(%rip), %r15
  jmp  springboard.next
loop.end:
  ...
```

```
# T-SGX
EB:
  ...
  leaq loop.header(%rip), %r15
  jmp  springboard.next
loop.body:
  ...
  incq 32(%rsp)
loop.header:
  ...
  cmpq $100, 32(%rsp)
  jbe  loop.body
  leaq loop.end(%rip), %r15
  jmp  springboard.next
loop.end:
  ...
```

Fig. 9: An example of the loop optimization.

More precisely, we distinguish between three types of memory accesses. First, addresses that are known completely at compile time can be mapped directly to a cache slot. Second, memory accesses given by an unknown base address and a known fixed offset (e.g., `rsp+8`, `rsp+16`) are grouped by the base pointer. We compute the maximum number of occupied ways separately for each group (base pointer) and add the maxima over all groups. Third, we model an access whose address is completely unknown as occupying one way of every slot. Finally, we add the largest way-count from each of the three cases to obtain an upper bound on the L1 requirements of the execution block.

We use a similar strategy to analyze the read set of execution blocks, and we count instructions as a proxy for execution time.

D. Optimization Techniques

An empty transaction with `XBEGIN` and `XEND` costs about 200 cycles. This can have a significant performance impact on the end-to-end application run time (see §VIII). The simple, basic-block-based partitioning uses basic block boundaries as the default place to begin and end transactions. However, it is often possible to place transactions around larger units of program execution, such as loops or functions. This subsection describes optimization techniques that follow this strategy and that can remove most of the overhead of the simple partitioning. Note that these optimizations do not interfere with OS scheduling, since interrupts cause transactions to be aborted (§II-C2).

1) Loops: Simple partitioning places transactions inside the loop body. That is, every iteration of even the simplest loops (*e.g.*, `memcpy()`) is at least one separate transaction. Our first optimization technique is to pull the transaction out of the loop where possible. Rather than creating a transaction for every loop iteration, we create a single transaction for the entire loop execution.

The main difficulty is to determine the write set of a loop. In general, this is not a tractable problem. However, in practice, many loops have simple relationships between the iteration number and the addresses of memory accesses in that iteration. For example, we frequently observe a simple linear relationship. That is, during the k-th iteration, the loop will access address $a + b * k$, where a and b are constants known at compile time. We use data-flow analysis to determine such relationships.

Given the write set of the loop, we perform the tests of §V-C1 to determine if the optimization can be applied. If the test fails because the number of loop iterations is too large or unknown, we can still apply the optimization by partially unrolling the loop. For example, if for up to 100 iterations the write set of the loop fits into the L1 cache, we place a transaction around every 100 iterations of the loop. This allows us to amortize the transaction cost over possibly many loop iterations (see Figure 9).

2) Functions and if-statements: This optimization attempts to merge all execution blocks within a function into a single execution block covering the entire function. We attempt to compute the write and read sets and instruction count for the entire function. If the function is complicated (*e.g.*, contains loops), this may not succeed, and we do not optimize the function. If we can obtain the read and write sets and the instruction count and if they pass the tests of §V-C1 then we merge the entire function into a single execution block.

Similarly, if we can determine the read and write sets of if-then-else statements and if they meet the conditions of §V-C1, we merge the if-then-else statement into a single execution block.

E. Abort Sequence

If a transaction fails, TSX will transfer control to the abort address specified in the `XBEGIN` instruction. T-SGX places this address on the springboard.

A simple version of the abort code restarts the transaction unconditionally until it succeeds. The appeal of this design lies in its simplicity. The abort code is stateless and only a few instructions long. However, if a transaction has to access a page that has been unmapped, this design will restart the transaction indefinitely, which is not an optimal defense strategy (§VII).

The alternative is for the abort code to monitor transaction aborts for signs of attacks and to stop program execution if an attack is detected. Lacking hardware support for distinguishing between page faults and regular interrupts as the cause of a transaction abort, we use the following criterion. *If a transaction aborts more than n times, the abort code will terminate program execution,* where n is a parameter that must be chosen such that the likelihood of seeing n consecutive transaction aborts due to benign causes under normal operation is very low. Based on the analysis of §VIII-B4, we set $n = 10$. The controlled-channel attack [65] requires millions of page faults to obtain sensitive information, so that 10 would be a reasonable threshold to defeat it.

Aborting execution when an unmapped page is detected may leak to the attacker that the enclave was trying to access this page. However, as explained in §VII, this strategy ensures that the attacker does not learn anything else through the page fault channel.

An implementation difficulty arises from the fact that the attack detection code is not stateless, as it has to count the number of times a transaction is aborted. In order to maintain the important springboard property that the only memory

accesses of springboard code outside transactions are execute accesses to the springboard page, we store the counter in a CPU register that we reserve in the compiler.

F. Preventing Reruns

The decision to run the enclave should be made by its owner and not by the attacker. This can be easily enforced by having the enclave code wait for a cryptographically secured authorization before it accesses sensitive data. We make this authorization an optional part of T-SGX.

The attack model of [65] assumes that the victim runs only once. Furthermore, the use model described in [65] (a remote user controlling a SGX-protected Haven instance or VM in the cloud via a remote desktop protocol) effectively includes an authorization to run (the remote user's commands) that is cryptographically secured (through the remote desktop protocol).

G. External calls

T-SGX places an `XEND` before calls from the enclave into the untrusted part of the address space (an `EEXIT` instruction). Similarly, T-SGX places a `XBEGIN` instruction close to the enclave entry points specified in the SGX Thread Control Structures (TCSs).

H. Illegal instructions

The T-SGX compiler ensures that no instructions that are illegal under SGX or TSX are generated for an enclave binary. This is unproblematic, as those instructions are not necessary to generate regular application code.

VI. Implementation

We have built a prototype of T-SGX based on the LLVM compiler. Our prototype produces T-SGX-enabled binaries that can be run in an enclave just like the original binary. Our prototype can handle arbitrary C and C++ code.

The main part of our prototype is integrated into the back-end of LLVM. It starts by performing the analysis described in §V based on the basic blocks produced by LLVM. After that, it modifies the instruction sequence as it is being emitted by LLVM. In particular, it places two instructions (to load the address of the next execution block into the `r15` register and to jump to the springboard) at the end of each execution block. In the case of 64-bit code, we reserve the `r15` register for this purpose (*i.e.*, to communicate the address of the next block to the springboard). Jump and call instructions (including indirect jumps and calls) are also made to jump to the springboard with the destination address loaded into `r15`.

As TSX uses the `rax` register, we reserve a second register to save the value of `rax` at the end of each execution block that writes to `rax` and to restore it at the beginning of each execution block that reads `rax` before writing to it.

Our prototype also includes a plugin to the LLVM front-end that injects a function wrapper for each exported function into the LLVM intermediate representation. The entire prototype consists of 4,110 lines of C++ code.

VII. Security Analysis

For T-SGX-based enclave programs, the attacker can only observe page fault locations (faulting addresses) on (a) the springboard page and (b) the unsecured pages containing function wrappers for the external enclave entry points. The latter can be ignored, as they do not access sensitive data.

Furthermore, as shown below, the attacker cannot use page faults to obtain deterministic notification when enclave execution accesses the springboard. These two properties of T-SGX disable the two main uses of the page-fault channel in the attacks of [65] and [54]: (a) leaking page numbers of memory accesses and (b) giving the attacker synchronization points that allow him/her to track the the victim's execution. An example of the latter is the strategic unmapping of code pages in [65] such that, every time the victim calls a function that accesses sensitive data (looking up a word in the Hunspell hash table, rendering a character, decoding an 8×8 pixel block of an image), a page fault interrupts (stops) the victim and invokes the attacker, who can then advance him/her state machine and update the set of unmapped pages. By blocking these two mechanisms, T-SGX effectively protects enclave programs against the known page-fault-channel-based attacks.

For the full-strength T-SGX variant that requires the enclave-owner's consent to run the enclave program (§V-F) and that aborts enclave execution as soon as a page fault is detected (§V-E), a stronger statement can be made: *The attacker will learn at most one page access by the victim.* Recall that the attacks of [65] required millions of page faults.

Given a reliable attack detection mechanism, the argument is straightforward. The first time one of the victim's transactions aborts, T-SGX will detect an attack and stop the execution. As the attacker will not be able to run the victim again, he/she cannot observe more than the one page access that caused the springboard to abort the execution. Whether the attack detection described in §V-E is sufficiently robust is arguable.

Attack detection. For enclave execution, it is reasonable to require that the enclave encounters no unexpected page faults. Thus, attack detection reduces to detecting page faults for T-SGX-secured pages. The problem would be trivial if the TSX hardware would distinguish between page faults and interrupts in the `eax` value provided to the abort handler. Lacking such hardware support, our abort handler declares an attack after a small number of consecutive transaction failures. This approximation is motivated by our evaluation (§VIII): Transactions tend to be short (1,000 to 2,000 cycles), and we have never observed more than three consecutive transaction aborts or false positives.

Restarting the transaction several times before aborting enclave execution could, in principle, give the attacker an opportunity to observe that the page has been accessed and to make the page accessible before the springboard terminates enclave execution. However, it appears that the attacker would have to rely mainly on other mechanisms (beyond the page-fault channel) to detect that the page had been accessed. In other words, while we cannot exclude the possibility that the attacker could gain more information, it appears that he/she would have to rely primarily on powerful mechanisms beyond the page-fault channel (e.g., cache side channels), which is not the focus of T-SGX.

Attacks on the Springboard. We noted above that the attacker cannot obtain deterministic notification of springboard accesses through the page-fault channel. More precisely, the attacker cannot force page faults on springboard accesses.

Before execution of sensitive enclave code starts, the springboard must be mapped and accessed since any sensitive code is called from the springboard. The attacker can, of course, unmap the springboard in the page tables at any time. However, accesses to springboard will continue to succeed (without causing page faults) as long as the springboard's mapping is in the TLB. Furthermore, this mapping is unlikely to be evicted quickly from the TLB, as the springboard is accessed very frequently (§VIII).

All reliable methods for removing the springboard's mapping from the TLB (*e.g.*, flushing the TLB) require the attacker to run code on the enclave's core or send an inter-processor interrupt (IPI) to the core, interrupting the enclave execution eventually. The key observation is that, by construction of T-SGX, the instruction pointer value at which enclave execution will resume is on the springboard. Thus, if the attacker wants enclave execution to proceed, he will have to map the springboard in the page tables before resuming the enclave (`ERESUME`). The instruction at which enclave execution resumes will be on the springboard, and its execution will establish a new TLB entry for the springboard, which sets the attack back to the beginning.

In summary, while the attacker can interrupt enclave execution asynchronously, he cannot use the page-fault channel to obtain deterministic notification of accesses to the springboard.

VIII. Evaluation

We evaluate T-SGX by answering the following questions.

- How general is the T-SGX approach? Can this approach be applied to a wide range of legacy real world applications without manual effort?
- What are the performance characteristics of T-SGX-based programs?
- What is the performance impact of running multiple instances of T-SGX-based applications simultaneously?

Experimental setup. The experiments were conducted on a generic PC with a Supermicro X11SSQ motherboard, an Intel Core i7-6700K 4 GHz (Skylake) CPU, and 64 GB of RAM. The machine ran Windows 10 Pro. We disabled hyperthreading because avoiding cache-timing attacks in the public cloud is recommended.

Target applications. We evaluate T-SGX by using the programs in the nbench benchmark suite and the three applications that were used by Xu *et al.* [65] to demonstrate the controlled-channel attack. Table I describes each program in detail, including source code size, description, and binary code size before and after applying T-SGX. The applications are fairly diverse, including cryptography, text processing, and image compression. While the nbench applications are generally small, the other three applications are one to two orders of magnitude larger, with FreeType exceeding 100,000 lines of code.

A. Application Binaries

This section shows various properties of T-SGX binaries. The main effort in obtaining these binaries lies in porting the applications into the SGX environment. Once we had working SGX applications, no further manual effort was required to apply the T-SGX protections.

After manually adapting the source code of each application to run on SGX (resolving header and linker dependencies), we compiled the code with Clang-Cl, a `cl.exe`-compatible driver mode program for Clang (based on LLVM version 3.7.1). We linked the resulting object files into executables with the Microsoft linker (`link.exe`) version 14.00.23506.0.

We built three versions of each application. (a) The baseline version runs in an SGX enclave without any protection. (b) The TSX-basic version is secured with TSX on SGX, yet without any optimization. (c) The T-SGX version is secured with TSX and optimized as described in §V-D. These optimization techniques improve performance without affecting security.

1) Execution Block Counts and Code Size: We first measure basic statistics of each application, in particular, static information such as the number of execution blocks and the impact on code size. Table I shows the results. The reported code sizes are the sizes (in bytes) of the code (`.text`) segments of all object files associated with the application. In the case of nbench where several applications share the same source file (and the same object file), we built per-application versions of nbench by commenting out all source code that did not belong in the application.

The code size increase (excluding the springboard page) from baseline to T-SGX varies between 15% and 32%. These overheads will likely result in somewhat increased pressure on the L1 instruction cache. However, there will be no effect on the application's data accesses. Thus, the increase in the overall memory requirements will be significantly lower than 30%, depending on the application and its inputs.

The table also reports the number of execution blocks in T-SGX. Dividing the size increase by the number of execution blocks reveals an average size increase of 9 to 17 bytes per execution block. This is roughly the space needed to store the two additional instructions that jump to the springboard and the occasional instructions to save and restore `rax` (§VI).

2) Distribution of Execution Block Sizes: The next measurement studies the size of execution blocks. Figure 10 displays the distribution of the number of instructions per execution block for T-SGX and TSX-basic across the 10 nbench applications.

We observe that the optimizations of §V-D have noticeably shifted the distribution for T-SGX toward larger blocks. The small blocks (containing at most 10 instructions) are mostly the result of (a) non-mergeable cases, such as a block immediately before or after a loop, (b) nested loops, and (c) calls to functions that may not satisfy the cache constraint.

We manually inspected two large outlier blocks (up to 120 instructions). Both correspond to functions that were merged into a single execution block by our optimizations.

Application	LoC	Description	#exec. blocks	Code segment size Baseline	Code segment size T-SGX	Memory Overhead	Avrage increase bytes per block
numeric sort	211	Numeric heap sort	23	1,014 B	1,276 B	25.8%	11.4 B
string sort	521	String heap sort	46	2,745 B	3,358 B	22.3%	13.3 B
bitfield	225	Bit operations	24	1,182 B	1,472 B	24.5%	12.1 B
fp emulation	1,396	Floating-point emulation	80	5,636 B	6,467 B	14.7%	10.4 B
fourier	235	Signal processing	20	1,163 B	1,386 B	19.2%	11.2 B
assignment	490	Assignment algorithm	92	3,605 B	4,758 B	32.0%	12.5 B
idea	353	Crypto	36	3,101 B	3,553 B	14.6%	12.6 B
huffman	448	Compression	44	2,960 B	3,648 B	19.2%	15.6 B
neural net	746	Back-propagation network simulation	82	4,183 B	4,941 B	18.1%	9.2 B
lu decomposition	441	Linear equations solving algorithm	62	3,307 B	4,136 B	25.1%	13.4 B
AVERAGE						22.0%	
libjpeg (9a)	34,763	JPEG library	4,557	272,881 B	350,274 B	28.4%	17.0 B
Hunspell (1.5.0)	24,794	Spell checking library	8,641	356,298 B	471,617 B	35.0%	13.3 B
FreeType (2.5.3)	135,528	Font rendering library	12,060	615,862 B	796,105 B	29.3%	14.9 B
AVERAGE						28.6%	

TABLE I: Benchmark programs (top) and applications (bottom) used to evaluate T-SGX.

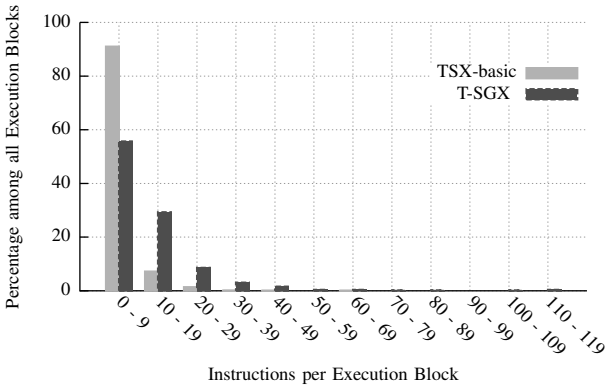

Fig. 10: Distribution of execution block sizes: The optimizations increase the size of a typical execution block.

	TX	CON	CAP	Abort Rate
numeric sort	481 times/s	20 times/s	0 times/s	0.0020%
string sort	317.3 times/s	5.3 times/s	0 times/s	0.0020%
bitfield	532 times/s	2.3 times/s	0 times/s	0.0120%
fp emulation	314 times/s	8.5 times/s	0 times/s	0.0006%
fourier	221.5 times/s	1.5 times/s	0 times/s	0.0006%
assignment	572.5 times/s	13.5 times/s	0 times/s	0.0020%
idea	707 times/s	9.5 times/s	0 times/s	0.0160%
huffman	530.7 times/s	8 times/s	0 times/s	0.0013%
neural net	485.5 times/s	35.2 times/s	0 times/s	0.0015%
lu decomposition	480 times/s	27.3 times/s	0 times/s	0.0016%

TABLE III: Rate and type of transaction aborts for the nbench applications for T-SGX.

B. Run-time Performance

This section demonstrates the run-time performance of T-SGX. Unless stated otherwise, measurement values are averaged over five runs of nbench and the real applications. For the nbench suite, we ran each program for five second and measured the number of iterations per second. For jpeglib, we measured how long it takes to decompress a 1220×813 (203,446 bytes) compressed jpeg image. The size of the decoded image is 8,926,740 bytes. The measurement includes the image decompression time but not general startup and initialization. For Hunspell, we picked the book *Around the World Eighty Days* as the input. The number of words extracted from the book is 63,704. We performed a spell check (using `Hunspell::spell`) on these words with the "en_US" dictionary as a single call into the enclave and measured the total time spent. We used the same input for FreeType. The number of characters in the book is 375,338. We measured the time required for a single enclave call that renders all these characters (using `FT_Load_Char`).

1) Run-time Overhead: Table II displays the run-time of the baseline, TSX-basic, and T-SGX versions of the applications and the associated overheads. We took the numbers for the nbench applications directly from the nbench outputs.

The overhead of T-SGX ranges from 4% to 118% with a geometric mean of 50%. While this overhead is significant, it does not appear prohibitive. The table also demonstrates the effectiveness of the optimizations of §V-D. Without these optimization techniques, the overhead would have been significantly higher (as high as 17.9×). It seems that additional optimization could reduce the overhead even further.

2) Transaction Properties: Table III displays the rate at which T-SGX transactions are aborted and the reason, as indicated by the value of the `eax` register at the time of the abort. We observe up to about 500 aborted transactions per second with an `eax` value of 0, indicating an interrupt or exception. This rate follows closely the per-core interrupt arrival rate, which we observed using Windows performance counters.

We observed no transaction abort with `eax` bit 3 set (CAP). This bit is set if an "internal buffer overflowed," which includes the case when a transaction's read or write set does not fit into the corresponding caches. This confirms the conservative nature of our cache model. We also observed small numbers of aborted transactions with `eax` bits 1 and 2 set (CON). These bits indicate "transaction may succeed on retry" and "another logical processor conflicts with read or write set," respectively.

3) Transaction Duration: Figure 11 displays the distribution of transaction durations. We measured the duration of each transaction by instrumenting the springboard code that begins and ends transactions with `rdtsc` instructions. As the `rdtsc` instruction is illegal under SGX and entering and leaving enclaves add significant noise to the measurement, we performed this experiment by running the applications outside SGX enclaves.

Application	Baseline	TSX-basic	(overhead)	T-SGX	(overhead)
numeric sort	12,682 iter/s	1,149.1 iter/s	(9.1×)	8,390.1 iter/s	(1.5×)
string sort	8,872.3 iter/s	1,991.1 iter/s	(4.1×)	7,218.7 iter/s	(1.2×)
bitfield	516,000,000 iter/s	26,100,000 iter/s	(17.9×)	443,000,000 iter/s	(2.1×)
fp emulation	319.8 iter/s	25.3 iter/s	(11.9×)	146.4 iter/s	(2.2×)
fourier	186,000 iter/s	31,847 iter/s	(5.4×)	98,847 iter/s	(1.9×)
assignment	1,741.9 iter/s	82.6 iter/s	(18.4×)	1,196 iter/s	(1.5×)
idea	3,814.1 iter/s	275.3 iter/s	(13.0×)	3,665.7 iter/s	(1.0×)
huffman	3,264.7 iter/s	162.6 iter/s	(16.6×)	1,641.5 iter/s	(2.0×)
neural net	45.7 iter/s	3.8 iter/s	(11.1×)	27.3 iter/s	(1.7×)
lu decomposition	1,197.6 iter/s	82.4 iter/s	(13.6×)	883.4 iter/s	(1.4×)
GEOMEAN			11.0×		1.5×
libjpeg	6,784.5 kB/s	846.4 kB/s	(8.0×)	4,674.1 kB/s	(1.5×)
Hunspell	176,000 word/s	36,333.3 word/s	(4.9×)	114,000 word/s	(1.6×)
FreeType	37,747.2 char/s	3,047.7 char/s	(12.4×)	28,394.5 char/s	(1.3×)
GEOMEAN			7.8×		1.4×

TABLE II: Run-time overhead of TSX-basic and T-SGX over baseline.

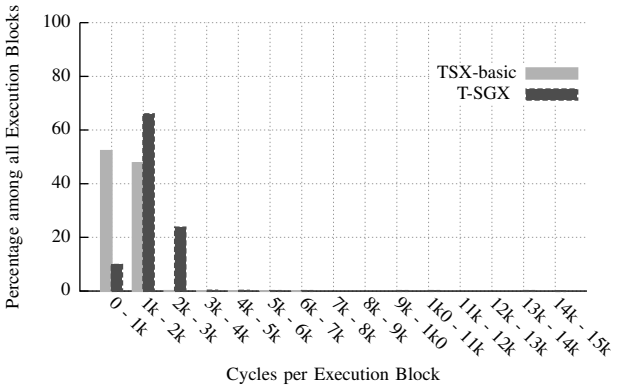

Fig. 11: Distribution of transaction times: Most transactions take less than 3,000 cycles.

Figure 11 shows the distribution of transaction duration for T-SGX and TSX-basic. For TSX-basic, most transactions take less than 1,000 cycles. As a result of the optimizations, a typical transaction for T-SGX takes between 1,000 and 2,000 cycles. Still, the transaction duration is short enough to easily meet the execution time constraint imposed by the interrupt frequency. For example, our 4 GHz processor should be able to complete 2,000-cycle-transactions even for interrupt rates of up to 2 million interrupts per second per core. Such a rate is orders of magnitude higher than the interrupt rates we have observed under normal conditions (thousands of interrupts per second per core). This observation is also consistent with Table III.

4) Transaction Abort Counts: We study the number of times a transaction aborts before it succeeds. We gather these counts by instrumenting the TSX management code on the springboard.

Table IV displays the distribution of abort counts across the 10 nbench applications. The overwhelming majority of transactions succeeds on the first try. A tiny fraction of transactions requires up to three retries. After executing many millions of transactions, we observed no transaction requiring more than three retries to complete. This observation can be used as the basis for a mechanism to detect attacks or anomalies.

Number of aborts	Percentage
0	99.9%
1	$1.7 \cdot 10^{-3}$%
2	$9.8 \cdot 10^{-6}$%
3	$3.7 \cdot 10^{-7}$%
4	0 %

TABLE IV: Distribution of the number of times a transaction aborts before it succeeds.

5) Multiple instances: The next experiment analyzes the performance of multiple T-SGX-protected enclaves running side by side. Our goal is to analyze whether T-SGX scales to multiple protected enclaves.

We measured the running time of baseline and T-SGX for the nbench applications, varying the number of concurrent instances from one to eight. For each measurement, we created n identical enclaves in n separate Windows processes ($n \in \{1, \ldots, 8\}$) running one of the 10 nbench applications for baseline or T-SGX and recorded the timing output of nbench for one of the enclaves. We repeated the measurement for n from 1 to 8, for all 10 nbench applications and for both configurations (baseline and T-SGX).

Figure 12 displays the results. The x-axis displays the number of concurrent instances. Each line corresponds to one nbench application. The y-value is the ratio of the number of iterations per second for T-SGX and for baseline. In other words, it is the inverse of the overhead. All lines are roughly constant, indicating that one can run multiple T-SGX-protected enclaves without affecting the overhead.

IX. Discussion

In this section we explain limitations of T-SGX and possible approaches to overcome them. Also, we explain other potential attacks against T-SGX and show how we can cope with them.

A. Limitations

One limitation of T-SGX is that it cannot correctly identify what causes an exception. A transactional execution aborts when an exception has been generated, but it does not let a program

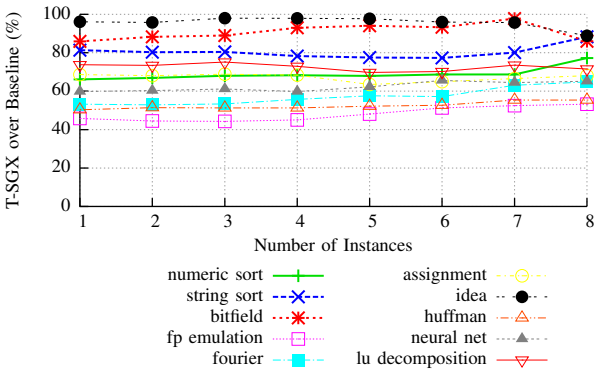

Fig. 12: Overhead with increasing number of parallel instances. It shows that T-SGX can be scaled for system-wide uses in practice.

know the vector number of the exception (§II-C2). T-SGX can distinguish a synchronous exception from an asynchronous exception by repeatedly executing a transactional region, but it cannot know whether the synchronous exception is a page fault, a divide-by-zero error, or something else. This could be a problem because T-SGX may mistake errors in the enclave software for an attack by the OS. To avoid this problem, we plan to develop an application exception handler (§II-C1) running inside an enclave that dynamically inspects the code and execution status to know the exception reason and to fix it to ensure continuous execution. Another limitation is that T-SGX cannot protect libraries without source code because it is a compiler-based approach. This problem could be solved when library developers apply T-SGX to their closed-source libraries. Also, we plan to improve T-SGX to support binary instrumentation. Another limitation is that T-SGX does not support page-level swapping between enclave memory and main memory, as Sanctum [11] does. This limitation would be problematic, especially when T-SGX runs in the public cloud while sharing the limited enclave memory with other processes. One possible solution to this problem is to swap out the whole memory of an enclave program to the main memory. We plan to study the effectiveness of this approach in the future. Lastly, T-SGX cannot support a multithreaded enclave program that wants use TSX for its original purpose: lock elision. However, this does not hurt the program's functionality because lock elision is just an optional feature. Instead, it can use a traditional lock for synchronization between different threads without any problem.

B. Other Side-channel Attacks

Cache timing attack. A cache timing attack by a malicious OS is a serious threat because the OS manages the virtual address mapping of every program [10]. To mitigate the threat, an enclave program needs to flush its private cache whenever the OS resumes its execution, but, generally, it cannot obtain such information. Fortunately, with T-SGX, an enclave program can know exactly when it is resumed by the OS such that it only needs to flush its private cache at that point. However, this mitigation is not enough to cope with asynchronous cache timing attacks that use the last-level cache (LLC) [28, 39]. We plan to study how to secure enclave programs from such attacks.

Memory bus snooping attack. A memory bus snooping attack is a hardware-level attack. By monitoring memory bus traffic, a malicious peripheral device can know which memory addresses are currently accessed by a CPU although the memory contents are encrypted by SGX. To prevent such an attack, SGX needs to provide software-level or hardware-level ORAM techniques [12, 38, 40, 49]. Also, we can minimize the number of memory accesses as much as possible by using cache-based [8, 17, 18, 66] or register-based [15, 45] computations.

X. Related Work

In this section, we discuss a number of important studies that are related to T-SGX.

Trusted execution environments. Mainstream computing environments are typically very complex. They provide only limited assurance for confidentiality and integrity in light of various attacks such as malware, kernel exploits, and malicious peripherals. Numerous researchers and companies have proposed a variety of TEEs to protect critical data and computations with higher assurance. TEEs typically do not trust the main OS because it could be compromised. Thus, they are implemented in places that even the OS cannot control, such as a trusted hypervisor or hardware. For example, Overshadow [7], NOVA [57], TrustVisor [43], Cloud Terminal [42], InkTag [21], MiniBox [37], and Sego [33] are TEEs based on trusted hypervisors. The basic idea of these systems is to provide isolated memory for each trusted process or module by using nested page tables or the extended page table feature of hardware-based virtualization. Also, all the interactions between a trusted process and the OS (*i.e.*, system calls) have to be managed by the trusted hypervisor. However, a hypervisor is also software and potentially vulnerable to various attacks [64]. Flicker [44] and TrustVisor [43] use trusted hardware (TPM [60]) and attempt to minimize the complexity of their TEE software. ARM's TrustZone [1], Intel's TXT [16] and SGX [24], and Samsung's KNOX [50] are widely-deployed hardware-based TEEs. Numerous researchers have proposed hardware-based TEE designs, such as TrInc [35], SICE [3], SecureSwitch [58], OASIS [47], TrustLite [32], and Sanctum [11].

OS attacks against TEEs. Although TEEs are designed to protect user processes from a malicious OS, the latter still has opportunities to attack the processes because they cannot access system resources (*e.g.*, storage, network) without the help of the OS. Iago attacks [6] exploit this limitation. For example, an Iago attack may manipulate the return value (*i.e.*, a virtual address) of the `mmap()` system call to make a target application overwrite a portion of its stack and, thereby, hijack control flow. Since any system call could potentially be exploited for this type of attack, the TEE has to carefully validate the return values of all system calls [21, 29]. The controlled-channel attacks [54, 65] this paper focuses on also rely on the fact that the OS manages system memory. Finally, AsyncShock [63] demonstrates that synchronization bugs that are mostly harmless in a traditional environment can allow an adversarial OS to compromise SGX enclaves.

SGX applications. Among the various hardware-based TEEs, Intel SGX recently has been receiving much attention because it is widely deployed (all Intel Skylake CPUs support it) and

because it allows developers to use almost the full unprivileged instruction set of the Intel CPU. For example, Haven [4], Graphene-SGX [61, 62], and SCONE [2] are SGX-based platforms to securely run an unmodified application in an untrusted cloud. VC3 [51], M2R [12], and Ohrimenko *et al*. [46] use SGX to perform data analytics, MapReduce computations, and machine learning computations while ensuring confidentiality and integrity. Also, Kim *et al*. [31], S-NFV [53], Pires *et al*. [48], and SecureKeeper [5] show how we can use SGX for securing network services, content-based routing, and distributed computing. Moat [56] and CONFIDENTIAL [55] design verification methodologies for enclave programs to check whether they are secure. OpenSGX [29] is an emulator for the execution of enclave programs for software development and in-depth debugging and testing. SGX-Shield [52] implements fine-grained address space layout randomization (ASLR) for SGX. Ryoan [22] introduces a distributed two-way sandbox to run untrusted enclave programs with sensitive user data while preventing possible information leakage.

XI. Conclusion

Intel SGX has been considered to be one of the most promising TEE technologies. However, the controlled-channel attack [54, 65]—a noise-free side channel—has drawn its security into question. This paper introduces T-SGX, which is a secure, efficient, and practical scheme for protecting any enclave program from controlled-channel attacks. It ensures that no page fault sequence will be leaked to attackers and is an order of magnitude faster than the state-of-the-art scheme [54] without requiring manual developer effort or hardware modifications.

Acknowledgment

We thank Byoungyoung Lee for constructive discussions, the anonymous reviewers for their helpful feedback, and GTISC lab members for their proofreading efforts. This research was supported by the NSF award DGE-1500084, CNS-1563848, CRI-1629851 ONR under grant N000141512162, DARPA TC program under contract No. DARPA FA8650-15-C-7556, and DARPA XD3 program under contract No. DARPA HR0011-16-C-0059, and ETRI MSIP/IITP[B0101-15-0644].

References

[1] ARM, "Building a secure system using TrustZone technology," Dec. 2008, pRD29-GENC-009492C.

[2] S. Arnautox, B. Tarch, F. Gregor, T. Knauth, A. Martin, C. Priebe, J. Lind, D. Muthukumaran, D. O'Keeffe, M. L. Stillwell, D. Goltzsche, D. Eyers, R. Kapitza, P. Pietzuch, and C. Fetzer, "SCONE: Secure Linux containers with Intel SGX," in *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Savannah, GA, Nov. 2016.

[3] A. M. Azab, P. Ning, and X. Zhang, "SICE: A hardware-level strongly isolated computing environment for x86 multi-core platforms," in *Proceedings of the 18th ACM Conference on Computer and Communications Security (CCS)*, Chicago, Illinois, Oct. 2011.

[4] A. Baumann, M. Peinado, and G. Hunt, "Shielding applications from an untrusted cloud with Haven," in *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Broomfield, Colorado, Oct. 2014.

[5] S. Brenner, C. Wulf, M. Lorenz, N. Weichbrodt, D. Goltzsche, C. Fetzer, P. Pietzuch, and R. Kapitza, "SecureKeeper: Confidential ZooKeeper using Intel SGX," in *Proceedings of the 16th Annual Middleware Conference (Middleware)*, 2016.

[6] S. Checkoway and H. Shacham, "Iago attacks: Why the system call API is a bad untrusted RPC interface," in *Proceedings of the 18th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Houston, TX, Mar. 2013.

[7] X. Chen, T. Garfinkel, E. C. Lewis, P. Subrahmanyam, C. A. Waldspurger, D. Boneh, J. Dwoskin, and D. R. Ports, "Overshadow: A virtualization-based approach to retrofitting protection in commodity operating systems," in *Proceedings of the 13th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Seattle, WA, Mar. 2008.

[8] P. Colpa, J. Zhang, J. Gleeson, S. Suneja, E. de Lara, H. Raj, S. Saroiu, and A. Wolman, "Protecting data on smartphones and tablets from memory attacks," in *Proceedings of the 20th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Istanbul, Turkey, Mar. 2015.

[9] B. Coppens, I. Verbauwhede, K. D. Bosschere, and B. D. Sutter, "Practical mitigations for timing-based side-channel attacks on modern x86 processors," in *Proceedings of the 30th IEEE Symposium on Security and Privacy (Oakland)*, Oakland, CA, May 2009.

[10] V. Costan and S. Devadas, "Intel SGX explained," Cryptology ePrint Archive, Report 2016/086, 2016, http://eprint.iacr.org/.

[11] V. Costan, I. Lebedev, and S. Devadas, "Sanctum: Minimal hardware extensions for strong software isolation," in *Proceedings of the 25th USENIX Security Symposium (Security)*, Austin, TX, Aug. 2016.

[12] T. T. A. Dinh, P. Saxena, E.-C. Cang, B. C. Ooi, and C. Zhang, "M2R: Enabling stronger privacy in MapReduce computation," in *Proceedings of the 24th USENIX Security Symposium (Security)*, Washington, DC, Aug. 2015.

[13] A. J. Feldman, W. P. Zeller, M. J. Freedman, and E. W. Felten, "SPORC: Group collaboration using untrusted cloud resources," in *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Vancouver, Canada, Oct. 2010.

[14] C. Gentry, "Fully homomorphic encryption using ideal lattices," in *Proceedings of the 41st Annual ACM Symposium on Theory of Computing (STOC)*, 2009.

[15] J. Götzfried and T. Müller, "Armored: CPU-bound encryption for Android-driven ARM devices," in *Proceedings of the 8th International Conference on Availability, Reliability and Security (ARES)*, 2013.

[16] J. Greene, "Intel trusted execution technology," *Intel Technology White Paper*, 2012.

[17] L. Guan, J. Lin, B. Luo, and J. Jing, "Copker: Computing with private keys without RAM," in *Proceedings of the 2014 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2014.

[18] L. Guan, J. Lin, B. Luo, J. Jing, and J. Wang, "Protecting private keys against memory disclosure attacks using hardware transactional memory," in *Proceedings of the 36th IEEE Symposium on Security and Privacy (Oakland)*, San Jose, CA, May 2015.

[19] S. M. Hand, "Self-paging in the Nemesis operating system," in *Proceedings of the 3rd USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, New Orleans, LA, Feb. 1999.

[20] M. Herlihy and J. Moss, "Transactional memory: Architectural support for lock-free data structures," in *Proceedings of the 20th ACM/IEEE International Symposium on Computer Architecture (ISCA)*, San Diego, CA, USA, 1993.

[21] O. S. Hofmann, S. Kim, A. M. Dunn, M. Z. Lee, and E. Witchel, "InkTag: Secure applications on an untrusted operating system," in *Proceedings of the 18th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Houston, TX, Mar. 2013.

[22] T. Hunt, Z. Zhu, Y. Xu, S. Peter, and E. Witchel, "Ryoan: A distributed sandbox for untrusted computation on secret data," in *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Savannah, GA, Nov. 2016.

[23] Intel, "Intel software guard extensions: Intel attestation service API," https://software.intel.com/sites/default/files/managed/3d/c8/IAS_1_0_API_spec_1_1_Final.pdf.

[24] ——, "Intel software guard extensions programming reference (rev2)," Oct. 2014, 329298-002US.

[25] ——, "Intel 64 and IA-32 architectures software developer's manual," Dec. 2015.

[26] Intel, "SGX Tutorial, ISCA 2015," http://sgxisca.weebly.com/, Jun. 2015.

[27] I. Ion, N. Sachdeva, P. Kumaraguru, and S. Čapkun, "Home is safer than the cloud!: Privacy concerns for consumer cloud storage," in *Proceedings of the Seventh Symposium on Usable Privacy and Security (SOUPS)*, Pittsburgh, Pennsylvania, 2011.

[28] 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 *Proceedings of the 36th IEEE Symposium on Security and Privacy (Oakland)*, San Jose, CA, May 2015.

[29] P. Jain, S. Desai, S. Kim, M.-W. Shih, J. Lee, C. Choi, Y. Shin, T. Kim, B. B. Kang, and D. Han, "OpenSGX: An open platform for SGX research," in *Proceedings of the 2016 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2016.

[30] S. Johnson, V. Scarlata, C. Rozas, E. Brickell, and F. Mckeen, "Intel software guard extensions: EPID provisioning and attestation services," https://software.intel.com/en-us/blogs/2016/03/09/intel-sgx-epid-provisioning-and-attestation-services.

[31] S. Kim, Y. Shin, J. Ha, T. Kim, and D. Han, "A first step towards leveraging commodity trusted execution environments for network applications," in *Proceedings of the 14th ACM Workshop on Hot Topics in Networks (HotNets)*, Philadelphia, PA, Nov. 2015.

[32] P. Koeberl, S. Schulz, A.-R. Sadeghi, and V. Varadharajan, "TrustLite: A security architecture for tiny embedded devices," in *Proceedings of the 9th European Conference on Computer Systems (EuroSys)*, Amsterdam, The Netherlands, Apr. 2014.

[33] Y. Kwon, A. M. Dunn, M. Z. Lee, O. S. Hofmann, Y. Xu, and E. Witchel, "Sego: Pervasive trusted metadata for efficienctly verified untrusted system services," in *Proceedings of the 21st ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Atlanta, GA, Apr. 2016.

[34] V. Leis, A. Kemper, and T. Neumann, "Exploiting hardware transactional memory in main-memory databases," in *Proceedings of the 30th IEEE International Conference on Data Engineering (ICDE)*, Chicago, IL, Mar.–Apr. 2014.

[35] D. Levin, J. R. Douceur, J. R. Lorch, and T. Moscibroda, "TrInc: Small trusted hardware for large distributed systems," in *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, Boston, MA, Apr. 2009.

[36] J. Li, M. Krohn, D. Mazières, , and D. Shasha, "Secure untrusted data repository (SUNDR)," in *Proceedings of the 6th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, San Francisco, CA, Dec. 2004.

[37] Y. Li, J. McCune, J. Newsome, A. Perrig, B. Baker, and W. Drewry, "MiniBox: A two-way sandbox for x86 native code," in *Proceedings of the 2014 USENIX Annual Technical Conference (ATC)*, Philadelphia, PA, Jun. 2014.

[38] C. Liu, A. Harris, M. Maas, M. Hicks, M. Tiwari, and E. Shi, "GhostRider: A hardware-software system for memory trace oblivious computation," in *Proceedings of the 20th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Istanbul, Turkey, Mar. 2015.

[39] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee, "Last-level cache side-channel attacks are practical," in *Proceedings of the 36th IEEE Symposium on Security and Privacy (Oakland)*, San Jose, CA, May 2015.

[40] M. Maas, E. Love, E. Stefanov, M. Tiwari, E. Shi, K. Asanović, J. Kubiatowicz, and D. Song, "PHANTOM: Practical oblivious computation in a secure processor," in *Proceedings of the 20th ACM Conference on Computer and Communications Security (CCS)*, Berlin, Germany, Oct. 2013.

[41] P. Mahajan, S. Setty, S. Lee, A. Clement, L. Alvisi, M. Dahlin, and M. Walfish, "Depot: Cloud storage with minimal trust," in *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Vancouver, Canada, Oct. 2010.

[42] L. Martignoni, P. Poosankam, M. Zaharia, J. Han, S. McCamant, D. Song, V. Paxson, A. Perrig, S. Shenker, and I. Stoica, "Cloud Terminal: Secure access to sensitive applications from untrusted systems," in *Proceedings of the 2012 USENIX Annual Technical Conference (ATC)*, Boston, MA, Jun. 2012.

[43] J. M. McCune, Y. Li, N. Qu, Z. Zhou, A. Datta, V. Gligor, and A. Perrig, "TrustVisor: Efficient TCB reduction and attestation," in *Proceedings of the 31th IEEE Symposium on Security and Privacy (Oakland)*, Oakland, CA, May 2010.

[44] J. M. McCune, B. J. Parno, A. Perrig, M. K. Reiter, and H. Isozaki, "Flicker: An execution infrastructure for TCB minimization," in *Proceedings of the 3rd European Conference on Computer Systems (EuroSys)*, Glasgow, Scotland, Mar. 2008.

[45] T. Müller, F. C. Freiling, and A. Dewald, "TRESOR runs encryption securely outside RAM," in *Proceedings of the 20th USENIX Security Symposium (Security)*, San Francisco, CA, Aug. 2011.

[46] O. Ohrimenko, C. F. Manuel Costa, S. Nowozin, A. Mehta, F. Schuster, and K. Vaswani, "SGX-enabled oblivious machine learning," in *Proceedings of the 25th USENIX Security Symposium (Security)*, Austin, TX, Aug. 2016.

[47] E. Owusu, J. Guajardo, J. McCune, J. Newsome, A. Perrig, and A. Vasudevan, "OASIS: On achieving a sanctuary for integrity and secrecy on untrusted platforms," in *Proceedings of the 20th ACM Conference on Computer and Communications Security (CCS)*, Berlin, Germany, Oct. 2013.

[48] R. Pires, M. Pasin, P. Felber, and C. Fetzer, "Secure content-based routing using Intel Software Guard Extensions," in *Proceedings of the 16th Annual Middleware Conference (Middleware)*, 2016.

[49] A. Rane, C. Lin, and M. Tiwari, "Raccoon: Closing digital side-channels through obfuscated execution," in *Proceedings of the 24th USENIX Security Symposium (Security)*, Washington, DC, Aug. 2015.

[50] Samsung, "White paper: An overview of Samsung KNOX," 2013, enterprise Mobility Solutions.

[51] F. Schuster, M. Costa, C. Fournet, C. Gkantsidis, M. Peinado, G. Mainar-Ruiz, and M. Russinovich, "VC3: Trustworthy data analytics in the cloud using SGX," in *Proceedings of the 36th IEEE Symposium on Security and Privacy (Oakland)*, San Jose, CA, May 2015.

[52] J. Seo, B. Lee, S. Kim, M.-W. Shih, I. Shin, D. Han, and T. Kim, "SGX-Shield: Enabling address space layout randomization for SGX programs," in *Proceedings of the 2017 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb.–Mar. 2017.

[53] M.-W. Shih, M. Kumar, T. Kim, and A. Gavrilovska, "S-NFV: Securing NFV states by using SGX," in *Proceedings of the 1st ACM International Workshop on Security in SDN and NFV*, New Orleans, LA, Mar. 2016.

[54] S. Shinde, Z. L. Chua, V. Narayanan, and P. Saxena, "Preventing your faults from telling your secrets," in *Proceedings of the 11th ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, Xi'an, China, May–Jun. 2016.

[55] R. Sinha, M. Costa, A. Lal, N. P. Lopes, S. Rajamani, S. A. Seshia, and K. Vaswani, "A design and verification methodology for secure isolated regions," in *Proceedings of the 2016 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Santa Barbara, CA, Jun. 2016.

[56] R. Sinha, S. Rajamani, S. Seshia, and K. Vaswani, "Moat: Verifying confidentiality of enclave program," in *Proceedings of the 22nd ACM Conference on Computer and Communications Security (CCS)*, Denver, Colorado, Oct. 2015.

[57] U. Steinberg and B. Kauer, "NOVA: A microhypervisor-based secure virtualization architecture," in *Proceedings of the 5th European Conference on Computer Systems (EuroSys)*, Paris, France, Apr. 2010.

[58] K. Sun, J. Wang, F. Zhang, and A. Stavrou, "SecureSwitch: BIOS-assisted isolation and switch between trusted and untrusted commodity OSes," in *Proceedings of the 19th Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2012.

[59] H. Takabi, J. B. Joshi, and G.-J. Ahn, "Security and privacy challenges in cloud computing environments," *IEEE Security & Privacy*, vol. 8, no. 6, pp. 24–31, 2010.

[60] Trusted Computing Group, "Trusted platform module (TPM) summary," http://www.trustedcomputinggroup.org/trusted-platform-module-tpm-summary/.

[61] C.-C. Tsai, K. S. Arora, N. Bandi, B. Jain, W. Jannen, J. John, H. A. Kalodner, V. Kulkarni, D. Oliveira, and D. E. Porter, "Cooperation and security isolation of library OSes for multi-process applications," in *Proceedings of the 9th European Conference on Computer Systems (EuroSys)*, Amsterdam, The Netherlands, Apr. 2014.

[62] C.-C. Tsai and D. Porter, "Graphene / Graphene-SGX Library OS - a library OS for Linux multi-process applications, with Intel SGX support," https://github.com/oscarlab/graphene.

[63] N. Weichbrodt, A. Kurmus, P. Pietzuch, and R. Kapitza, "AsyncShock: Exploiting synchronisation bugs in Intel SGX enclaves," in *Proceedings of the 21th European Symposium on Research in Computer Security (ESORICS)*, Crete, Greece, Sep. 2016.

[64] Xen, "Xen security advisories," http://xenbits.xen.org/xsa/.

[65] Y. Xu, W. Cui, and M. Peinado, "Controlled-channel attacks: Deterministic side channels for untrusted operating systems," in *Proceedings of the 36th IEEE Symposium on Security and Privacy (Oakland)*, San Jose, CA, May 2015.

[66] N. Zhang, K. Sun, W. Lou, and Y. T. Hou, "CaSE: Cache-assisted secure execution on ARM processors," in *Proceedings of the 37th IEEE Symposium on Security and Privacy (Oakland)*, San Jose, CA, May 2016.