Controlled Data Races in Enclaves: Attacks and Detection

Sanchuan Chen, *Fordham University;* Zhiqiang Lin, *The Ohio State University;* Yinqian Zhang, *Southern University of Science and Technology*

https://www.usenix.org/conference/usenixsecurity23/presentation/chen-sanchuan

This paper is included in the Proceedings of the 32nd USENIX Security Symposium.

August 9–11, 2023 • Anaheim, CA, USA

978-1-939133-37-3

Controlled Data Races in Enclaves: Attacks and Detection

Sanchuan Chen[1]
Fordham University
schen409@fordham.edu

Zhiqiang Lin
The Ohio State University
zlin@cse.ohio-state.edu

Yinqian Zhang
Southern University of Science and Technology
yinqianz@acm.org

Abstract

This paper introduces *controlled data race attacks*, a new class of attacks against programs guarded by trusted execution environments such as Intel SGX. Controlled data race attacks are analog to controlled channel attacks, where the adversary controls the underlying operating system and manipulates the scheduling of enclave threads and handling of interrupts and exceptions. Controlled data race attacks are of particular interest for two reasons: First, traditionally non-deterministic data race bugs can be triggered deterministically and exploited for security violation in the context of SGX enclaves. Second, an intended single-threaded enclave can be concurrently invoked by the adversary, which triggers unique interleaving patterns that would not occur in traditional settings. To detect the controlled data race vulnerabilities in real-world enclave binaries (including the code linked with the SGX libraries), we present a lockset-based binary analysis detection algorithm. We have implemented our algorithm in a tool named SGXRACER, and evaluated it with four SGX SDKs and eight open-source SGX projects, identifying 1,780 data races originated from 476 shared variables.

1 Introduction

Trusted execution environment (TEE) such as Intel SGX allows programmers to protect application secrets in hardware-isolated enclaves, without trusting the system software such as operating systems and hypervisors. It provides the strongest security guarantee to a user level application to date: any memory reads or writes to an enclave from other software are prohibited, regardless of their privileges. The adoption of TEE in clouds has enabled a new computing paradigm known as confidential cloud computing for data analytics [48, 50], machine learning [42], and bioinformatics [27]. Today, many mainstream cloud providers offer confidential cloud services, including Alibaba Cloud's SGX VM instances [1], Microsoft Azure's confidential computing [2], Google's confidential virtual machines [3], etc.

Unfortunately, TEEs, such as Intel SGX, are not absolutely secure, due to their large attack surfaces from such as hardware (e.g., the VoltJockey attack [46], Rowhammer attack [24]), micro-architecture (e.g., cache side channels [38], speculative execution side channels [9]), operating systems (e.g., controlled side channels [7, 60], Iago attacks [8]), and applications from the enclave code itself (e.g., the buggy code to cause buffer overflow [29], use after free, double free, or null pointer deference [26]). These attacks are particularly severe since the adversary can control the underlying system software, which serves syscalls [8], handles interrupts [55] and exceptions [60], and schedules threads [59].

In this paper, we show the existence of another important attack category against TEE enclaves: *controlled data race* attacks. Controlled data race attacks are particularly interesting for two reasons: First, unlike traditional data race bugs that only occur in non-deterministic manners, a data race in SGX can be exploited deterministically to breach the security of the enclave code, because the adversary controls the OS and is in charge of thread scheduling and interrupt/exception handling. Therefore, a data race bug in the enclave code can become a security vulnerability. Second, an intended single-threaded enclave can be unexpectedly invoked by the adversary in a concurrent manner, which triggers unique interleaving patterns that would not occur in traditional settings. Hence, enclave code that is not developed to be reentrant can be exploited in controlled data race attacks.

While data race problems have been broadly studied in traditional computing environment (e.g., multi-core) [12–14, 33, 35, 41, 47], no data race detector in the enclave code has been proposed for handling the unexpected interleavings caused by a privileged attacker in TEEs. Given the privileged attackers who have the ability of creating new enclave threads and making new enclave calls, enclave code has more possible interleavings, both intended and unintended. Moreover, enclave code is primarily written in C/C++, with synchroniza-

[1] The bulk of this work was performed when this author was at The Ohio State University.

tion locks often written using inline assembly. Thus, binary analysis is more appealing than source code analysis.

Therefore, to detect the controlled data race vulnerabilities, we propose a static *reentrancy-aware* binary analysis tool named SGXRACER that systematically identifies possible shared (i.e., the racing) variables and explores both intended and unintended thread interleavings in enclave code to inspect whether there are proper synchronizations on shared variables. A data race is identified if there is a lack of synchronization primitives when TCSnum is configured to be more than one. The key idea is to assume that every ecall can run concurrently with another ecall (including itself), given a strong privileged attacker who can abuse enclave thread creation, ecall invocation, and fine-grained enclave code execution. At a high level, SGXRACER contains two phases of analysis: the variable analysis phase and the data race detection phase. The variable analysis phase recovers shared variables and lock variables from enclave code and generates locksets and lock acquisition histories. The data race detection phase considers each ecall to be possibly concurrent and performs a reentrancy-aware lockset-based data race detection.

We have implemented SGXRACER, which can analyze a variety of SGX binaries developed in different programming languages such as C/C++, and Rust. We have evaluated SGXRACER with eight open source SGX projects crawled from github.com and four popular SGX SDKs, namely, Intel SGX SDK, Open Enclave SDK, Rust-SGX SDK, and Rust EDP SDK, among which SGXRACER has identified 476 shared variables, which contribute to $1,780$ data races. Such a high alarming number of data races shows that data race in SGX enclave is indeed a serious problem. We have informed Intel, Microsoft, Baidu, Fortanix and developers of the evaluated SGX projects about the discovered data races, one of which has filed CVE-2020-5499 for the identified vulnerability.

Contributions. In short, this paper makes the following contributions:

- **Novel Attacks.** We are the first to show controlled data race attack, an attack that can be launched at the trusted component interface (*i.e.,* enclave call), and demonstrate the severity of these attacks.
- **Systematic Detection.** We present a reentrancy-aware, binary analysis based, controlled data race detection algorithm for SGX programs, with a set of enabling techniques such as shared variable analysis, lock variable analysis, synchronization primitive identification at the binary code level.
- **Empirical Evaluation.** We have implemented SGXRACER and evaluated it with eight popular open source SGX projects and also four SGX SDKs, and identified thousands of possible data races among them.

2 Background

Intel SGX Threads. Multiple threads can be executed concurrently inside an enclave. Each thread has a thread control structure (TCS) inside the enclave, which contains control fields such as the thread's execution flag, the number of TCS (i.e., TCSnum), the maximum number of TCS (i.e., TCSMaxNum), etc., specified in the configuration file.

However, thread creation inside an enclave is not supported in SGX. A thread is first created outside the enclave and then bound to a trusted thread execution context inside the enclave. The binding is controlled by untrusted code outside the enclave, and an enclave may have either binding or non-binding mode. A variety of SGX SDKs [16, 21, 37, 57] all provide synchronization primitives (e.g., sgx_spin_lock and sgx_thread_mutex_lock in Intel SGX SDK) to support multithreading within enclaves. (A summary of synchronization primitives and API functions provided by these SGX SDKs can be found in Table 5 in Appendix §A).

Concurrency Bugs. A concurrency bug may occur when multiple threads of a program run concurrently. Various concurrency vulnerabilities have been discovered over the years, such as data race [12–14, 40, 41, 47], atomicity violation [33, 35], and deadlocks [14]. These vulnerabilities are prevalent in multi-threaded programs [34] and are notoriously difficult to detect due to their nondeterministic natures.

Particularly, a data race occurs in a multi-threaded program if two threads access the same memory location without order constraints on the two accesses, and at least one access is a write. Moreover, a data race is challenging to observe since a program can exhibit different behaviors even when rerun with the same input. Furthermore, a data race often silently violates the programmer's intention without causing a crash and could be noticed much later from the root cause.

To detect a data race, there are two classical approaches: (1) lockset-based approach [47], which detects a data race if two threads access a memory location without holding a common lock, and (2) happens-before based approach [13], which detects a data race if two accesses from different threads are not ordered based on Lamport's happens-before relation [28].

3 Controlled Data Race Attacks

Threat Model. We consider an adversary who has full control of the SGX platform except the trusted components (*e.g.*, enclave and CPU). We assume that the adversary has the capability of launching, suspending, resuming, and terminating the enclaves at will. We also assume that the adversary could arbitrarily create untrusted threads outside the enclave and make ecalls to trigger the enclave's execution. Moreover, we assume the adversary has the capability of performing side-channel attacks (e.g., page faults [60], cache [6], branch shadowing [30]) against the enclave to track its control flow,

```
void ecall_0(void) {
    sgx_thread_mutex_lock(&global_mutex_0);
    global_counter = 0;
    ...
    if(global_counter == 0) foo();
    sgx_thread_mutex_unlock(&global_mutex_0);
}
void ecall_1(void) {
    sgx_thread_mutex_lock(&global_mutex_1);
    global_counter = 1;
    sgx_thread_mutex_unlock(&global_mutex_1);
}
```

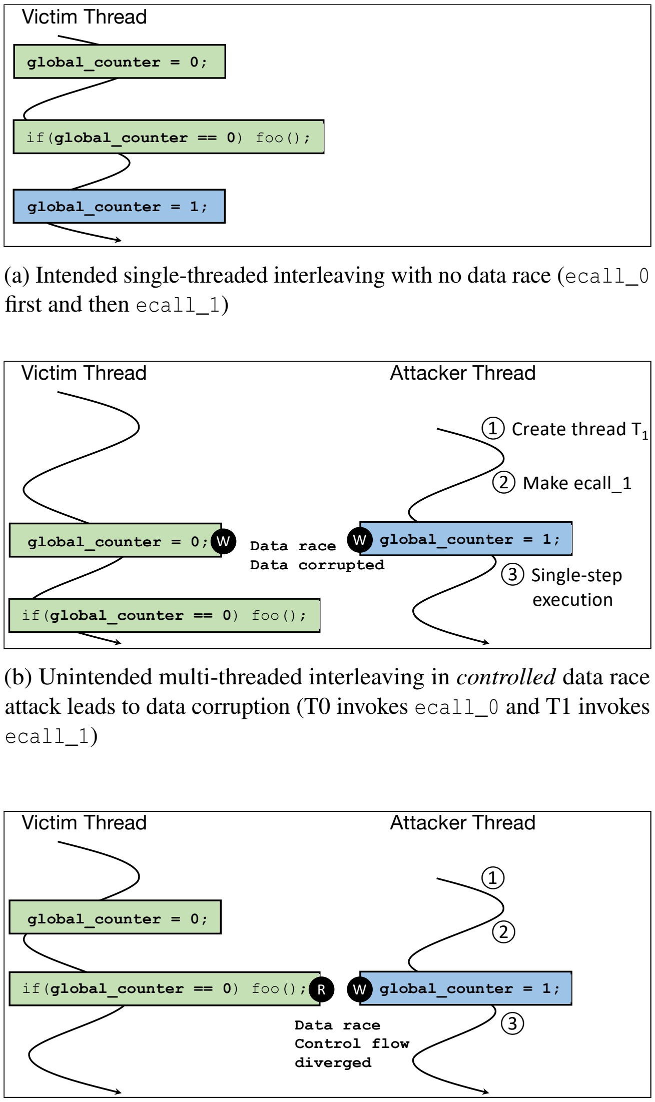
Figure 1: Working example

although we do not assume that the enclave has any secret-dependent control flow that permits leakage of the secrets through the side channels.

Controlled Data Race Attacks. A *controlled data race attack* can be launched when there is no synchronization protection for a “shared” variable and TCSnum is greater than one. As illustrated in a working example in Figure 1, two enclave calls ecall_0 and ecall_1 are intended to execute as a single-threaded program; however, a malicious attacker can create an attacker thread when the TCSnum is greater than one and perform the *controlled* data race attack.

Figure 2 shows thread interleavings for the working example. Specifically, Figure 2a shows the intended thread interleaving, in which no data race occurs, as it is a single-threaded program. Figure 2b and Figure 2c show the unintended thread interleavings, where malicious attacker creates another attacker thread to make the program multi-threaded. Figure 2b and Figure 2c illustrate two interleavings of controlled data race threads, and in the first interleaving (W-W race), the data in victim thread T_0 is corrupted by the malicious thread T_1 and we call it *data corruption attack*, and in the second interleaving (R-W race), the control flow in victim thread T_0 is diverged by the malicious thread T_1 and we call it *control flow diversion attack*.

In particular, *controlled* data race attack is carried out in three steps (as illustrated in Figure 2b): **Step** ①–attacker creates an untrusted thread which will be bound to the context of trusted enclave threads later, **Step** ②–attacker makes ecalls in each created thread at his/her choice to enable concurrent execution of trusted enclave threads, and **Step** ③–attacker forces the occurrence of a *controlled* data race by making the concurrent threads synchronize at specific program points via interrupts or (controlled) page faults.



(a) Intended single-threaded interleaving with no data race (ecall_0 first and then ecall_1)

(b) Unintended multi-threaded interleaving in *controlled* data race attack leads to data corruption (T0 invokes ecall_0 and T1 invokes ecall_1)

(c) Unintended multi-threaded interleaving in *controlled* data race attack leads to control flow diversion

Figure 2: Intended and unintended thread interleavings in working example

An End-to-end Binary Only Attack. Using SGXRACER, an attacker can analyze the enclave binary directly without accessing the source code, and further carry out a binary only controlled data race attack. We illustrate the attack by first detecting the data races in the working example, in which SGXRACER reports the vulnerable assembly lines. In addition to the victim thread, we create a malicious thread and make the enclave call. By pinning both victim and malicious threads on two designated cores, we change the enclave asynchronous enclave exit (AEX) callback routine, where we check the CPU core currently accessing the callback and set Intel Advanced Programmable Interrupt Controller (APIC) [22] timer.

We send two types of APIC timer interrupts to the victim thread and the attacker thread. One type of APIC timer inter-

```c
void aex_cb_func(void) {
  /* check attack mode */
  if(attack == 1) {
    /* get current cpu core */
    int cur_cpu = get_cpu();
    /* if thread reaches break point */
    if(erip == BREAK_POINT[cpu]) {
      /* send shorter interrupts to
          delay execution */
      if(times < EXCESSIVE_INTERRUPTS_NUM) {
      apic_timer_irq(SGX_BREAK_POINTS_TIMER_INTERVAL);
      times++;
      }
    }
    /* otherwise send stepping interrupts */
    else apic_timer_irq(SGX_STEP_TIMER_INTERVAL);
  }
}
```

Figure 3: Sending APIC interrupts in AEX call back function.

```
<ecall_0>:
...
3a3c: call a170 <sgx_thread_mutex_lock>
3a41: movq $0x0,0x20754(%rip) # 241a0 <global_counter>
...
3a4c: mov 0x2074d(%rip),%rax # 241a0 <global_counter>
3a53: test %rax,%rax
3a56: jne 3a5d <ecall_0+0x30>
3a58: call 3a22 <foo>
...
3a64: call a3d0 <sgx_thread_mutex_unlock>

<ecall_1>:
...
3a7b: call a170 <sgx_thread_mutex_lock>
3a80: movq $0x1,0x20715(%rip) # 241a0 <global_counter>
...
3a92: call a3d0 <sgx_thread_mutex_unlock>
```

Figure 4: Assembly code of the working example.

rupt has a longer interval SGX_STEP_TIMER_INTERVAL that single-steps the execution of the victim thread and the other has a shorter interval SGX_SYNC_POINTS_TIMER_INTERVAL that delays the execution of the victim thread and the attacker thread to trigger *controlled* data races.

While the victim thread executes ecall_1, the attacker thread makes ecall_2. Multiple APIC interrupts are sent to the victim thread and attacker thread to force their single-stepped executions. To handle these APIC interrupts, the threads need to perform asynchronous enclave exits (AEXs) to the outside world. In the AEX callback function, the rip register is read from the EDBGRD instruction. Once the synchronization points are encountered, excessive APIC interrupts are sent to the specific CPU core to delay the thread execution, causing the data race. The code snippet for sending APIC interrupts in the AEX callback function is listed in Figure 3. Note that the instruction under execution can also be inferred using side channels instead of EDBGRD instruction, which is out of the scope of our paper, and we leave it for future work.

Root Causes. The exploited data race is caused by *non-reentrancy*, in which the programmers first make the ecall non-reentrant, and then multiple ecalls reenter the enclave concurrently, either intended by the programmers, or unintended by malicious attackers. Both the intended and unintended ”non-reentrant” controlled data races are actually quite different from traditional data races, since they can be controlled in a deterministic manner from a layer below, namely the malicious OS. This attack is possible for the following three obvious reasons. First, thread creation in SGX is not supported by the enclave itself, and instead is controlled by OS. This is because enclave applications are partitioned in such a way [19, 32] that minimizes the size of its trusted computing base (TCB) and, meanwhile, makes thread management out of its TCB. As such, it provides a malicious OS with the capabilities to create an arbitrary number of threads to attack a victim process. Second, an enclave call (ecall) has no ordering guarantee [19] and can be called in arbitrary order. A malicious OS can deliberately create multiple threads to trigger the same ecall, so that any instruction accessing global or shared heap variables may become reentrant, potentially causing data races. Third, unlike in traditional settings, an enclave thread execution can be precisely controlled by a malicious OS using either page faults [60] or APIC timer interrupts [55, 56]. The attacker could even cause single-step enclave execution at the granularity of the instruction level, which leaves ample room for them to synchronize two threads precisely at any point of their execution and thus cause *controlled* data races.

Yet, there is still one more possibility that can cause controlled data races. A programmer may be well aware of the concurrent access of a shared variable if he or she knows that there will be two threads accessing it and will properly guard the access. However, if an SGX program is never intended for multithreading, programmers may forget to add protection of the possible shared variable access during the development. Consequently, if TCSnum is also set to be greater than one, a controlled data race attack could just target such an intended single-threaded enclave program, since the malicious OS can create multiple threads and allow each of the threads to instantiate the single-threaded enclave code (e.g., via calling ecalls). In this way, any global variable used in a single-threaded enclave program now accidentally becomes shared by these concurrent threads. It turns out that many developers (even including developers of the SGX SDK) could make such mistakes, as demonstrated in our evaluation (see §6).

4 Controlled Data Race Detection

Detecting controlled data races in enclave code imposes substantial challenges, which never rise in earlier works. Specifically, the widely used TEE implementation, Intel SGX, adopts a new threat model, where the application program is partitioned into trusted components and untrusted components, and trusted components can be invoked via ecall interface, whereas in traditional data race detector, the program is considered as a whole and follows the intended program control flow, without such a threat model in mind. Statically detecting such data races in enclave binary code requires addressing the following challenges:

Shared variables in enclave binaries. Unlike variables in program source code that are declared and obvious to identify (e.g., variable `global_mutex_0` used at line 2 in Figure 1), the variables in the binaries are hard to identify, since the symbols are all gone and the variables have all been translated into just registers and memory addresses. We have to rely on the access of memory addresses and registers to abstract the variables, based on each instruction's semantics, which has been proved to be a challenging task. Besides, to detect a data race, we have to know the specific access types (e.g., *R*-Read, or *W*-Write) to the variables, since only *R*/*W* or *W*/*W* accesses cause races.

In the literature, one promising technique to statically identify shared variables in binary is the value set analysis (VSA) [4]. Recently, there has been significant development with VSA in binary analysis (e.g., [17,58,62]), and certainly we can leverage these advances to identify shared variables when developing SGXRacer. To further differentiate the specific access of the variables, we can rely on the instruction semantics, such as "`inc %rax`" implying both a read and write access of `rax`, "`mov $1, %rax`", and "`pop %rax`" implying write accesses of `rax`, and "`cmp $0, %rax`" implying a read access of `rax`.

Lock variables in enclave binaries. After identifying the shared variables, we must further identify whether they are protected by any synchronization primitives, and then exclude synchronized memory accesses from our analysis. However, there are many synchronization primitives typically provided by SGX SDKs, such as thread once, barriers, spin locks, mutex locks, reentrant mutex locks, read-write locks, and condition variables. In addition, there are programmer defined synchronization primitives such as locks built from `LOCK` prefix (e.g., an instruction sequence "`mov $0x1,%ecx`", "`lock cmpxchg %ecx,(%rdx)`" which moves a constant value 1 to the lock variable indexed by register `rdx` and then locks it), and also the special `xchg` instruction which asserts the lock signal regardless of `LOCK` prefix.

For standard synchronization primitives, fortunately, the SGX SDK provides the corresponding APIs. For instance, as shown in our working example Figure 1, there are APIs such as `sgx_thread_mutex_lock`, and `sgx_thread_mutex_unlock`. We can identify them based on the APIs provided. To identify self-defined synchronization primitives, we can first identify the instructions with `LOCK` prefix and perform data flow analysis to identify the lock variables associated with the lock instructions and the specific lock/unlock values written to them (e.g., 0 to unlock, and 1 to lock).

Variable Name	Line#	A-LOC	PC	R/W
global_counter	3	(0x241a0, ⊥, ⊥)	0x3a41	W
global_counter	5	(0x241a0, ⊥, ⊥)	0x3a4c	R
global_counter	10	(0x241a0, ⊥, ⊥)	0x3a80	W
global_mutex_0	2	(0x24000, ⊥, ⊥)	0x3a35	R
global_mutex_0	6	(0x24000, ⊥, ⊥)	0x3a5d	R
global_mutex_1	9	(0x24040, ⊥, ⊥)	0x3a74	R
global_mutex_1	11	(0x24040, ⊥, ⊥)	0x3a8b	R

Table 1: Shared variable accesses in the working example

Unintended thread interleavings. As ecalls can be reentered arbitrarily by a malicious attacker, all ecalls become concurrent, which introduces many more possible thread interleavings than a traditional data race detector needs to consider. Intuitively, for an SGX program with m ecalls ($m > 1$), there are $\binom{m}{2} + \binom{m}{1} = \frac{1}{2}m(m+1)$[1] possible concurrent ecall combinations, some of which are not originally intended to be concurrent and now become unintended interleavings. For instance, Figure 1 is a simple enclave program which has two ecalls: `ecall_0` and `ecall_1`. In the original program, `ecall_0` and `ecall_1` are never assumed to be concurrent and should only yield the intended interleaving in Figure 2a where no data races occur on shared variable `global_counter`. However, in controlled data race attacks, the attacker can concurrently reenter `ecall_0` and `ecall_1` at a very fine-grained granularity, and can result in the unintended interleavings in Figure 2b and Figure 2c which have data races on shared variable `global_counter`.

Prior to our efforts in detecting the controlled data races in this TEE setting, earlier data race detectors (*e.g.,* [33,40]) do not have a malicious attacker in mind, and thus no notions of data races in trusted program components (*i.e.,* enclave). These detectors detect data races based on program's intrinsic control flow and leaves out the unintended reentrancy by the attackers. Our insight is to design a reentrancy-aware data race detector, which focuses all possible reentrant ecalls and explores their possible combinations. This helps efficiently detect new data races due to the enclave call reentrancy in SGX settings.

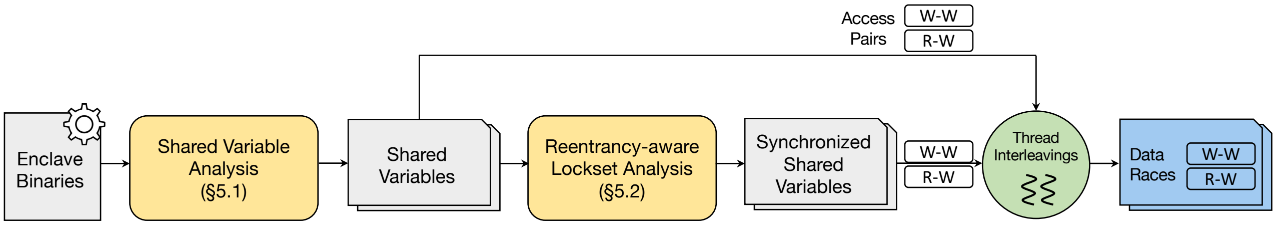


Figure 5: Overview of SGXRACER

5 Detailed Design

An overview of SGXRACER is illustrated in Figure 5. To detect a data race in enclave binaries, it has two analysis phases: (*i*) *shared variable analysis* to identify the set of all possible shared variables and their accesses, and (*ii*) *reentrancy-aware lockset analysis* to identify the set of all possible synchronization primitives protected shared variables and their accesses. The intersection of these two sets considering both the intended thread interleavings and the unintended thread interleavings will be the final detected data races.

5.1 Shared Variable Analysis

To find out data races on shared variables, we first need to detect all shared variable accesses in the enclave binary, since the data race must come from these shared variable accesses. Variables in a program can be divided into three categories: global variables, heap variables, and stack variables. Since stack variables are local, they are typically not shared. Therefore, we only need to analyze global variables and heap variables. To identify (or index) a global variable, we can use its static memory address, but for heap variables, their addresses are dynamic, and we cannot use their run-time addresses and instead we can use their allocation site to identify (index) them. Also, since a data race occurs when particular instructions access an unprotected shared variable, we have to identify the shared variables at each instruction. Thus, a data flow analysis is needed to identify the definitions (i.e., the *W* accesses) and uses (i.e., the *R* accesses) of these shared variables among the instructions.

Therefore, we have designed a data flow analysis based algorithm 1 to identify the shared variable accesses. In a nutshell, the algorithm inspects every instruction in enclave binary and finds variable definitions and variable uses (line 2-27). At each data definition site, the data flow analysis first updates the variable values according to the corresponding instruction semantics, such as data arithmetic using VSA [4] (line 4). Then at each data definition (line 5-19) and use site (line 20-27), the algorithm further identifies whether the data use of the variable is a global variable or a heap variable, whether the variable is a shared variable, and whether the access is an *R*, a *W*, or both. The output of the data flow analysis is a set of shared variable accesses, as shown in Table 1 for our working example. Next, we present in greater detail how the algorithm identifies global and heap variables.

Global variable identification. A global variable is identified at the variable definition site if the data flow analysis shows that the variable address value could be resolved to a value in binary data sections (data pointers in, e.g., `.text`) or text sections (code pointers in, e.g., `.data`, `.bss` and `.rodata`), as shown in line 7-8. Global variables are accessible from different threads and thus are shared variables on their own. Note that a global variable can be accessed via direct memory access (e.g., "`mov (0x248bb0),%rax`" where `0x248bb0` is a global memory address, or indirect access via an instruction sequence (e.g., "`lea $0x248fa0,%rax`" and "`mov (%rax),%r8`") where a global address `0x248fa0` is first loaded into `rax`, and then dererferenced to load its value into `r8`. We follow the standard data flow analysis to identify them. Having recognized the global variables from the binary, SGXRACER further distinguishes the different types of the access (i.e., *R*, *W*, or *RW*) as shown in line 9-12 and line 23-24.

Heap variable identification. Heap variables are not necessarily shared across threads, and only when they are passed through pointer references as function parameters or to a global variable. As such, SGXRACER first identifies a heap variable with depth-one context sensitivity, *i.e.,* with the call site and caller function (at line 13-14), although we can use context sensitivity with depth-two or more, we choose depth-one context for performance trade-offs. Then, SGXRACER tracks heap variables during the data flow analysis to check whether the pointer of a heap variable is passed to a shared variable (e.g., global variable or function parameter) at the variable definition sites. If so, the heap variable becomes a shared heap variable (described in line 5-6). For instance, in the following assembly code of function `do_save_tcs`:

[1] Note that $\binom{m}{2}$ denotes the total number combinations of any two independent ecalls (e.g., <`ecall_0`, `ecall_1`>), and we have to add additional $\binom{m}{1}$ ecall pairs which are those the same ecalls (e.g., <`ecall_0`, `ecall_0`>), to get the total number of all combinations.

Algorithm 1: Shared variable analysis

```
1  Function SharedVariableAnalysis(binary):
       /* Identify shared variable accesses via data flow analysis */
2      foreach instruction $i \in binary$ do
3          case $i$ defines $v$ do
4              Value(v) $\leftarrow$ Update(Value(v))
5              if $DestOp(i) \in SharedVariable \land SrcOp(i) \in HeapVariable$
6                  SharedVariable $\leftarrow$ SharedVariable $\sqcup$ SrcOp(i)
               // If defines a global variable
7              if $MemoryAccess(i) \land Addr(v) \in [global\_start, global\_end]$
8                  SharedVariable $\leftarrow$ SharedVariable $\sqcup$ v
9                  case $R \notin MemoryAccess(i) \land W \in MemoryAccess(i)$ do
10                     SharedAccess $\leftarrow$ SharedAccess $\sqcup$ (v, i, W)
11                 case $R \in MemoryAccess(i) \land W \in MemoryAccess(i)$ do
12                     SharedAccess $\leftarrow$ SharedAccess $\sqcup$ (v, i, RW)
               // If defines a heap variable
13             if $CallInst(i) \land CallTarget(i) \in AllocationFuncs$
14                 HeapVariable $\leftarrow$ HeapVariable $\sqcup$ v (callsite, caller)
15             if $v \in HeapVariable \land v \in SharedVariable$
16                 case $R \notin MemoryAccess(i) \land W \in MemoryAccess(i)$ do
17                     SharedAccess $\leftarrow$ SharedAccess $\sqcup$ (v, i, W)
18                 case $R \in MemoryAccess(i) \land W \in MemoryAccess(i)$ do
19                     SharedAccess $\leftarrow$ SharedAccess $\sqcup$ (v, i, RW)
20         case $i$ uses $v$ do
               // If uses a global variable
21             if $MemoryAccess(i) \land Addr(v) \in [global\_start, global\_end]$
22                 SharedVariable $\leftarrow$ SharedVariable $\sqcup$ v
23                 case $R \in MemoryAccess(i)$ do
24                     SharedAccess $\leftarrow$ SharedAccess $\sqcup$ (v, i, R)
               // If uses a heap variable
25             if $v \in HeapVariable \land v \in SharedVariable$
26                 case $R \in MemoryAccess(i)$ do
27                     SharedAccess $\leftarrow$ SharedAccess $\sqcup$ (v, i, R)
       /* Generate shared variable access pairs */
28     foreach $(v, i_0, acc_0) \in SharedAccess \land (v, i_1, acc_1) \in SharedAccess$ do
29         AccessPair $\leftarrow$ AccessPair $\sqcup$ ((v, $i_0$, $acc_0$), (v, $i_1$, $acc_1$))
```

```
88d4 <_ZL11do_save_tcsPv>:
...
896a: callq 100b1 <dlmalloc>
896f: mov %rax,-0x10(%rbp)
...
89a5: mov -0x10(%rbp),%rax
89a9: mov %rax,(0x248b48)#<_ZL10g_tcs_node>
```

a heap variable is allocated at instruction address `896a` but not yet a shared variable. Until at address `89a9`, when the heap variable pointer is assigned to a global variable at `0x248b48`, this heap variable becomes a shared variable. The variable access type is also inferred (at line 15-19, and line 25-27).

Generating shared variable access pairs. Based on the data flow analysis results, our shared variable analysis generates possible data race pairs (line 28-line 29), namely *shared variable access pairs*, which is a set of access pairs between two threads (i.e., thread_0 access and thread_1 access, assuming two threads) on the same shared variable. In particular, if a shared variable has n accesses, there are totally $\binom{n}{2} + \binom{n}{1}$ shared variable access pairs for this variable, i.e., a combination of different shared variable accesses $\binom{n}{2}$ plus a combination of identical shared variable accesses $\binom{n}{1}$. For instance, in our working example, the data flow analysis generates in total seven shared variable accesses, as shown in Figure 6. The total number of access pairs is 12, as $\binom{3}{2} + \binom{3}{1}$ = 6 pairs, $\binom{2}{2} + \binom{2}{1}$ = 3 pairs, and $\binom{2}{2} + \binom{2}{1}$ = 3 pairs.

5.2 Reentrancy-aware Lockset Analysis

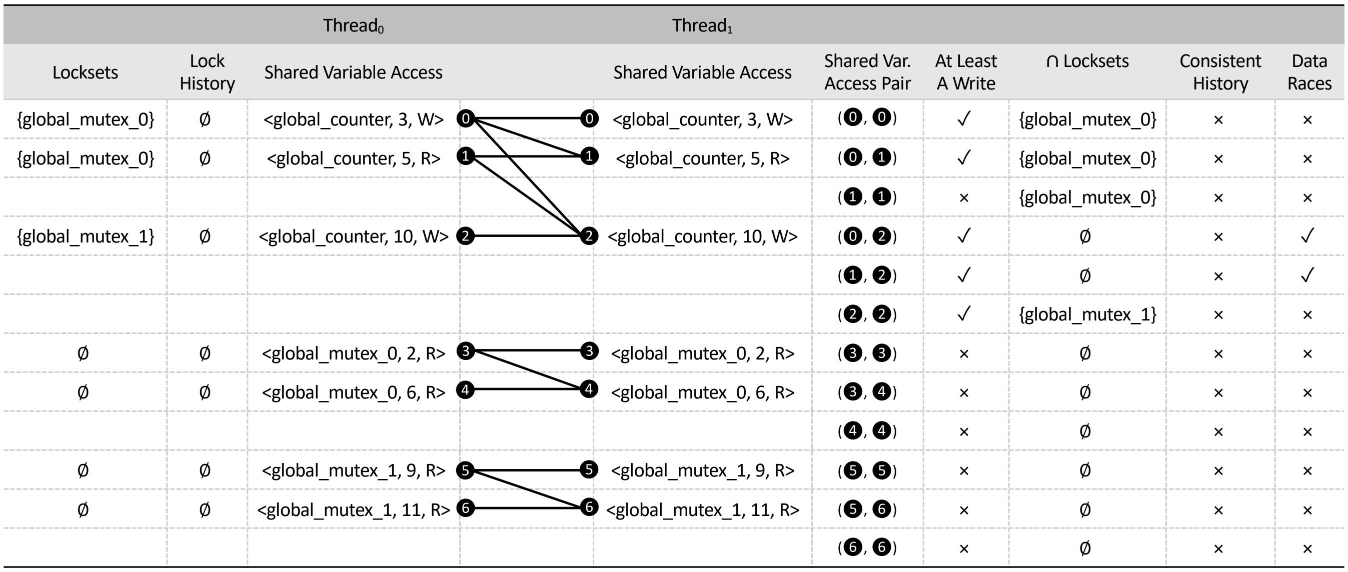
Having generated the shared variable access pairs from the variable analysis, SGXRACER further finds out whether any of them have been protected by synchronization primitives (essentially locks), and if so remove them from the detection results. Our reentrancy-aware lockset analysis first perform liveness analysis of lock variable (§5.2.1) to identify whether there is any lock associated with the shared variable accesses, and then considers all thread interleavings caused by concurrent reentrant ecalls, either intended by the programmers, or unintended by the malicious attackers and compute the corresponding lockset for the shared variable access (§5.2.2).

5.2.1 Liveness Analysis of Lock Variables

SGXRACER identifies two sets of lock variables: variables defined by the standardized synchronization APIs, and variables defined by programmers:

Identifying lock variables defined by synchronization APIs. SGX SDKs typically provide a variety of synchronization primitives (as shown in Table 5 in Appendix §A). Fundamentally, all of these synchronization primitives can be translated into a lock representation. Therefore, to have a uniformed algorithm, we have to first translate and map them into locks. Specifically, 1) thread-once is mapped as the call-once function holding a unique mutex lock; 2) barrier is mapped as holding *N* mutex locks and *N* is the number of predefined waiting threads; 3) spinlock is mapped as a mutex lock on the `spinlock` object; 4) reentrant mutex is also mapped as a mutex lock; 5) read-write lock is mapped as a read or write lock depending on whether it is used for read or write; 6) conditional variable is mapped as `lock` and `unlock` operations on the associated conditional variable.

Identifying self-defined lock variables. Besides off-the-shelf synchronization primitives, SGX program can also use other self-defined synchronization primitives, e.g., locks built from `xchg` instruction. To identify them, SGXRACER scans the binary for instructions `xchg`, `lock xchg`, `cmpxchg` and `lock cmpxchg` and considers these instructions to be a lock synchronization primitive through data flow analysis. A lock



Thread₀			Thread₁					
Locksets	Lock History	Shared Variable Access	Shared Variable Access	Shared Var. Access Pair	At Least A Write	∩ Locksets	Consistent History	Data Races
{global_mutex_0}	∅	<global_counter, 3, W> ⓿	⓿ <global_counter, 3, W>	(⓿, ⓿)	✓	{global_mutex_0}	×	×
{global_mutex_0}	∅	<global_counter, 5, R> ❶	❶ <global_counter, 5, R>	(⓿, ❶)	✓	{global_mutex_0}	×	×
				(❶, ❶)	×	{global_mutex_0}	×	×
{global_mutex_1}	∅	<global_counter, 10, W> ❷	❷ <global_counter, 10, W>	(⓿, ❷)	✓	∅	×	✓
				(❶, ❷)	✓	∅	×	✓
				(❷, ❷)	✓	{global_mutex_1}	×	×
∅	∅	<global_mutex_0, 2, R> ❸	❸ <global_mutex_0, 2, R>	(❸, ❸)	×	∅	×	×
∅	∅	<global_mutex_0, 6, R> ❹	❹ <global_mutex_0, 6, R>	(❸, ❹)	×	∅	×	×
				(❹, ❹)	×	∅	×	×
∅	∅	<global_mutex_1, 9, R> ❺	❺ <global_mutex_1, 9, R>	(❺, ❺)	×	∅	×	×
∅	∅	<global_mutex_1, 11, R> ❻	❻ <global_mutex_1, 11, R>	(❺, ❻)	×	∅	×	×
				(❻, ❻)	×	∅	×	×

Figure 6: The step-by-step internal results showing how SGXRACER detects the two data races for our working example

or unlock is identified according to the associated instruction operand. The lock or unlock semantics is based on the value of the operand (e.g., 1 means lock, and 0 means unlock).

Liveness analysis of lock variables. With the identified lock variables, we perform a liveness analysis with them. The detailed algorithm is presented in algorithm 2. More specifically, at each variable definition site, if a lock variable is defined by either standard APIs or user-defined (line 5), then we generate a GEN set (line 6). This variable will be live for the remaining instructions until it is killed by either unlock APIs or user-defined unlocks (line 13), and correspondingly we generate a KILL set (line 14). For instance, in our working example, line 2 and line 9 are call sites of `lock` synchronization function `sgx_thread_mutex_lock` and thus `global_mutex_0` and `global_mutex_1` are generated API defined lock variables, and they are live at lines 3-5 for `global_mutex_0`, and line 10 for `global_mutex_1` because they are killed at line 6 and line 11, respectively.

5.2.2 Reentrancy-aware Lockset Analysis

After identifying all lock variables in enclave binary code, we further explore the reentrancy of the enclave, *i.e.,* considering all possible thread interleavings including those caused by multiple maliciously reentered ecalls. Reentrancy-awareness is the unique feature of our lockset algorithm because threat model in SGX, specifically the enclave call (ecall) interface between the trusted and untrusted components, is not available in the target code of previous data race detector. In a nutshell, our reentrancy-aware lockset analysis first generates the lockset and lock acquisition history, and then enumerates all thread interleavings that lead to non-reentrancy and uses lockset and lock acquisition history to detect controlled data races.

Note that other than lockset, lock acquisition history is also needed for our analysis as no common lock does not necessarily lead to data race. For instance, as shown in Figure 7, the shared variable access at line 6 holds a lockset of {global_mutex_0} and the access at line 14 holds a lockset of {global_mutex_1}. The intersection of the two locksets is an empty set, but in fact there is no data race between them. Intuitively, to reach line 6, thread 0 must have acquired `global_mutex_0`, which prevents thread 1 from acquiring it in line 11, and thus line 6 and line 14 cannot execute in parallel, and no data race can occur. In the following, we formally introduce *lockset*, *lock acquisition history*, and *consistent lock acquisition histories*:

Definition 1 (Lockset). *Given thread T_i and an instruction I, we define the lockset(T_i, I) to be the possible set of locks alive at instruction I with thread T_i.*

Definition 2 (Lock Acquisition History) *Given thread T_i and an instruction I, for lock l, if $l \in$ lockset(T_i, I), then we define the lock acquisition history LockHistory(T_i, l) to be the set of locks that were acquired (and possibly released) by T_i after the last acquisition of l by T_i.*

Definition 3 (Consistent Lock Acquisition Histories). *Given two locks, l_0 and l_1, their acquisition histories are consistent if and only if there do not exist locks l_0 and l_1, such that l_0 is in lock acquisition history of l_1 and l_1 is in lock acquisition history of l_0.*

To collect lockset and lock acquisition history, SGXRACER also relies on the liveness analysis, as shown in line 6-15. In particular, for lockset, each time when a lock is defined, the

Algorithm 2: Lockset analysis

```
1  Function LockAnalysis(binary, AccessPair):
       /* Generate locksets and lock acquisition history
          via data flow analysis                                  */
2      foreach instruction $i \in binary$ do
3          if i defines v
               // Update variable value
4              Value(v) $\leftarrow$ Update(Value(v))
5          if $(CallInst(i) \wedge CallTarget(i) = lock) \vee SelfDefineLock(i)$
               // GEN lock variables
6              LockVariable $\leftarrow$ LockVariable $\sqcup$ Op(i)
               // GEN locksets
7              LockSet(i) $\leftarrow$ LockSet(i) $\sqcup$ Op(i)
8              foreach lock l do
9                  if $l = Op(i)$
                       // KILL lock acquisition history
10                     LockHistory(i,l) $\leftarrow \emptyset$
11                 else
                       // GEN lock acquisition history
12                     LockHistory(i,l) $\leftarrow$ LockHistory(i,l) $\sqcup$ Op(i)
13         if $(CallInst(i) \wedge CallTarget(i) = unlock) \vee$
             $SelfDefineUnlock(i)$
               // KILL lock variables
14             LockVariable $\leftarrow$ LockVariable $-$ Op(i)
               // KILL locksets
15             LockSet(i) $\leftarrow$ LockSet(i) $-$ Op(i)
       /* Generate synchronized shared variable access
          pairs via reentrancy-aware lockset analysis             */
16     foreach $((v, l_0, acc_0), (v, l_1, acc_1)) \in AccessPair$ do
           // check if at least one access is a write
17         if $acc_0 = R \wedge acc_1 = R$
18             SynAccessPair $\leftarrow$ SynAccessPair $\sqcup$ $((v, l_0, acc_0), (v, l_1,$
                 $acc_1))$
           // check if there are common locks
19         else if $LockSet(i) \sqcap LockSet(i) \neq \emptyset$
20             SynAccessPair $\leftarrow$ SynAccessPair $\sqcup$ $((v, l_0, acc_0), (v, l_1,$
                 $acc_1))$
           // check if lock acquistion histories are
              consistent
21         foreach different locks $l_a$, $l_b$ do
22             if $l_a \in LockHisotry(i_0, l_b) \wedge l_b \in LockHisotry(i_1, l_a)$
23                 SynAccessPair $\leftarrow$ SynAccessPair $\sqcup$ $((v, l_0, acc_0),$
                     $(v, l_1, acc_1))$
       /* Generate data races                                     */
24     DataRace $\leftarrow$ AccessPair $-$ SynAccessPair
```

liveness analysis generates (GEN) the lockset (i.e., the lock is added to the lockset), and kills (KILL) the lockset (i.e., the lock is removed from it) if the lock is unlocked. As for lock acquisition history, each time when a lock is defined, the liveness analysis kills (KILL) the lock history for this lock (set it to be empty) and generates (GEN) the lock history for other locks (add it to the lock history). In this way, the liveness analysis is able to generate lockset and lock acquisition history for each instruction. As in our working example, for instance, line 3 has the lockset of {global_mutex_0} since global_mutex_0 is the only lock acquired (at line 2) and not yet released (at line 6). LockHistory(3, {global_mutex_0}) is empty, since after the last acquisition of lock {global_mutex_0}, no locks are acquired (and possibly have been released). The lockset and lock acquisition history for each shared variable access are listed in Figure 6.

```
1  void ecall_0(void) {
2      sgx_thread_mutex_lock(&global_mutex_0);
3      sgx_thread_mutex_lock(&global_mutex_1);
4      ...
5      sgx_thread_mutex_unlock(&global_mutex_1);
6      global_counter = 0;
7      sgx_thread_mutex_unlock(&global_mutex_0);
8  }
9  void ecall_1(void) {
10     sgx_thread_mutex_lock(&global_mutex_1);
11     sgx_thread_mutex_lock(&global_mutex_0);
12     ...
13     sgx_thread_mutex_unlock(&global_mutex_0);
14     global_counter = 1;
15     sgx_thread_mutex_unlock(&global_mutex_1);
16 }
```

Figure 7: Motivating example of lock acquisition history

Reentrancy-aware Lockset-based Data Race Detection. Our reentrancy-aware lockset-based data race detection algorithm extends lockset algorithm in [25], with extra awareness of reentrancy, *i.e.,* at line 16, we assume that every access pair can be concurrent, since malicious attacker can reenter the enclave at any time. The awareness of reentrancy is one of main differences between our algorithm and prior works (*e.g.,* [33, 40]), in which more interleavings are being considered due to the unique threat model of Intel SGX. Then, for every shared variable access pair, SGXRACER checks three conditions: (1) whether one of the accesses is a write (line 17-18), (2) whether there is no common lock in the two locksets (line 19-20), and (3) whether the lock acquisition history for these two instructions is consistent (line 21-23). If all of these conditions are met, SGXRACER marks the shared variable access pair as a data race. For instance, in our working example, after counting interleavings caused by reentrancy, column six of Figure 6 lists 12 shared variable access pairs as potential data races. Among them, two data races are identified since they satisfy the three required conditions, as listed in column nine of Figure 6. Throughout this paper, the number of data races are reported along with the number of shared variables that contribute to the data races.

6 Evaluation

We have implemented SGXRACER, which currently supports four well-known SGX SDKs: Intel SGX SDK, Microsft Open Enclave SDK, Apache Teaclave Rust-SGX SDK, and Fortanix Rust EDP SDK. SGXRACER uses angr [51] to parse SGX enclave binary code and performs variable analysis and data race detection. The source code of SGXRACER

	Intel SGX	Open Enclave	Rust-SGX	Rust EDP
Variables				
# Shared Var. Access (R)	317	591	362	161
# Shared Var. Access (W)	119	214	134	21
# Shared Var. Access (R&W)	6	7	16	7
# Uniq. Shared Var.	143	197	138	81
# Lock Var. Access (Mutex)	7	9	0	20
# Lock Var. Access (Spinlock)	53	105	19	0
# Lock Var. Access (Others)	0	4	1	5
# Uniq. Lock Var.	9	14	6	1
Lockset and Acquisition History				
Ins. Lockset Size (Max.)	2	7	5	1
Ins. Lockset Size (Min.)	0	0	0	0
Ins. Lockset Size (Ave.)	0.46	0.36	0.93	0.30
Acqui. History Size (Max.)	8	13	5	0
Acqui. History Size (Min.)	0	0	0	0
Acqui. History Size (Ave.)	3.34	0.1	1.47	0
Performance (effectiveness and efficiency)				
# Shared Variables	18	32	13	2
# Data Races	39	134	28	6
FP	7	18	1	0
FP %	17.95%	13.43%	0.00%	0.00%
# Shared Variable Access Pairs	1,567	9,625	2,000	145
Variable Analysis Time (m)	508.8	12,614.4	453.6	441.2
Data Race Detection Time (m)	0.4	2.2	0.2	0.2
Total Time (m)	509.2	12,616.6	453.8	441.4

	Intel SGX	Open Enclave	Rust-SGX	Rust EDP
Variable Distribution				
Data Variable	libsgx_trts.a (4)	liboecore.a (17)	libenclave.a (1)	std::sys (1)
	libtlibc.a (4)	liboeenclave.a (3)	libsgx_trts.a (4)	std::panicking (1)
	libunwind.a (2)	liboelibc.a (2)	libtlibc.a (4)	std::thread (0)
	libcpprt.a (1)	liboesyscall.a (1)	libunwind.a (1)	std::sys_common (0)
	libirc.a (6)	libmbedcrypto.a (8)	libirc.a (1)	std::sync (0)
Total (DV)	17	31	11	2
Code Pointer	libsgx_trts.a (0)	liboecore.a (1)	libenclave.a (0)	std::sys (0)
	libtlibc.a (0)	liboeenclave.a (0)	libsgx_trts.a (0)	std::panicking (0)
	libunwind.a (1)	liboelibc.a (0)	libtlibc.a (0)	std::thread (0)
	libcpprt.a (0)	liboesyscall.a (0)	libunwind.a (1)	std::sys_common (0)
	libirc.a (0)	libmbedcrypto.a (0)	std::panicking (1)	std::sync (0)
Total (CP)	1	1	2	0
Total	18	32	13	2
Function Distribution				
Libraries	libsgx_trts.a (9)	liboecore.a (31)	libenclave.a (1)	std::sys (1)
	libtlibc.a (5)	liboeenclave.a (1)	libsgx_trts.a (8)	std::panicking (2)
	libunwind.a (5)	liboelibc.a (15)	libtlibc.a (4)	std::thread (1)
	libcpprt.a (2)	liboesyscall.a (3)	libunwind.a (3)	std::sys_common (1)
	libirc.a (5)	libmbedcrypto.a (18)	libirc.a (1)	std::sync (1)
			std::panicking (1)	alloc::sync (1)
Total	26	68	18	7

Table 2: Data race detection results for the four SGX SDKs

SGX Applications															
		# Detected Data Races				# Total	Variable	Race	Total	# Shared Var.		# Lock Var.		Ave.	Ave. Acq.
	Var.	Races	FP	FP %	Acc. Pair	Inter.	Ana. (m)	Det. (m)	Time(m)	Access	Var.	Access	Var.	Lockset	History
Cryptography															
mbedtls-SGX [61]	37	105	16	15.24%	7,817	15,317	205.2	2.4	207.6	372	84	68	3	0.198	0.000
intel-sgx-ssl [20]	138	646	7	1.08%	5,431	10,002	230.4	3.5	234	1,217	331	138	4	0.164	0.038
TaLoS [54]	148	487	0	0.00%	3,692	6,743	629.4	6.7	636	938	283	63	6	0.297	0.139
Network															
LibSEAL [31]	2	4	0	0.00%	13	20	138.6	5.4	144	930	352	0	0	0.000	0.000
Database															
SGX_SQLite [49]	95	356	3	0.84%	2,673	4,867	38.4	1.2	39.6	641	177	34	1	0.007	0.000
stealthdb [52]	2	4	1	25.00%	16	26	157.8	6.4	164.4	9	5	38	1	0.018	0.000
Learning															
SGXDeep [23]	8	24	0	0.00%	190	332	330.6	6.9	337.8	293	96	29	2	0.634	0.606
Others															
hot-calls [18]	0	0	0	0.00%	1	1	28.8	0.8	29.4	5	5	44	1	0.027	0.000

Table 3: Data race detection results for the SGX applications

has been made publicly available at `https://github.com/OSUSecLab/SGXRacer`.

To evaluate SGXRACER, we crawled 73 real-world SGX projects from `github.com`. 29 of them set `TCSnum` to one in their configuration files and hence are not vulnerable to the attacks. In the rest 44 projects, we select the project that has at least 10 stars in `github.com`, resulting in eight projects in total, and compile these eight projects with Intel SGX SDK (version 2.6) to get their final binaries for our evaluation. We also additionally analyzed other three SDK libraries from Open Enclave SDK (0.7.0), Rust-SGX SDK (1.0.8), and Rust EDP SDK (commit dbe1430) to detect whether the SDK implementations contain any data races for the possible ecalls. All of our experiments were carried out on a Dell x86-64 PC with eight Intel Core i7-7700 processors and 32GB memory.

6.1 Effectiveness

6.1.1 Detection Results

SGX SDKs. We first apply SGXRACER to detect data races in SGX SDKs. Surprisingly, among them, SGXRACER has detected in total 39 races, 134 races, 28 races, and 6 races, in these four SDKs, with 18, 32, 13, and 2 shared variables contributing to these data races, as reported in Table 2. Open Enclave SDK has far more data races than the other three SDKs, and we found that a key reason is that a shared variable, namely `mul_count` (which is a unit test variable left in binary), is involved in 55 data races, consisting 41% of its total detected data races.

To also illustrate how SGXRACER analyzes these SDKs, we also provided internal statistics, including the number of shared variable accesses, unique shared variables, lock variable accesses, unique lock variables, lockset size, etc., in Table 2. For instance, as listed in row 3–6, the number of read-only accesses of shared variable in each SDK are 317, 591, 362, and 161, respectively.

Finally, to investigate which libraries these variables and functions belong to, we also manually identified shared variables and functions involved in each data race in the source code of the four SDKs, and list the distributions in Table 2. Interestingly, some data races come from third-party libraries, e.g., libirc.a, which is a closed-source library and uses global variables without concerning thread safety. We also find that some SDKs even leave their unit test code in binary which involves data races on global variables with no lock protection. For instance, Open Enclave SDK binary has global variables such as mul_count which introduce data races. The testing code left by developers increases the trusted computing base (TCB) and attack surface, which should be avoided in a trusted execution environment such as Intel SGX. Another interesting finding is that, even though Rust-SGX reuses some code from Intel SGX SDK, as it alters the compilation configurations, Rust-SGX is compiled to a different binary, which leads to different race detection results.

SGX Applications. Next, we tested SGXRACER with eight open source projects collected from github.com. We group these projects into different categories and present the evaluation results in Table 3. As shown in this table, SGXRACER detects in total 1,626 data races and 430 contributing variables from these eight SGX projects. The SGX application with the maximum number of data races is intel-sgx-ssl (which has 646 data races with 138 contributing variables), and also there is one SGX application that has 0 data race due to the properly synchronized shared variable accesses.

Similar to the internal statistics for the SGX SDKs, we also show the internal statistics for these SGX applications in Table 3. For instance, we can notice from this table that the average number of shared variable accesses is 550.6 with a maximum of 1,217 (intel-sgx-ssl) and the average number of unique variables is 53.5 with a maximum of 352 (LibSEAL).

6.1.2 Analysis of False Positives

To determine whether there are false positives in our result, we manually inspected the source code of four SGX SDKs and eight SGX application projects. The manual inspection follows the criteria below:

- Whether the two accesses are on the same shared variable;
- Whether at least one of the two accesses is write;
- Whether the two accesses can happen at the same time.

In four evaluated SGX SDKs, we find 7, 29, 0, and 0 false positives, which resulted in a false positive rate of 8.74%. For the eight SGX projects, we find 16, 7, 0, 0, 3, 1, 0, 0 false positives, with a false positive rate of 5.27%. The overall false positive rate is 6.43%. The false positives are due to two reasons: (1) Initialization routines. For instance, an initialization routine _GLOBAL__sub_I_tmem_mgmt.cpp in SGX project intel-sgx-ssl is used to initialize the static variable addr_info_map, which cannot be executed in parallel with other accesses to this static variable. (2) Dead code. An example in Open Enclave SDK is mbedtls_ecp_self_test, which is used for unit testing in the mbedtls library. Although mbedtls_ecp_self_test is compiled into the enclave code, it cannot be called from the enclave entry. The dead code will not race with other functions. Since SGX SDKs contain more library functions with initialization routines and dead code, SDKs have higher false-positive rates than applications.

In addition to the false positives we found in manual analysis, we expect more false positives due to the following reasons. First, we use a binary analysis based approach, and we might have missed some shared variables, if they are highly sophisticated or obfuscated in the instruction access (e.g., using an array index to pass a pointer, and our data flow analysis could miss them). Fortunately, we have not witnessed such a case, and we found that identified shared variables and lock variables do not involve these sophisticated operations. Second, while SGXRACER can identify standardized synchronization primitives using SDK APIs, it could miss the unusual programmer self-defined synchronization primitives.

Heap variables. Heap variables are different from global variables (which is accessible through a global address) such that heap variables are created each time an allocation function is called. This allocation function call is identified by its call site and calling context. In SGXRACER, we use depth-one context sensitivity, *i.e.,* call sites and caller function contexts. Due to the context-sensitive analysis, we observed no false positives of heap variables in our benchmarks. We also observe no allocation function wrappers. Per our manual analysis, we identify 51, 116, 96, and 36 heap variables for four SGX SDKs and 115, 523, 461, 0, 137, 65, 153, and 22 heap variables for eight SGX applications, among which we observe no false positive explosions.

In order to evaluate the performance of SGXRACER under excessive heap allocations, we modified the working example and add 1024 heap allocation sites in two enclave call functions via calling malloc wrapper function, as shown in Figure 9. Our evaluation shows that SGXRACER does not introduce additional false positives due to excessive heap allocations (*i.e.,* 1024) thanks to the context sensitivity in our analysis. The total analysis time without excessive heap allocations is 23.671s, while with 1024 excessive heap allocations total analysis time is 26.165s, with an increase of 12.436%.

6.1.3 Analysis of False Negatives

We conducted a manual inspection of the source code of four SGX SDKs and eight SGX application projects similarly as in the analysis of false positives. However, this is extremely challenging as it requires ground truth of data races in our benchmarks. Should we have such data, we would not need to perform our analysis. We did not observe false negatives in

```
pub extern "C" fn t_global_init_ecall(id: u64,
    path: * const u8, len: usize)
{
    enclave::set_enclave_id(id as sgx_enclave_id_t);
    let s = unsafe {
        let str_slice = slice::from_raw_parts(path, len);
        str::from_utf8_unchecked(str_slice)
    };
    enclave::set_enclave_path(s);
}
```

(a) A data race on ENCLAVE_ID in Apache Teaclave Rust-SGX SDK

```
bool oe_configure_shm_capacity(size_t cap) {
    ...
    capacity = cap;
    return true;
}
void* oe_shm_malloc(size_t size) {
    ...
    shm.capacity = capacity;
    ...
}
```

(b) A data race on capacity in Open Enclave SDK

```
static uint64_t seed;


void srand(unsigned s) {
  seed = s-1;
}


int rand(void) {
  seed = 6364136223846793005ULL*seed + 1;
  return seed>>33;
}
```

(c) A data race on seed in Open Enclave SDK

```
static oe_allocation_failure_callback_t _failure_callback;
void oe_set_allocation_failure_callback(
  oe_allocation_failure_callback_t function) {
  _failure_callback = function;
}
void* oe_malloc(size_t size) {
  ...
  if (_failure_callback)
    _failure_callback(__FILE__, __LINE__,
                      __FUNCTION__, size);
  ...
}
```

(d) A data race on _failure_callback in Open Enclave SDK

```
network *final_net;
void ecall_build_network(char *file_string,
  size_t len_string, char *weights, size_t size_weights){
  ...
  network *net = (network *)malloc(sizeof(network));
  list *sections = sgx_file_string_to_list(file_string);
  net = sgx_parse_network_cfg(sections);
  ...
  final_net = net;
  ...
}
```

(e) A data race in SGXDeep

```
BASIC_CONSTRAINTS *in_bc = NULL;
void * ecall_X509_get_ext_d2i(X509 *x, int nid,
  int *crit, int *idx)
{
  void* ret = X509_get_ext_d2i(x, nid, crit, idx);
  ...
  in_bc = (BASIC_CONSTRAINTS*)ret;
  out_bc->ca = in_bc->ca;
  out_bc->pathlen = in_bc->pathlen;
  ...
}
```

(f) A data race in TaLoS

Figure 8: The code snippet in our security case studies

our code inspection. Nonetheless, four possible false negative types might occur in the evaluated SGX SDKs and applications (Table 2 and Table 3):

- Limited context sensitivity, *e.g.,* in heap variable identification, not all context sensitivity levels are considered.
- Unsolved variables, *i.e.,* shared variables that are difficult to analyze for value set analysis (*e.g.,* obfuscated binaries).
- Self-defined synchronizations, *e.g.,* self-defined synchronization primitives with `xchg`, `lock xchg`, `cmpxchg` and `lock cmpxchg` instructions.
- Vulnerable synchronizations, *i.e.,* synchronization primitives with data races (SGXRACER assumes that there are no such races). We inspected the SGX SDKs and applications binary code, but identified no such FNs. Note that it is possible in other SDKs and applications.
- Self-defined heap allocations, *i.e.,* heap variables SGXRACER misses (*e.g.,* using an unknown heap allocation function). All SGX SDKs and applications we evaluate do not use self-defined heap allocations. For instance, Apache Teaclave Rust-SGX intended to use `malloc` functions, with all other functions such as `sbrk` wrapped.

6.2 Security Case Studies

Certainly, not all data races are exploitable. Automatically detecting whether a data race is exploitable is a separate problem and requires automatic security impact analysis. Nevertheless, in the following, we demonstrate six manual case studies to show how some of these races can be exploited.

Setting two Rust-SGX enclaves with the same `ENCLAVE_ID`. In Apache Teaclave Rust-SGX SDK, SGXRACER detects a data race on shared variable `ENCLAVE_ID` in function `t_global_init_ecall`, as shown

```
void* malloc_wrapper(size_t size) { return malloc(size);}
void ecall_0(void) {
  sgx_thread_mutex_lock(&global_mutex_0);
  ...
  heapvar[0] = (char *)malloc_wrapper(64); ...;
  heapvar[511] = (char *)malloc_wrapper(64);
  sgx_thread_mutex_unlock(&global_mutex_0);
}
void ecall_1(void) {
  sgx_thread_mutex_lock(&global_mutex_1);
  ...
  heapvar[512] = (char *)malloc_wrapper(64); ...;
  heapvar[1023] = (char *)malloc_wrapper(64);
  sgx_thread_mutex_unlock(&global_mutex_1);
}
```

Figure 9: Excessive heap allocations in working example

in Figure 8a. We exploited this data race by calling initialize_enclave in two threads at the same time, which successfully makes two Rust-SGX enclaves have the same ENCLAVE_ID, as shown in Figure 10a.

Crashing oe_shm_malloc in Open Enclave SDK. In Open Enclave SDK, SGXRACER detects a data race in the shared variable capacity in function oe_shm_malloc and oe_configure_shm_capacity, as shown in Figure 8b. We set capacity to zero in another thread so that subsequent calls to the function oe_shm_malloc will fail, as shown in Figure 10b. This gives the attackers an ability to Denial of Service at the attackers' discretion.

Corrupting function srand seeds and de-randomizing function random output. In Open Enclave SDK, SGXRACER detects a data race in the shared variable seed in function srand and rand in the third party musl libc [39], as shown in Figure 8c. We kept setting the shared variable seed to a chosen value (e.g., 0x0) and successfully derandomized the return value of the rand function in another thread, as shown in Figure 10c. This shows a potential devastating security threat that data races could be used to disable pseudorandom number generation in enclave code.

Changing a callback function from another thread. The code snippet from Open Enclave SDK in Figure 8d has a data race on shared variable _failure_callback. We let one thread keep updating the failure callback function to NULL pointer. Thus, the failure callback function in the second thread will be overwritten to NULL and the thread avoids executing the intended callback function by checking its value at line 8, as illustrated in Figure 10d. This case shows that a data race could even happen on shared pointers, which might be more devastating than non-pointers.

Replacing the built network in deep learning. SGX application intel-sgx-deep-learning [23] has a data race on shared variable final_net in function ecall_build_network, as shown in Figure 8e. The write of shared variable final_net at line 9 is improperly synchronized without any lock, and a concurrent thread could replace the built network with its own network, which is later used in deep learning.

Corrupting X509 certificate extension decoding. SGX application ToLoS [54] has a data race on shared variable in_bc in function ecall_X509_get_ext_d2i as illustrated in Figure 8f. Function ecall_X509_get_ext_d2i is used for decoding X509 certificate extension with a specific OID, which further calls function X509_get_ext_d2i at line 4 and assigns the decoding results to the shared variable in_bc at line 6 without proper synchronization. A malicious thread could corrupt this buffer, leading to incorrect decoding.

6.3 Efficiency

We also report SGXRACER efficiency to analyze these benchmarks in Table 2 and Table 3. More specifically, Table 2 lists the average processing time of SGXRACER in four SGX SDKs. The total processing time is 509.2 minutes, 12,616.6 minutes, 453.8 minutes, and 441.4 minutes for four SDKs, in which the data race detection phase takes 0.4 minutes, 2.2 minutes, 0.2 minutes, and 0.2 minutes, respectively. The results show that most of the processing time is used in the variable analysis phase, which takes a long time to analyze the data flows. Table 3 lists the time overhead in eight SGX applications, and the average total time used in each application is 224.1 minutes and the maximum total time used is 636.0 minutes (TaLoS). The average variable analysis time is 219.9 minutes with a maximum of 629.4 minutes (TaLoS) and the average data race detection time is 4.2 minutes with a maximum of 6.9 minutes (SGXDeep).

7 Discussion and Future Work

Potential Defenses. The alarming number of data races detected by SGXRACER and proof-of-concept exploits developed in security case studies show the importance of defense against controlled data race attacks. An immediate defense is to set parameters such as TCSnum in Enclave configuration file to 1 if multithreading is not involved. Another defense is defensive programming [43], e.g., putting shared variables in critical sections or Intel Transactional Synchronization Extensions (TSX) transactions. Especially in our analysis, global variables are rarely guarded by synchronization primitives such as mutex locks, which could be improved to defense against controlled data race attacks. Attacks and potential defenses work similarly in Intel SGX2 [36] as in Intel SGX, as in Intel SGX version 1, attackers can already initiate new threads and allocate heap memory at application level, while SGX2 introduces more dynamic memory management support at architecture level, such as committing enclave memory and changing access permissions at run-time. But SGX2 does offer defense opportunities, *e.g.*, dynamically setting a page to non-executable when intentional data races are absent.

```c
void* thread_function(void* data) {
    initialize_enclave();
}
int SGX_CDECL main(int argc, char *argv[]) {
    ...
    /* create and initialize two threads
        at the same time */
    pthread_create(&threads[0], NULL, thread_function,
                   (void *)data);
    pthread_create(&threads[1], NULL, thread_function,
                   (void *)data);
    ...
}
```

(a) PoC for the data race on ENCLAVE_ID

```c
void ecall_thread_1()
{
    // malloc
    oe_shm_malloc(1000);

}

void ecall_thread_2()
{
    // set shm capacity to zero
    oe_configure_shm_capacity(0);

}
```

(b) PoC for the data race on capacity

```c
void ecall_thread_1()
{
    // keep setting the seed to a constant value
    while(1)
        srand(100000000000000);
}
void ecall_thread_2()
{
    while(1)
        oe_printf("rand returns: %d\n", rand());
}
```

(c) PoC for the data race on seed

```c
void unexpected_callback() {
    oe_printf("unexpected_callback triggered\n");
}
void ecall_thread_1() {
    // set the failure callback to another callback
    oe_set_allocation_failure_callback(unexpected_callback);
}
void ecall_thread_2() {
    // trigger the failure  callback
    oe_malloc(10000000000000000000000000000);
}
```

(d) PoC for the data race on _failure_callback

Figure 10: Proof-of-concept (PoC) code for exploiting the identified controlled data races

Automating the Exploitability Analysis. Since not all data races are exploitable, it is important to further filter the non-security related ones so that developers can prioritize the bug fix. While we have demonstrated in the case study that this can be done through manual analysis, it is actually a challenging task to automate. At a high level, we can notice that in order to automate the exploitability analysis, we have to first understand the execution flow that depends on the data races, and then identify the executions that can cause security damages (e.g., a data-only attack). It appears that a data flow analysis with a security impact analysis can together solve this problem, and we leave the exploration of these techniques to one of our future efforts.

Heap Function Recognition. SGXRACER needs to identify heap variables by recognizing heap related functions. Currently, it relies on list of well-known symbols of heap functions (e.g., `malloc`, `free`) to identify heap related memory accesses. However, this is not a fundamental limitation as we could adopt existing techniques such as MemBrush [11] to recognize these heap related functions in the binary code.

Handling Other Concurrency Bugs. SGXRACER only identifies data races in SGX binaries, a particular category of concurrency bugs. There are also other concurrency bugs, such as atomicity violations and deadlocks. Extending SGXRACER to detect these bugs will be another interesting avenue to work with. We believe that SGXRACER has most of the building blocks, especially with the variable analysis and the lockset analysis algorithm. The extension we envision can include the specific detection policies. For instance, to detect deadlocks, it requires semantic knowledge of the locks and how they are locked. Again, we leave these for another future work.

Improving the Precision of the Analysis. Our manual confirmation with the identified data races has yielded 7 false positives, and in theory SGXRACER could have more false positives for several reasons, as shown in §6.1. One possible direction to improve the precision of the analysis is to rely on compilers (e.g., LLVM) to only analyze the SGX programs with source code. One direction of our future work is to investigate this approach.

8 Related Work

Synchronization Vulnerabilities in SGX. AsyncShock [59] demonstrated that synchronization vulnerabilities in SGX are serious security vulnerabilities. and can be used to hijack enclave control flow or bypass access control. In particular, AsyncShock exploits existing synchronization vulnerabilities, *e.g.,* use-after-free (UAF) and time-of-check-to-time-of-use (TOCTTOU) vulnearbilities, in multi-threaded enclave code by manipulating thread scheduling via forcing segmentation faults on enclave pages. Different from AsyncShock which exploits off-the-shelf n-day synchronization vulnerabilities, SGXRACER detects 0-day vulnerable data races at 0 days instead, including unintended data races caused by arbitrarily created threads and invoked enclave calls.

Swami [53] suggested that an attacker can instantiate multiple copies of the enclave concurrently and trigger interrupts, which forces data races in the enclave code to occur, by analyzing the malleability of the enclave. In contrast, SGXRacer explores concurrent vulnerabilities at the granularity of enclave calls, based on the observation that enclave calls are made at the attacker's will. SGXRacer further automatically detects real-world data races in SGX programs, including SDKs, whereas Swami [53] only manually analyzed platform enclaves to identify the data race.

Data Race Detection. There are two categories of data race detection algorithms. One category is based on Lamport's happen-before relation [28] and checks whether shared memory location accesses from different threads are ordered by happen-before relation [13, 28, 44]. Dinning et al. [13] proposed a dynamic data race detection algorithm called task recycling, in which each program block has a unique task identifier consisting of a task and a version number. Concurrency information is recorded in a parent vector, which is a vector of version numbers. The parent vector entry and version number of two threads are compared to check whether they are concurrent. Happen-before relation-based approaches are precise but less efficient, since they require recording accesses information to each shared memory location.

The other category is the lockset-based algorithm, which detects whether two threads that access the same shared memory location have a common lock [14, 15, 25, 40, 45, 47]. Eraser [47] is a dynamic data race detector for lock-based multi-threaded programs, which maintains a set of candidate locks for each shared variable and a set of locks held by each thread. The set of candidate locks for each shared variable is constantly refined by intersection with the set of locks held by the current thread, and if the refinement result set is an empty set, Eraser triggers a warning. Kahlon et al. [25] proposed a fast and accurate static data race detection based on the lockset algorithm and the leveraged lock acquisition history to further refine the data race warnings. SGXRacer further extends the data race detection algorithm in [25] with reentrancy-awareness, in which data races caused by concurrent non-reentrant ecall invocations are also detected.

We list the comparison with related works in Table 4. To our knowledge, SGXRacer is the first static binary analysis tool for data race detection, with or without Intel SGX. As listed in Table 4, prior work focuses on source code, e.g., Java source code or C source code, while SGXRacer focuses on binary code. With this unique feature in mind, SGXRacer proposes data flow analyses such as shared variable analysis (*i.e.*, §5.1) and reentrancy-aware lockset analysis (*i.e.*, §5.2), in which both algorithms are tailored for enclave-call interface and static binary analysis.

Paper	Year	Distributed Algorithms	Data Race Detection	Happen-Before	Lockset	Target: Java Source Code	Target: C Source Code	Target: Binary Code	Dynamic	Static
Lamport et al. [28]	1978	✓		✓						
Dinning et al. [13]	1990	✓		✓						
Perković et al. [44]	1996		✓	✓					✓	
Savage et al. [47]	1997		✓		✓				✓	
Flanagan et al. [15]	2000		✓		✓	✓				✓
Engler et al. [14]	2003		✓		✓		✓			✓
Naik et al. [40]	2006		✓		✓	✓				✓
Pratikakis et al. [45]	2006		✓		✓		✓			✓
Kahlon et al. [25]	2007		✓		✓		✓			✓
Lu et al. [33]	2007		✓	✓	✓		✓			✓
SGXRacer	2023		✓		✓			✓		✓

Table 4: A Comparison with closely related works

Value Set Analysis. Value set analysis (VSA) [4,5] is a static binary code analysis technique that over-approximates the value of each variable in the binary code. VSA employs a value set to represent possible memory addresses and numeric values. The analysis has an abstract memory model that separates the address space into three kinds of memory regions, i.e., global region, stack region, and heap region, and thus the value set is often represented as a 3-tuple, where each element denotes a memory region offset.

Balakrishnan et al. [4] originally proposed value set analysis, and used it for binary code alias analysis, *i.e.*, memory accesses can be alias if their value sets intersect. DeepVSA [17] facilitated value set analysis with deep learning techniques in postmortem program analysis, where incomplete control flow information cannot provide enough context for value set analysis. A layer of instruction embedding and a bi-directional sequence-to-sequence neural network is used to infer memory regions, and therefore provides a better alias analysis.

SelectiveTaint [10] further leveraged the alias analysis ability of VSA and improved the efficiency of data flow analysis. Specifically, SelectiveTaint statically analyzed the binary code and conservatively determined the set of instructions that needs data flow tracking, and then instrumented the binary code with static binary rewriting. Different from the aforementioned approaches [4,10,17], SGXRacer mainly utilizes VSA to carry out data flow analyses, identifying shared variables, lock variables, and lock sets in enclave binary code for data race detection.

9 Conclusion

We have presented the controlled data race attack, a new attack launched deterministically by a malicious OS when enclave data is not properly guarded. To detect the vulnerable code that is subject to this attack, we also propose SGXRacer, which detects controlled data races in the enclave code, by systematically exploring possible concurrent enclave ecalls from both intended and unintended thread interleavings. We have implemented SGXRACER and evaluated it with eight open source SGX applications and four SGX SDKs, with which SGXRACER has identified totally 1,780 data races among 476 shared variables.

Acknowledgments

We thank anonymous reviewers for their invaluable feedback. We also thank researchers and engineers from Intel, Microsoft, and Baidu for their inspiring insights. We particularly thank Yu Ding for his help with Apache Teaclave Rust-SGX SDK. Zhiqiang Lin was partially supported by ARO award W911NF2110081 and DARPA award N6600120C4020. Yinqian Zhang was in part supported by Ant Group.

References

[1] Alibaba Cloud Released Industry's First Trusted and Virtualized Instance with Support for SGX 2.0 and TPM - Alibaba Cloud Community, 2022. https://www.alibabacloud.com/blog/alibaba-cloud-released-industrys-first-trusted-and-virtualized-instance-with-support-for-sgx-2-0-and-tpm_596821.

[2] Azure Confidential Computing – Protect Data-In-Use | Microsoft Azure, 2022. https://azure.microsoft.com/en-us/solutions/confidential-compute/.

[3] Introducing Google Cloud Confidential Computing with Confidential VMs | Google Cloud Blog, 2022. https://cloud.google.com/blog/products/identity-security/introducing-google-cloud-confidential-computing-with-confidential-vms.

[4] G. Balakrishnan and T. Reps. Analyzing memory accesses in x86 executables. In E. Duesterwald, editor, *Compiler Construction*, pages 5–23, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[5] G. Balakrishnan, T. Reps, D. Melski, and T. Teitelbaum. Wysinwyx: What you see is not what you execute. In *Working Conference on Verified Software: Theories, Tools, and Experiments*, pages 202–213. Springer, 2005.

[6] F. Brasser, U. Müller, A. Dmitrienko, K. Kostiainen, S. Capkun, and A.-R. Sadeghi. Software grand exposure: SGX cache attacks are practical. In *11th USENIX Workshop on Offensive Technologies (WOOT 17)*, Vancouver, BC, Aug. 2017. USENIX Association.

[7] J. V. Bulck, N. Weichbrodt, R. Kapitza, F. Piessens, and R. Strackx. Telling your secrets without page faults: Stealthy page table-based attacks on enclaved execution. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 1041–1056, Vancouver, BC, Aug. 2017. USENIX Association.

[8] S. Checkoway and H. Shacham. Iago Attacks: Why the System Call API is a Bad Untrusted RPC Interface. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, pages 253–264, New York, NY, USA, 2013. ACM.

[9] G. Chen, S. Chen, Y. Xiao, Y. Zhang, Z. Lin, and T. H. Lai. Stealing intel secrets from sgx enclaves via speculative execution. In *Proceedings of the 2019 IEEE European Symposium on Security and Privacy*, June 2019.

[10] S. Chen, Z. Lin, and Y. Zhang. Selectivetaint: Efficient data flow tracking with static binary rewriting. In *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, Aug. 2021.

[11] X. Chen, A. Slowinska, and H. Bos. Who allocated my memory? detecting custom memory allocators in c binaries. In *2013 20th Working Conference on Reverse Engineering (WCRE)*, pages 22–31. IEEE, 2013.

[12] J.-D. Choi, K. Lee, A. Loginov, R. O'Callahan, V. Sarkar, V. Sarkar, and M. Sridharan. Efficient and precise datarace detection for multithreaded object-oriented programs. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, PLDI '02, pages 258–269, New York, NY, USA, 2002. ACM.

[13] A. Dinning and E. Schonberg. An empirical comparison of monitoring algorithms for access anomaly detection. In *Proceedings of the Second ACM SIGPLAN Symposium on Principles &Amp; Practice of Parallel Programming*, PPOPP '90, pages 1–10, New York, NY, USA, 1990. ACM.

[14] D. Engler and K. Ashcraft. RacerX: Effective, static detection of race conditions and deadlocks. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, pages 237–252, New York, NY, USA, 2003. ACM.

[15] C. Flanagan and S. N. Freund. Type-based race detection for java. *SIGPLAN Not.*, 35(5):219–232, may 2000.

[16] Fortanix. Rust Enclave Development Platform, 2021. https://edp.fortanix.com/.

[17] W. Guo, D. Mu, X. Xing, M. Du, and D. Song. DEEPVSA: Facilitating value-set analysis with deep learning for postmortem program analysis. In *28th USENIX Security Symposium (USENIX Security 19)*, Santa Clara, CA, 2019. USENIX Association.

[18] hot-calls, 2017. https://github.com/oweisse/hot-calls.

[19] Intel. *Intel Software Guard Extensions Developer Guide*, June 2019. https://download.01.org/intel-sgx/linux-2.6/docs/Intel_SGX_Developer_Guide.pdf.

[20] Intel. Intel Software Guard Extensions SSL, 2021. https://github.com/intel/intel-sgx-ssl.

[21] Intel. SDK for Intel Software Guard Extensions, 2021. https://software.intel.com/en-us/sgx/sdk.

[22] Intel. *Intel Software Guard Extensions Programming Reference*, 2022. https://download.01.org/intel-sgx/sgx-linux/2.17/docs/Intel_SGX_Developer_Guide.pdf.

[23] intel-sgx-deep-learning, 2019. https://github.com/landoxy/intel-sgx-deep-learning.

[24] Y. Jang, J. Lee, S. Lee, and T. Kim. SGX-Bomb: Locking Down the Processor via Rowhammer Attack. In *Proceedings of the 2nd Workshop on System Software for Trusted Execution*, SysTEX '17, New York, NY, USA, 2017. Association for Computing Machinery.

[25] V. Kahlon, Y. Yang, S. Sankaranarayanan, and A. Gupta. Fast and accurate static data-race detection for concurrent programs. In *Proceedings of the 19th International Conference on Computer Aided Verification*, CAV'07, pages 226–239, Berlin, Heidelberg, 2007. Springer-Verlag.

[26] M. R. Khandaker, Y. Cheng, Z. Wang, and T. Wei. COIN Attacks: On Insecurity of Enclave Untrusted Interfaces in SGX. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, pages 971–985, New York, NY, USA, 2020. Association for Computing Machinery.

[27] C. Kockan, K. Zhu, N. Dokmai, N. Karpov, M. O. Külekci, D. P. Woodruff, and S. C. Sahinalp. Sketching algorithms for genomic data analysis and querying in a secure enclave. *Nature Methods*, 17:295–301, 2020.

[28] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, July 1978.

[29] J. Lee, J. Jang, Y. Jang, N. Kwak, Y. Choi, C. Choi, T. Kim, M. Peinado, and B. B. Kang. Hacking in darkness: Return-oriented programming against secure enclaves. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 523–539, Vancouver, BC, Aug. 2017. USENIX Association.

[30] S. Lee, M.-W. Shih, P. Gera, T. Kim, H. Kim, and M. Peinado. Inferring fine-grained control flow inside SGX enclaves with branch shadowing. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 557–574, Vancouver, BC, 2017. USENIX Association.

[31] LibSEAL, 2018. `https://github.com/lsds/LibSEAL`.

[32] J. Lind, C. Priebe, D. Muthukumaran, D. O'Keeffe, P.-L. Aublin, F. Kelbert, T. Reiher, D. Goltzsche, D. Eyers, R. Kapitza, C. Fetzer, and P. Pietzuch. Glamdring: Automatic application partitioning for Intel SGX. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 285–298, Santa Clara, CA, 2017. USENIX Association.

[33] S. Lu, S. Park, C. Hu, X. Ma, W. Jiang, Z. Li, R. A. Popa, and Y. Zhou. MUVI: Automatically inferring multi-variable access correlations and detecting related semantic and concurrency bugs. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, pages 103–116, New York, NY, USA, 2007. ACM.

[34] S. Lu, S. Park, E. Seo, and Y. Zhou. Learning from mistakes: A comprehensive study on real world concurrency bug characteristics. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIII, pages 329–339, New York, NY, USA, 2008. ACM.

[35] S. Lu, J. Tucek, F. Qin, and Y. Zhou. AVIO: Detecting atomicity violations via access interleaving invariants. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XII, pages 37–48, New York, NY, USA, 2006. ACM.

[36] F. McKeen, I. Alexandrovich, I. Anati, D. Caspi, S. Johnson, R. Leslie-Hurd, and C. Rozas. Intel® software guard extensions (intel® sgx) support for dynamic memory management inside an enclave. In *Proceedings of the Hardware and Architectural Support for Security and Privacy 2016*, HASP 2016, New York, NY, USA, 2016. Association for Computing Machinery.

[37] Microsoft. Open Enclave SDK, 2021. `https://openenclave.io/sdk/`.

[38] A. Moghimi, G. Irazoqui, and T. Eisenbarth. Cachezoom: How SGX amplifies the power of cache attacks. In *19th International Conference on Cryptographic Hardware and Embedded Systems - CHES 2017*, pages 69–90, 2017.

[39] musl-libc. musl-libc, 2020. `https://www.musl-libc.org/`.

[40] M. Naik, A. Aiken, and J. Whaley. Effective static race detection for Java. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '06, pages 308–319, New York, NY, USA, 2006. ACM.

[41] R. O'Callahan and J.-D. Choi. Hybrid dynamic data race detection. In *Proceedings of the Ninth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '03, pages 167–178, New York, NY, USA, 2003. ACM.

[42] O. Ohrimenko, F. Schuster, C. Fournet, A. Mehta, S. Nowozin, K. Vaswani, and M. Costa. Oblivious multi-party machine learning on trusted processors. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 619–636, Austin, TX, Aug. 2016. USENIX Association.

[43] T. Peierls, B. Goetz, J. Bloch, J. Bowbeer, D. Lea, and D. Holmes. *Java Concurrency in Practice*. Addison-Wesley Professional, 2005.

[44] D. Perkovic and P. J. Keleher. Online data-race detection via coherency guarantees. In *Proceedings of the 2nd USENIX Symposium on Operating System Design and Implementation*, 1996.

[45] P. Pratikakis, J. S. Foster, and M. Hicks. LOCKSMITH: Context-sensitive correlation analysis for race detection. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '06, pages 320–331, New York, NY, USA, 2006. ACM.

[46] P. Qiu, D. Wang, Y. Lyu, and G. Qu. VoltJockey: Breaching TrustZone by Software-Controlled Voltage Manipulation over Multi-Core Frequencies. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS '19, pages 195–209, New York, NY, USA, 2019. Association for Computing Machinery.

[47] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A dynamic data race detector for multi-threaded programs. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*, SOSP '97, pages 27–37, New York, NY, USA, 1997. ACM.

[48] 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 *2015 IEEE Symposium on Security and Privacy*, pages 38–54. IEEE, 2015.

[49] SGX_SQLite, 2018. https://github.com/yerzhan7/SGX_SQLite.

[50] F. Shaon, M. Kantarcioglu, Z. Lin, and L. Khan. A practical encrypted data analytic framework with trusted processors. In *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS’17)*, Dallas, TX, November 2017.

[51] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna. SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *IEEE Symposium on Security and Privacy*, 2016.

[52] stealthdb, 2019. https://github.com/cryptograph/stealthdb.

[53] Y. Swami. Intel SGX remote attestation is not sufficient. In *Black Hat USA 2017*, 2017.

[54] TaLoS, 2019. https://github.com/lsds/TaLoS.

[55] J. Van Bulck, F. Piessens, and R. Strackx. SGX-Step: A practical attack framework for precise enclave execution control. In *Proceedings of the 2Nd Workshop on System Software for Trusted Execution*, SysTEX’17, pages 4:1–4:6, New York, NY, USA, 2017. ACM.

[56] J. Van Bulck, F. Piessens, and R. Strackx. Nemesis: Studying Microarchitectural Timing Leaks in Rudimentary CPU Interrupt Logic. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS ’18, pages 178–195, New York, NY, USA, 2018. ACM.

[57] H. Wang, P. Wang, Y. Ding, M. Sun, Y. Jing, R. Duan, L. Li, Y. Zhang, T. Wei, and Z. Lin. Towards memory safe enclave programming with rust-sgx. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019.

[58] R. Wang, Y. Shoshitaishvili, A. Bianchi, A. Machiry, J. Grosen, P. Grosen, C. Kruegel, and G. Vigna. Ramblr: Making reassembly great again. In *Proceedings of the Network and Distributed System Security Symposium*, 2017.

[59] N. Weichbrodt, A. Kurmus, P. Pietzuch, and R. Kapitza. AsyncShock: Exploiting synchronisation bugs in Intel SGX enclaves. In I. Askoxylakis, S. Ioannidis, S. Katsikas, and C. Meadows, editors, *Computer Security – ESORICS 2016*, pages 440–457, Cham, 2016. Springer International Publishing.

[60] Y. Xu, W. Cui, and M. Peinado. Controlled-Channel Attacks: Deterministic side channels for untrusted operating systems. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy*, SP ’15, pages 640–656, Washington, DC, USA, 2015. IEEE Computer Society.

[61] F. Zhang. SGX-mbedtls, 2019. https://github.com/bl4ck5un/mbedtls-SGX.

[62] Z. Zhang, W. You, G. Tao, G. Wei, Y. Kwon, and X. Zhang. BDA: Practical Dependence Analysis for Binary Executables by Unbiased Whole-program Path Sampling and Per-path Abstract Interpretation. In *Proceedings of the ACM on Programming Languages Volume 3 Issue OOPSLA (OOPSLA 2019)*, 2019.

A Synchronization Primitives and Functions in Four SGX SDKs

A summary of the synchronization primitives and API functions provided in evaluated SGX SDKs is presented in Table 5.

SGX SDKs	Sync. Primitive	Function
Intel SGX SDK	Spinlock	sgx_spin_lock sgx_spin_unlock
	Mutex	sgx_thread_mutex_lock sgx_thread_mutex_trylock sgx_thread_mutex_unlock
	Condition Variable	sgx_thread_cond_wait sgx_thread_cond_signal sgx_thread_cond_broadcast
Open Enclave SDK	Thread-once	oe_pthread_once
	Spinlock	oe_pthread_spin_lock oe_pthread_spin_unlock
	Mutex	oe_pthread_mutex_lock oe_pthread_mutex_trylock oe_pthread_mutex_unlock
	Read-write Lock	oe_pthread_rwlock_rdlock oe_pthread_rwlock_wrlock oe_pthread_rwlock_unlock
	Condition Variable	oe_pthread_cond_wait oe_pthread_cond_signal oe_pthread_cond_broadcast
Rust-SGX SDK	Thread-once	Once::call_once Once::call_once_force
	Barrier	Barrier::wait
	Spinlock	SgxThreadSpinlock::lock SgxThreadSpinlock::unlock
	Mutex	SgxThreadMutex::lock SgxThreadMutex::trylock SgxThreadMutex::unlock SgxThreadMutex::unlock_lazy
	Reentrant Mutex	SgxReentrantThreadMutex::lock SgxReentrantThreadMutex::trylock SgxReentrantThreadMutex::unlock
	Read-write Lock	SgxThreadRwLock::read SgxThreadRwLock::try_read SgxThreadRwLock::write SgxThreadRwLock::try_write SgxThreadRwLock::read_unlock SgxThreadRwLock::write_unlock
	Condition Variable	SgxThreadCondvar::wait SgxThreadCondvar::wait_timeout SgxThreadCondvar::signal SgxThreadCondvar::broadcast SgxThreadCondvar::notify_one SgxThreadCondvar::notify_all
Rust EDP SDK	Thread-once	Once::call_once Once::call_once_force
	Barrier	Barrier::wait
	Mutex	Mutex::lock Mutex::trylock Mutex::unlock
	Reentrant Mutex	ReentrantMutex::lock ReentrantMutex::trylock ReentrantMutex::unlock
	Read-write Lock	RWLock::read RWLock::try_read RWLock::write RWLock::try_write RWLock::read_unlock RWLock::write_unlock
	Condition Variable	Condvar::wait Condvar::wait_timeout Condvar::notify_one Condvar::notify_all

Table 5: Synchronization primitives and functions in four SGX SDKs