AsyncShock: Exploiting Synchronisation Bugs in Intel SGX Enclaves

Nico Weichbrodt, Anil Kurmus†, Peter Pietzuch*, and Rüdiger Kapitza

TU Braunschweig, †IBM Research Zurich, *Imperial College London
{weichbr,kapitza}@ibr.cs.tu-bs.de,kur@zurich.ibm.com,prp@imperial.ac.uk

Abstract. Intel's Software Guard Extensions (SGX) provide a new hardware-based trusted execution environment on Intel CPUs using *secure enclaves* that are resilient to accesses by privileged code and physical attackers. Originally designed for securing small services, SGX bears promise to protect complex, possibly cloud-hosted, legacy applications. In this paper, we show that previously considered harmless synchronisation bugs can turn into severe security vulnerabilities when using SGX. By exploiting use-after-free and time-of-check-to-time-of-use (TOCTTOU) bugs in enclave code, an attacker can hijack its control flow or bypass access control.
We present *AsyncShock*, a tool for exploiting synchronisation bugs of multithreaded code running under SGX. AsyncShock achieves this by only manipulating the scheduling of threads that are used to execute enclave code. It allows an attacker to interrupt threads by forcing segmentation faults on enclave pages. Our evaluation using two types of Intel Skylake CPUs shows that AsyncShock can reliably exploit use-after-free and TOCTTOU bugs.

Keywords: Intel Software Guard Extensions (SGX); Threading; Synchronisation; Vulnerability

Accepted for publication in:
21st European Symposium on Research in Somputer Science (ESORICS 2016)
http://www.ics.forth.gr/esorics2016/

1 Introduction

Recently, Intel's Software Guard Extensions (SGX), a new hardware-supported trusted execution environment for CPUs, has reached the mass market[1]. Similarly to previous trusted execution environments such as ARM TrustZone [1],

[1] https://qdms.intel.com/dm/i.aspx/5A160770-FC47-47A0-BF8A-062540456F0A/PCN114074-00.pdf

SGX allows the execution of applications inside *secure enclaves*, without trusting other applications, the operating system (OS) or the boot process. Unlike previous solutions, SGX supports hardware multithreading, which is a fundamental requirement for modern performant applications.

Secure enclaves reduce the overall trusted computing base (TCB) to essentially the TCB of the enclave. SGX by itself, however, cannot prevent vulnerable enclave applications from being exploited. Although it was initially assumed that only small tailored applications would be executed inside enclaves [11], a recent trend is to consider enclaves as a generic isolation environment for arbitrary applications: VC3 [21] uses enclaves to secure computation for the Hadoop map/reduce framework; Haven [2] places a library OS inside an enclave for running unmodified Windows server applications.

This trend towards more complex, multi-threaded applications inside enclaves opens up new attacks. In particular, existing applications are designed to protect against a threat model that is not the same as the one for enclave code—traditional applications assume that the OS is trusted. As recent work has shown, an untrusted OS enables powerful side channel [28] and Iago [5] attacks.

In this paper, we explore a new angle for mounting attacks against SGX enclaves. We show that *synchronisation bugs* that are unlikely to be exploitable outside of SGX become reliably exploitable by carefully scheduling enclave threads. We achieve this by manipulating the page access permissions of enclave pages to force segmentation faults that interrupt enclave execution. Through this method, we are able to widen the traditionally small attack window of synchronisation bugs and increase the chances of a successful exploit.

Typically, the impact of such *concurrency attacks* [29] is to prevent or slow down certain activities in favour of others, create inconsistencies, extract data, bypass access control, or hijack the control flow of the attacked program (e.g., CVE-2009-1837, CVE-2010-5298, CVE-2013-6444). In the case of SGX, the impact of controlling code execution within an enclave is higher. At the time of writing, Intel only licenses the creation of SGX production enclaves after examination of the software development practices of the licensee[2]. Controlling enclave code execution would be a way to circumvent this practice, similarly to how "jailbroken" iPhones can execute non-Apple approved applications.

The contributions of the paper are:

- we show that synchronisation bugs are easier to exploit within SGX enclaves than in traditional applications. This is partly because, by design, the attacker can control thread scheduling of enclaves in the SGX attacker model;
- we describe AsyncShock, a tool that facilitates the reliable and semi-automated exploitation of synchronisation bugs in SGX enclaves. AsyncShock leverages the ability of the untrusted OS to arbitrarily interrupt and reschedule enclave threads. AsyncShock is designed to target enclaves built with the official SGX Software Development Kit (SDK) for Linux[3];

[2] `https://software.intel.com/en-us/articles/intel-sgx-product-licensing`

[3] `https://software.intel.com/en-us/blogs/2016/04/11/intel-software-guard-extensions-sdk-for-linux-availability-update`

– we explain how to track enclave execution near critical sections by removing permissions from pages, which triggers notifications when enclave execution has reached a particular point;
– we show how use-after-free and TOCTTOU [3] bugs can be exploited by AsyncShock; and
– we provide evaluation results of attack success rates by AsyncShock on current Intel Skylake CPUs, exploring a variety of different implementations of the attack.

The paper is structured as follows: §2 provides background on SGX, the assumed attacker model and the impact of synchronisation bugs when using SGX; §3 describes our forced segmentation fault approach and the AsyncShock tool; §4 gives evaluation results and discusses protective measures; §5 surveys related work on SGX and similar attacks; and §6 concludes the paper.

2 Background

First, we give a brief introduction to trusted execution as implemented by SGX. After that, we present an attacker model that is tailored towards typical usage scenarios of SGX. Finally, we discuss the impact of synchronisation bugs.

2.1 SGX in a Nutshell

SGX allows developers to create an isolated context inside their applications, called a secure *enclave* [18, 13]. Enclaves feature multiple properties: (i) enclaves are isolated from other untrusted applications (including higher-privileged ones) through memory access control mechanisms enforced by the CPU; (ii) memory encryption is used to defend against physical attacks and to secure swapped out enclave pages; and (iii) enclaves support remote attestation at the level of enclave instances.

Programming Model. A typical workflow for using SGX with the support of the SGX SDK [12] starts with creating an enclave as part of an application. The necessary instructions for creating an enclave are only callable from kernel mode (ring 0) and thus require kernel support. Once successfully performed, the application can issue `Ecalls` ① to enter an enclave as seen in Figure 1. Inside the enclave, input parameters passed with the call can be processed, and enclave code is executed. Developers specify the enclave interface and the direction of data with a SDK-specific file written in the Enclave Description Language (EDL) [12]. The SDK handles data movement across the enclave boundary by performing the necessary memory copy operations. However, this is only supported for primitive data types and flat structures. Data structures with pointers are not deep-copied and therefore expose the enclave to TOCTTOU attacks.

`Ocalls` ② may be performed to leave the enclave and execute untrusted application code before an `Ecall` returns ③ to the enclave. While the enclave

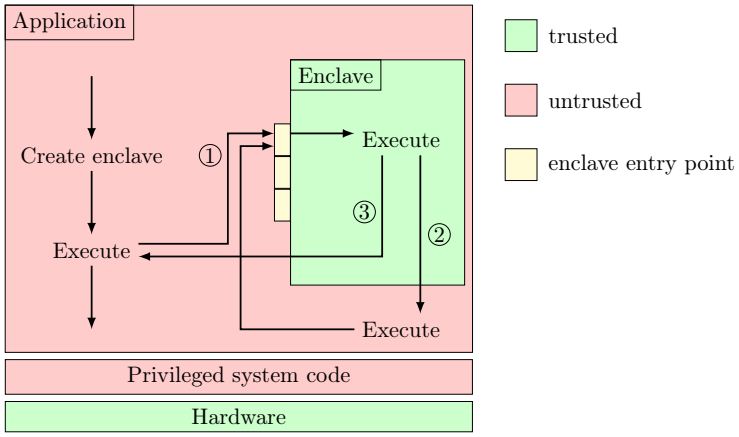

Fig. 1. Basic enclave interaction showing an `Ecall` ① into an enclave, an `Ocall` ② and the return ③ to the untrusted application. (The enclave entry points are shown in yellow.)

has access to inside and outside memory, the untrusted application is not allowed to access memory inside the enclave: any attempt to read enclave data results in abort page semantics, i.e. reading `0xFF`; write attempts are simply ignored.

Memory Management. Enclave creation and its memory layout are handled by an SGX kernel module. During enclave creation, the enclave code and data are copied page-by-page into the Enclave Page Cache (EPC), which is protected system memory. Mapped pages and their permissions are saved in the Enclave Page Cache Map (EPCM). Enclave page permissions are thus managed twice, once through the OS page table and once through the EPCM. Accessing an enclave page also leads to two permissions checks: once by the Memory Management Unit (MMU) reading the permissions from the page table, and once by SGX reading them from the EPCM. While it is possible to restrict page table permissions further using `mprotect`, it is not possible to extend them because the EPCM cannot be modified. The possibility of removing page permissions is important for AsyncShock—it means that an attacker can mark pages and get notified when they are used.

Support for Multithreading and Synchronisation Mechanisms. Each enclave must have at least one entry point that defines an address at which the enclave may be entered. The SDK implements a trampoline to allow multiple `Ecalls` through a single entry point. Multithreading is supported by having multiple entry points and permitting multiple threads to enter them concurrently. Similar to regular applications, interrupts may occur during enclave execution and must be handled. SGX achieves this by performing an Asynchronous Enclave Exit (AEX), which saves the current processor state into enclave memory, leaves the enclave and jumps to the Interrupt Service Routine (ISR). Enclave execution is resumed after the ISR finishes, restoring the saved processor state.

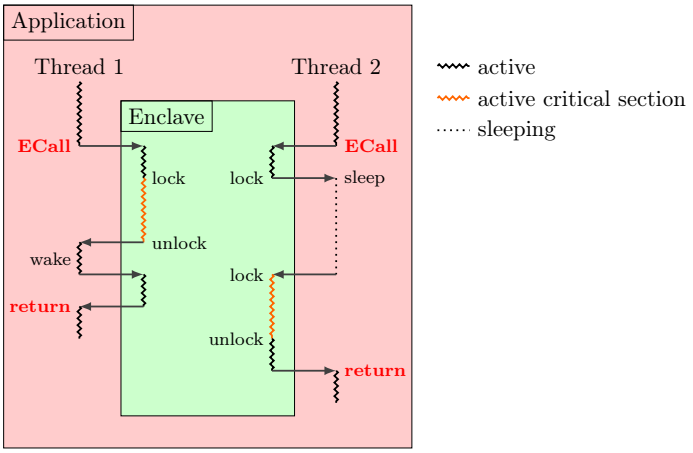

Fig. 2. Mutex lock/unlock operations provided by the SGX SDK may exit the enclave.

The SGX SDK offers synchronisation primitives such as mutexes and condition variables. These primitives do not operate exclusively inside the enclave: for instance, thread blocking requires a system call that is unavailable inside enclaves. Furthermore, managing a lock variable outside of the enclave is not advised because an attacker could change it. A hybrid approach has been adopted by Intel in which the lock variables are maintained inside the enclave whereas system calls are issued outside. Therefore, using synchronisation primitives may result in enclave exits. Figure 2 shows this behaviour for a mutex lock operation.

2.2 Attacker Model

We consider a typical attacker model for SGX enclaves: an attacker has full control over the environment that starts and stops SGX enclaves. They have full control of the OS and all code invoked prior to the transfer of control, using `Ecalls`, to the SGX enclave, and also when an enclave calls outside code via `Ocalls`. More specifically, the attacker can interrupt and resume SGX threads (see §2.1), which is the main attack vector exploited in this paper.

The attacker's goal is to compromise the confidentiality or integrity of the SGX enclave. For example, they may want to gain the ability to execute arbitrary code within the enclave. Note that we ignore availability threats, such as crashing an enclave: the untrusted OS can simply stop SGX threads.

2.3 Synchronisation Bugs in Software

Synchronisation bugs are caused by the improper synchronised access of shared data by multiple threads, and previous studies have shown that they are a widespread issue [15, 27]. A large number of tools were proposed to help developers find different kinds of synchronisation bugs, such as *atomicity violations* [6, 8, 16], *order violations* [9, 17, 32] and *data races* [20, 31]. These studies, however,

do not explore the security implications of discovered bugs—in most cases, the discovered bugs lead to memory corruption or crashes. Although such bugs may seem benign and unlikely to occur, synchronisation bugs are likely to lead to exploitable security vulnerabilities [26, 7, 23].

Unlike traditional applications, in the context of SGX, enclave code is trusted both by its developper and Intel to run untampered on untrusted machines (e.g., hosted at an untrusted cloud service provider). Memory corruption inside an enclave may therefore be used to hijack execution of the enclave, potentially leading to the disclosure of enclave cryptographic keys. In addition, such vulnerabilities may be used by malicious attackers, e.g., botnet herders, to bypass Intel's vetting process and design rootkits that run inside the enclave and are undetectable by security software running in the OS: by design, the OS cannot introspect an enclave running in production mode. Therefore, vulnerabilities in enclaves are worrisome to enclave developpers, enclave hosters, and Intel.

In the following, we show that synchronisation bugs are a real security threat to enclave developers by exploiting two examples of the common atomicity-violation bugs: a use-after-free bug as well as a TOCTTOU bug.

3 Exploiting Synchronisation Bugs with Scheduler Control

Exploiting synchronisation bugs inside an SGX enclave can be broken down into: (i) finding an exploitable synchronisation bug; (ii) providing a way to interrupt and schedule enclave threads; and (iii) determining experimentally *when* to interrupt and schedule enclave threads. Next we describe each of these steps through the example of a use-after-free bug. In addition, we describe the AsyncShock tool, which generalises this approach and allow the easy adaptation of these steps to other vulnerabilities. We explain how AsyncShock exploits a TOCTTOU bug.

3.1 Exploiting Synchronisation Bugs inside an Enclave

We focus on the atomicity-violation class of bugs and show how such a bug can be exploited. Figure 3 shows an example of an atomicity violation. A possible use-after-free bug occurs if the first thread is interrupted directly after the `free` but before the assignment. The second thread performs a `NULL` check during this time, which succeeds even though the pointer has been freed. The call to `free` and the assignment were intended to be an atomic block by developer, but this is not reflected in the implementation.

During execution, such an interruption is a scheduling decision by the OS, and the probability that the interruption occurs at the right point is low. Furthermore, the thread itself is not paused but is scheduled again later while the second thread is still executing. The second thread may thus be interrupted during its execution before the freed pointer can be used.

As shown in the litterature [29, 25], the attack window for memory races is small in practice. In some cases, the attacker may only have a single chance to exploit the vulnerability. Even if an attacker can execute the application many

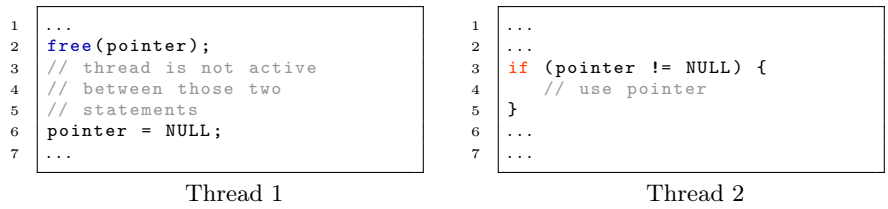

Fig. 3. Simple use-after-free. Thread one frees `pointer` but is not able to set it to `NULL` because thread two is scheduled in.

times, it may still take a long time until the interruption occurs at precisely the correct time. Being able to increase the attack window would thus help exploit such bugs more effectively. The AsyncShock tool aims to help exploit synchronisation bugs that are present inside an enclave by pausing and resuming threads during execution, which is possible when threads are inside an SGX enclave. We explore two techniques for interrupting threads, as described in the following sections.

3.2 Interrupting Threads via Linux Signals

One approach to interrupt threads is to leverage the Linux signal subsystem. Handling a signal interrupts the thread and redirects control to the signal handler. We therefore register a signal handler for the `SIGUSR1` and `SIGUSR2` signals. We use the `SIGUSR1` signal to pause a thread and the `SIGUSR2` signal to resume it again. A control thread sends these signals to specific threads based on a configurable delay. Elapsed time since the application start is measured and compared to the delay in a loop. When the delay is reached, a signal is issued. The signal is sent by the `pthread_kill` function provided by `pthreads`.

Pausing the thread is performed by using a condition variable to wait inside the signal handler that suspends the thread. Sending the resume signal causes a second signal handler invocation, which in turn uses a condition variable signal to resolve the wait in the first signal handler's invocation. Each thread has its own condition variable, facilitating the pausing and resuming of multiple threads.

While this approach works, it is unreliable and depends on the specifics of the Linux task scheduler. We experimented with different delays for the same exploit but observed the same success rate regardless of the delay. We suspect that the signal dispatching is too slow, leading to inaccurate interruptions. Furthermore, this approach requires a deterministic runtime of the program because the delay is fixed—non-deterministic execution inside the enclave defeats this approach.

3.3 Interrupting Threads via Forced Segmentation Faults

We explore another approach based on interrupting threads to force segmentation faults. Using `mprotect`, we remove the "read" and "execute" permissions

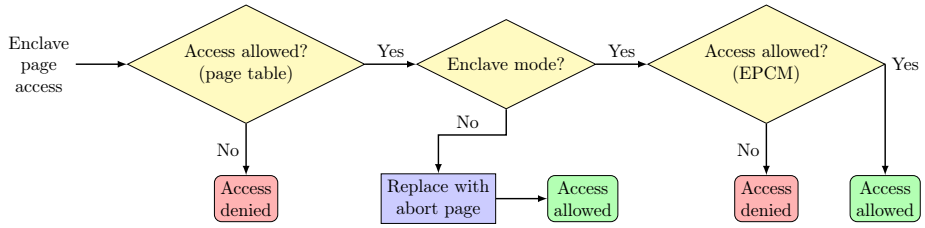

Fig. 4. Memory access permission checks on an enclave page. Permissions are checked and managed twice: once by the MMU (page table) and once by SGX (EPCM).

from enclave pages, i.e. *marking* the page. As soon as an enclave page with stripped permissions is accessed, a `SIGSEGV` signal is dispatched by the kernel as a response to the fault generated by the MMU, notifying the attacker of the page access.

This approach exploits the fact that memory access checks with SGX are performed twice, as shown in Figure 4. The call to `mprotect` changes the permissions inside the page table, but not inside the EPCM. Therefore the access fails at the page table check, even though the real permissions are unchanged.

We install our own signal handler, as described in §3.2, but this time for `SIGSEGV`. Inside the handler, we can restore the page permissions, start a timer with a configurable delay and resume execution. If a timer is started, it can remove the permissions upon expiration. This leads to another `SIGSEGV`, which again invokes our handler. We can now employ the same thread stopping mechanism described for the signal approach using condition variables. The `mprotect` approach is more reliable than the signal approach because page permissions are changed instantaneously.

3.4 AsyncShock Tool

AsyncShock incorporates the described approaches into an easy-to-use tool. It is implemented as a shared library, which is preloaded using the `LD_PRELOAD` mechanism of the dynamic linker. To interact with the target application, AsyncShock provides its own implementation of certain functions that shadow their real implementations. An example is `pthread_create`, which is normally provided by the C standard library. AsyncShock provides its own implementation that observes thread creation and takes actions upon the creation of specific threads.

To use AsyncShock, an attacker must know how the scheduling needs to be influenced to successfully trigger an exploit. They then must transform the attack into a series of actions in reaction to certain events. Possible events include thread creation, segmentation faults and timer expirations; possible actions include pausing or resuming a thread, starting a timer or changing page permissions. We call this series of actions the attack *playbook*. AsyncShock enforces that the targeted application behaves according to the playbook while also manipulating the environment.

```
on thread creation "thread1":
   remove read,exec on enclave+0x5000

on thread creation "thread2":
   pause thread this

on segfault 1:
   set read,exec on enclave+0x5000
   remove read,exec on enclave+0x1000

on segfault 2:
   set read,exec on enclave+0x1000
   resume thread thread2
   pause thread this
```

Listing 1.1. Example playbook for the use-after-free bug from Figure 3.

A textual representation of a playbook for the use-after-free bug from Figure 3 is given in Listing 1.1. It includes the definition of four reactions to events: on thread creation of the first thread, an enclave page (enclave base address + `0x5000`) is stripped of its read and execute permissions. By using `objdump`, we find that the `free` function is located on this code page, and we mark it to get notified when it is called. As soon as a thread calls the `free` function, a segmentation fault occurs, which is handled by the signal handler registered by AsyncShock. It reapplies the removed permissions and removes the permissions at another page. The marked page contains the calling function that we mark to get notified when `free` finishes.

The resulting segmentation fault is again handled by AsyncShock. This time, the faulting thread is paused, and the second thread is allowed to continue. As a result, the attack window has been widened for the second thread to exploit the bug.

3.5 AsyncShock in Action

We use AsyncShock to successfully exploit a use-after-free bug inside an enclave and take control of the instruction pointer. Listing 1.2 shows the exploited enclave code.

The code contains two `Ecalls`, one set-up `Ecall` only executed once and another `Ecall`. While the enclave contains no threads, the second `Ecall` is used by two untrusted threads to enter the enclave simultaneously. However, a synchronisation bug exists between lines 26 and 27 if multiple threads execute the `Ecall` function in line 19. `glob_str_ptr` is a shared variable between all executions that is freed inside the `Ecall` and set to `NULL`. The bug triggers if a thread has just executed the `free` but not yet the assignment, while a second thread enters the `Ecall` function. Due to the nature of the memory allocator provided by the SDK, the `malloc` call (line 20) provides the old `glob_str_ptr` address, which leads to `glob_str_ptr` and `my_func_ptr` pointing to the same memory. The second thread passes the `NULL` check and copies the user provided input

```
1  char *glob_str_ptr;
2
3  int other_functions(const char *c) { /* do other things */ }
4
5  int puts(const char * c) {
6     printf("%s", c);
7     return 0;
8  }
9
10 struct my_func_ptr {
11    int (*my_puts) (const char *);
12    char desc[8];
13 } my_func_ptr;
14
15 void ecall_setup() {
16    glob_str_ptr = malloc(sizeof(struct my_func_ptr));
17 }
18
19 void ecall_print_and_save_arg_once(void *str) {
20    struct my_func_ptr *mfp = malloc(sizeof(struct my_func_ptr));
21    mfp->my_puts = puts;
22    if (glob_str_ptr != NULL) {
23       memcpy(glob_str_ptr, (char *)str, sizeof(glob_str_ptr));
24       glob_str_ptr[sizeof(glob_str_ptr)] = '\0';
25       mfp->my_puts(glob_str_ptr);
26       free(glob_str_ptr);
27       glob_str_ptr = NULL;
28    }
29    free(mfp);
30 }
```

Listing 1.2. Example enclave containing a user-after-free bug.

to `glob_str_ptr`, which sets `my_func_ptr`. The function call in line 25 now receives its address from the user-provided input and can be given the address of another enclave function, thus hijacking the control flow inside the enclave.

We use AsyncShock with a playbook similar to the one shown in Listing 1.1 to exploit the bug. Figure 5 shows how AsyncShock exploits the bug in detail. AsyncShock lies dormant until one of its overwritten functions are called. The application first creates a thread that is paused immediately by AsyncShock ①. A second thread is created that is allowed to execute ②. At this point, the "read" and "execute" permissions are removed from the code page containing the `free` function. The second thread enters the enclave and begins execution. When it calls `free` ③, an access violation occurs, resulting in an AEX and a segmentation fault caught by AsyncShock ④. The permissions are restored for this page, but removed for another before the thread is allowed to continue.

When the next marked page is hit ⑤, resulting in another AEX and segmentation fault ⑥, we know that `free` has returned. In the signal handler, the permissions are restored again. We stop the thread and signal the sleeping thread to execute ⑦. This concludes the successful exploit.

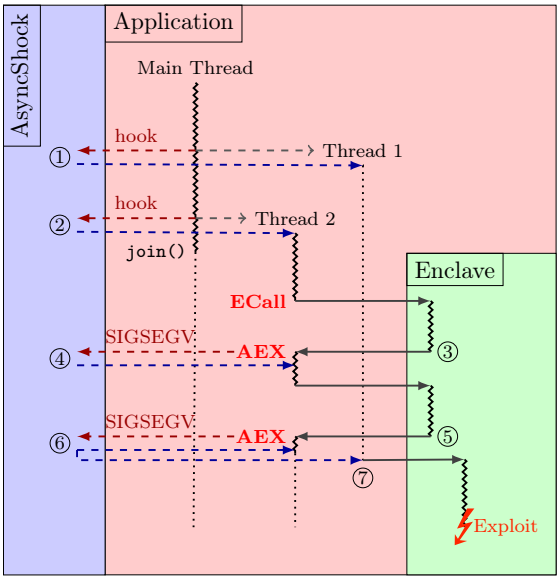

Fig. 5. AsyncShock exploiting the synchronisation bug from Listing 1.2

3.6 AsyncShock and a TOCTTOU Bug

To show how AsyncShock can be adapted to a different type of bug, we exploit a TOCTTOU bug. Listing 1.3 shows an enclave with three `Ecalls`: two threads enter the enclave, the first through the `ecall_writer_thread` and the second through the `ecall_checker_thread` `Ecall`. The second thread checks (line 20) if the shared variable `data` contains the string `"bad data"` and, if so, does not access it. Other content leads to a successful check and results in the second use of the variable. The first thread writes to the shared variable after executing a non-deterministic amount of time.

A TOCTTOU bug exists in lines 18 (check) and 19 (use). AsyncShock exploits this bug by delaying the writer thread, interrupting enclave execution after the check and then letting the writer thread proceed with the write to the shared variable. Interrupting between the check and the use in this example is challenging because the code pages containing `strncmp` and `memcpy` also contain some frequently called methods of the SDK. We therefore opt to start a timer right before entering the enclave, which expires between the check and the use. The timer has a configureable delay that postpones its execution. The correct delay must be determined empirically by observing the behaviour of the application with different delays. In our example, we observe the most successful exploits by choosing delays between 80000 and 120000 cycles, as described in §4.3.

```
static char data[] = {'g', 'o', 'o', 'd', ' ', 'd', 'a', 't', 'a', '\0'};
static int random_wait = 0;

void ecall_setup() {
   random_wait = get_random_int();
}

void ecall_writer_thread() {
//This function has a constant delay >> check + use plus a random delay
//to simulate complex execution that takes a non-deterministic amount of time
   for (int i = 0; i < 100000; ++i);
   for (int j = 0; j < random_wait; ++j);
   snprintf(data, 10, "bad data");
}

int ecall_checker_thread() {
   char *str = calloc(1, 10);
   if (strncmp("bad data", data, 9) != 0) {
      memcpy(str, data, 10);
      printf("Access ok: %s\n", str);
      free(str);
      return 0;
   } else {
      printf("Sorry, no access!\n");
      return -1;
   }
}
```

Listing 1.3. Example enclave containing a TOCTTOU bug.

4 Evaluation

To show the effectiveness of AsyncShock, we evaluate it by exploiting two atomicity violation bugs. First, we describe our evaluation set-up. After that, we present the results of exploiting a use-after-free bug and a TOCTTOU bug inside an enclave. We finish with a discussion of possible defenses.

4.1 Experimental Set-up

We evaluate the effectiveness of AsyncShock by exploiting a use-after-free bug, as well as a TOCTTOU bug, on real SGX hardware. We used a Dell Optiplex 7040 with an i7-6700 Intel CPU and 24 GB of memory. We also evaluate AsyncShock on a white-box server with an Intel E3-1230v5 CPU and 32 GB of memory. Both CPUs have four cores and are capable of hyper-threading, doubling the possible active threads. For our evaluation, hyper-threading has not been disabled. The desktop machine runs Ubuntu Linux 14.04.3 Desktop with kernel version 3.19.0-49; the server machine runs Ubuntu Linux 14.04.4 Server with kernel version 3.13.0-85. The server machine has a lower base load because fewer processes exist due to the missing desktop environment. All evaluations use a pre-release version of the SGX SDK which Intel provided to us.

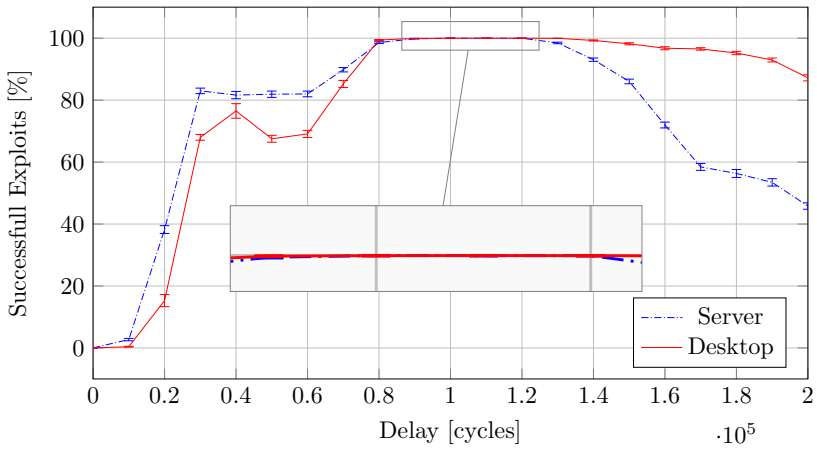

Fig. 6. Graph showing the success rate of AsyncShock exploiting the TOCTTOU bug at different timer delays.

4.2 Exploiting a Use-After-Free Bug

First, we establish a baseline by running the application without AsyncShock. We execute the application with its enclave one million times without observing a single successful exploit. We conclude that the attack window is too small to be exploitable just through controlled input.

We exploit the bug shown in Listing 1.2. Given the playbook from Listing 1.1, we can reliably exploit the use-after-free bug. We also modify the playbook to change the function arguments for the second thread so that the use-after-free results in a control flow modification, i.e. a call to `other_function`, which is otherwise not called. We execute the exploit 100000 times on both machines and observe a 100% success rate.

4.3 Exploiting a TOCTOU Bug

To put the high rate chance of exploiting the use-after-free bug into perspective, we also consider a more difficult bug to exploit reliably: a TOCTTOU bug inside an enclave. Here we exploit the enclave code shown in Listing 1.3. We also establish a baseline by executing the application without AsyncShock. As with the use-after-free bug, we also do not see a single exploit occurring by chance. The non-deterministic delay in the writer thread is long enough so that the other thread can always perform the check and the use on the same data.

Next, we try to exploit the bug with AsyncShock. We evaluate a wide range of delays for the timer, as described in §3.6. Each delay is executed 10000 times. We record the successful exploits every 100 executions, obtaining 100 result sets per delay. We report the mean success rate for a given delay, with error bars representing a 95% confidence interval.

Delay (cycles)	Success rate	
	Server	Desktop
80,000	98.57% ± 0.27	99.40% ± 0.17
90,000	99.81% ± 0.08	99.93% ± 0.05
100,000	99.99% ± 0.02	99.99% ± 0.03
110,000	99.98% ± 0.03	99.99% ± 0.02
120,000	99.98% ± 0.03	99.98% ± 0.03

Table 1. Deatiled success rates for selected delays.

Figure 6 shows the results for the TOCTTOU exploit. As can be seen, the success rate varies not only with timer delay, but also differs for both machines with the same delay. We attribute this behaviour to the differences in base load and active processes on both machines. We are able to achieve near 100% success rates with timer delays of 80,000 cycles to 120,000 cycles. (As explained in §3.3, the delay is the time until AsyncShock removes the "execute" permissions from an enclave page, effectively forcing a stop to execution.) Our goal is to stop the enclave between the check and the use, which we achieve almost always with the correct delay.

Table 4.3 shows the results in more detail for selected delays. With a delay of 100,000, AsyncShock can almost always exploit the TOCTTOU bug with a low deviation. In conclusion, AsyncShock can be used to reliably exploit atomicity violation bugs with a high success rate.

4.4 Protective Measures Against AsyncShock

Our experimental results show that synchronisation bugs can lead to viable attacks against SGX enclaves. However, there already exist defense mechanisms for protecting from these attacks.

A first defence against the use-after-free bug is the sanitisation of user input as AsyncShock changes the `Ecall` parameters to direct the control flow. In general, sanitisation is advisable when unexpected input can be abused in a similar way to Iago attacks [5]. Enclave code should always check outside input for validity as an attacker may change the result from `Ocalls` or the parameters to `Ecalls` when using the SDK. In addition, enclave developers should not rely on the SDK's ability to defend against simple TOCTTOU attacks. While the SDK does copy `Ecall` parameters into enclave memory before passing them to enclave functions, it does not deep-copy data structures. Pointers in data structures are not followed and may lead to an enclave accessed outside memory. This type of vulnerability has often been exploited in OS kernels (e.g., [14] for Windows, and in general in filesystems [25]).

Another defense against the use-after-free bug presented here is possible because the bug relies on the in-enclave implementation of `malloc` to return recently freed memory. The attack can be mitigated by heap hardening methods, such as the one recently implemented in Internet Explorer through delayed

free [10], or even with tools such as AddressSanitizer [22] that delay the reuse of recently freed memory or by changing the behaviour of the in-enclave memory allocator.

Protection from all synchronisation bugs can be achieved by prohibiting threading altogether—if only a single thread can enter the enclave at any time, no inconsistencies are possible due to serial execution. Such a solution, however, negatively impacts performance. If parallelism is needed, one can also adapt other techniques to work inside enclaves such as Stable Multithreading [30] or use tools such as ThreadSanitizer [23] during development in order to find and eliminate synchronisation bugs.

While many hardening techniques are applicable to enclave code, some traditional techniques do not work in the context of SGX. For example, the use of address space layout randomisation (ASLR) [19] is not directly applicable inside enclaves because any changes of the enclave memory would change the enclave measurement and therefore fail the signature check.

5 Related Work

Because SGX is a new technology with limited production use, only few use cases have been described so far. Haven [2] executes unmodified Windows applications inside an enclave. To achieve this, the combination of a shield module and a library OS provides the necessary execution support. The shield module manages synchronisation primitives and ensures their correct behaviour, similar to the SGX SDK. Furthermore, Haven tries to defend against Iago attacks be sanitising and checking the parameters of `Ecalls` and results of `Ocalls`. Haven also proposes a decoupling of enclave threads and host threads via user-level scheduling to hinder the exploitation of synchronisation bugs. However, AsyncShock should still be effective as it marks pages in close proximity to the synchronisation bugs to force an AEX. Thus, it does not necessarily need to observe the enclave-internal thread scheduling.

Fine-grained page tracking can be used for powerful side channel attacks [28]. For example, a JPEG image generated inside an enclave could be reconstructed outside: by paging out enclave pages to repeatedly induce page faults, memory accesses could be related to certain code paths. In contrast, AsyncShock is geared towards the exploitation of synchronisation bugs, albeit it can also be used to extract information from an enclave. However, for synchronisation bugs, AsyncShock only needs a small number of marker pages to track the enclave execution close to the critical section.

Yang et al. [29] identify concurrency attacks as a risk to real-world systems. They classify different types of attacks based on memory access patterns, and identify the attack window as an important factor for exploitability. Memory races usually have a small attack window at the level of nanoseconds. AsyncShock widens the attack window by stopping threads when a critical state is reached, steering other activities to allow reliable exploitation of memory-based concurrency bugs.

Synchronisation bugs have also been studied for their security implications. For instance, TOCTTOU races often affect filesystem-related code, typically when performing access control decisions. Dean and Hu [7] propose a countermeasure to alleviate those risks. Borisov et al. [4] show that this probabilistic countermeasure can be reliably defeated with filesystem mazes. Tsafrir et al. [25] propose another way to instrument those access checks to make the exploitation of those races significantly more difficult even against filesystem mazes.

Twiz and Sgrakkyu [26] extensively treat techniques for the exploitation of logical bugs in OS kernels. Jurczyk and Coldwind [14] describe how to exploit race conditions via memory access patterns in the Windows kernel. The Windows kernel copies the arguments to system calls from user to kernel space. However, the kernel does not copy pointer-referenced data in some cases. The authors exploit this by using the Bochs CPU emulator to interrupt the kernel, similar to how AsyncShock swaps out the data between two reads by the kernel—a classical TOCTTOU attack. However, in contrast, AsyncShock attacks an SGX enclave and not the kernel, in a setting where the attacker controls the scheduler and has reliable side channels on a thread's progress.

Moat [24] makes a first step towards the verification of SGX enclaves. The authors propose an approach to verify that enclave code is unable to disclose secrets. They employ static analysis on the x86 machine code, introducing "ghost variables" to track the secrecy of data in a manner similar to taint tracking. With this method, they are able to find occurrences of possible sensitive data disclosure. While their approach is promising for detecting data disclosure, they, unlike AsyncShock, do not consider multi-threaded code in enclaves.

6 Conclusion

This paper analyses the impact of synchronisation bugs inside SGX enclaves. We have shown that the impact of synchronisation bugs is greater within SGX enclaves than in traditional applications, because their exploitation becomes highly reliable through attacker-controlled scheduling. We described AsyncShock, a tool for thread manipulation, and showed how it can be used to exploit synchronisation bugs by widening the attack window through controlled thread pausing and resuming. AsyncShock operates as a preloaded library without modifications of the target application or host OS. We demonstrated that synchronisation bugs can be exploited inside SGX enclaves using AsyncShock for control flow hijacking or bypassing access checks.

Acknowledgements

We would like to thank the anonymous reviewers for their input. This project has received funding from the European Union's Horizon 2020 Research and Innovation Programme under Grant Agreement No. 645011 and No. 644412.

References

1. *ARM TrustZone.* http://www.arm.com/products/processors/technologies/trustzone/index.php
2. Baumann, A., Peinado, M., Hunt, G.: *Shielding Applications from an Untrusted Cloud with Haven.* In: Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation, OSDI'14, pp. 267–283 (2014)
3. Bishop, M., Dilger, M.: *Checking for Race Conditions in File Accesses.* Computing systems 2(2), 131–152 (1996)
4. Borisov, N., Johnson, R., Sastry, N., Wagner, D.: *Fixing Races for Fun and Profit: How to Abuse Atime.* In: Proceedings of the 14th Conference on USENIX Security Symposium - Volume 14, SSYM'05, pp. 20–20 (2005)
5. Checkoway, S., Shacham, H.: *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, pp. 253–264 (2013)
6. Chew, L., Lie, D.: *Kivati: Fast Detection and Prevention of Atomicity Violations.* In: Proceedings of the 5th European Conference on Computer Systems, EuroSys '10, pp. 307–320 (2010)
7. Dean, D., Hu, A.J.: *Fixing Races for Fun and Profit: How to Use Access(2).* In: Proceedings of the 13th Conference on USENIX Security Symposium - Volume 13, SSYM'04, pp. 14–14 (2004)
8. Flanagan, C., Freund, S.N.: *Atomizer: A Dynamic Atomicity Checker for Multithreaded Programs.* SIGPLAN Not. 39(1), 256–267 (2004)
9. Gao, Q., Zhang, W., Chen, Z., Zheng, M., Qin, F.: *2ndStrike: Toward Manifesting Hidden Concurrency Typestate Bugs.* In: Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS XVI, pp. 239–250 (2011)
10. Hariri, Zuckerbraun, Gorenc, *Abusing Silent Mitigations.* BlackHat USA'15
11. Hoekstra, M., Lal, R., Pappachan, P., Phegade, V., Del Cuvillo, J.: *Using Innovative Instructions to Create Trustworthy Software Solutions.* In: Proceedings of the 2Nd International Workshop on Hardware and Architectural Support for Security and Privacy, HASP '13, 11:1–11:1 (2013)
12. Intel, *Intel® Software Guard Extensions SDK for Linux* OS , Revision 1.5.* https://01.org/intel-software-guard-extensions/documentation/intel-sgx-sdk-developer-reference
13. Intel, *Intel(R) Software Guard Extensions Programming Reference, Revision 2.* https://software.intel.com/sites/default/files/managed/48/88/329298-002.pdf
14. Jurczyk, M., Coldwind, G.: *Identifying and Exploiting Windows Kernel Race Conditions via Memory Access Patterns.* In: Bochspwn: Exploiting Kernel Race Conditions Found via Memory Access Patterns, p. 69 (2013)
15. Lu, S., Park, S., Seo, E., Zhou, Y.: *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, pp. 329–339 (2008)
16. Lu, S., Tucek, J., Qin, F., Zhou, Y.: *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, pp. 37–48 (2006)

17. Lucia, B., Ceze, L.: *Finding Concurrency Bugs with Context-aware Communication Graphs.* In: Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 42, pp. 553–563 (2009)
18. McKeen, F., Alexandrovich, I., Berenzon, A., Rozas, C.V., Shafi, H., Shanbhogue, V., Savagaonkar, U.R.: *Innovative Instructions and Software Model for Isolated Execution.* In: Proceedings of the 2Nd International Workshop on Hardware and Architectural Support for Security and Privacy, HASP '13, 10:1–10:1 (2013)
19. PaX, *PaX address space layout randomization (ASLR).* (2003)
20. Savage, S., Burrows, M., Nelson, G., Sobalvarro, P., Anderson, T.: *Eraser: A Dynamic Data Race Detector for Multithreaded Programs.* ACM Trans. Comput. Syst. 15(4), 391–411 (1997)
21. Schuster, F., Costa, M., Fournet, C., Gkantsidis, C., Peinado, M., Mainar-Ruiz, G., Russinovich, M.: *VC3: Trustworthy Data Analytics in the Cloud Using SGX.* In: Security and Privacy (SP), 2015 IEEE Symposium on, pp. 38–54 (2015)
22. Serebryany, K., Bruening, D., Potapenko, A., Vyukov, D.: *AddressSanitizer: A fast address sanity checker.* In: Proceedings of the 2012 USENIX Annual Technical Conference (USENIX ATC 12), pp. 309–318 (2012)
23. Serebryany, K., Iskhodzhanov, T.: *ThreadSanitizer: data race detection in practice.* In: Proceedings of the Workshop on Binary Instrumentation and Applications, pp. 62–71 (2009)
24. Sinha, R., Rajamani, S., Seshia, S., Vaswani, K.: *Moat: Verifying Confidentiality of Enclave Programs.* In: Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security, CCS '15, pp. 1169–1184 (2015)
25. Tsafrir, D., Hertz, T., Wagner, D., Da Silva, D.: *Portably Solving File TOCTTOU Races with Hardness Amplification.* In: FAST'08, pp. 1–18 (2008)
26. Twiz, Sgrakkyu, *Attacking the Core: Kernel Exploitation Notes.* Phrack 64 file 6
27. Xiong, W., Park, S., Zhang, J., Zhou, Y., Ma, Z.: *Ad Hoc Synchronization Considered Harmful.* In: OSDI, pp. 163–176 (2010)
28. Xu, Y., Cui, W., Peinado, M.: *Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems.* In: Security and Privacy (SP), 2015 IEEE Symposium on, pp. 640–656 (2015)
29. Yang, J., Cui, A., Stolfo, S., Sethumadhavan, S.: *Concurrency Attacks.* In: Presented as part of the 4th USENIX Workshop on Hot Topics in Parallelism, (2012)
30. Yang, J., Cui, H., Wu, J., Tang, Y., Hu, G.: *Making Parallel Programs Reliable with Stable Multithreading.* Commun. ACM 57(3), 58–69 (2014)
31. Yu, Y., Rodeheffer, T., Chen, W.: *RaceTrack: Efficient Detection of Data Race Conditions via Adaptive Tracking.* In: Proceedings of the Twentieth ACM Symposium on Operating Systems Principles, SOSP '05, pp. 221–234 (2005)
32. Zhang, W., Sun, C., Lu, S.: *ConMem: Detecting Severe Concurrency Bugs Through an Effect-oriented Approach.* In: Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems, ASPLOS XV, pp. 179–192 (2010)