Detect Iago Vulnerabilities in Legacy Code with Reverse Syscall Fuzzing

by

Rongzhen(Gavin) Cui

A thesis submitted in conformity with the requirements
for the degree of Master of Applied Science
Graduate Department of Computer Engineering
University of Toronto

Abstract

Detect Iago Vulnerabilities in Legacy Code with Reverse Syscall Fuzzing

Rongzhen(Gavin) Cui
Master of Applied Science
Graduate Department of Computer Engineering
University of Toronto
2020

There has been interest in mechanisms that allow legacy code to be securely used to implement trusted code that runs in Trusted Execution Environments (TEEs), such as Intel SGX. However, because legacy code generally assumes the presence of an operating system, this naturally raises the spectre of Iago attacks on the legacy code. We develop Emilia, which automatically detects Iago vulnerabilities in legacy applications using reverse system call fuzzing. We use Emilia to discover 50 Iago vulnerabilities in 17 applications, and find that Iago vulnerabilities are widespread and common. We conduct an in-depth analysis of the vulnerabilities we found and conclude that while common, the majority (84%) can be mitigated with simple, stateless checks in the system call forwarding layer, while the rest are best fixed by finding and patching them in the legacy code. Finally, we study and evaluate different trade-offs in the design of Emilia.

Acknowledgements

First, I would like to thank my supervisor, Professor David Lie, whose expertise was invaluable in formulating the research questions and methodology. Your insightful feedback pushed me to sharpen my thinking and brought my work to a higher level.

I would also like to thank Lianying(Viao) Zhao at Carleton University for exchanging ideas and helping paperwork.

Finally, I would like to thank University of Toronto, the Department of Electrical and Computer Engineering and Ontario government for their financial support.

Contents

1 Introduction **1**

2 Background **4**

2.1 Isolation techniques . . . 4

2.1.1 Handling system calls . . . 5

2.1.2 Protect trusted files . . . 5

2.2 Iago Attacks . . . 6

2.2.1 SSL Replay with `getpid` . . . 6

2.2.2 Compromising any program using `malloc` . . . 6

3 Related Work **8**

3.1 Mitigation of Iago vulnerabilities . . . 8

3.2 Iago Attack Analysis . . . 9

4 Design of Emilia **10**

4.1 Threat Model and Assumptions . . . 10

4.2 Reverse Syscall Fuzzing . . . 11

4.3 Objective . . . 13

4.4 Measuring syscall coverage . . . 13

4.5 Achieving syscall coverage . . . 14

4.6 Fuzzing Strategies . . . 15

4.6.1 Target Selection . . . 15

4.6.2 Fuzzing Value Sets . . . 18

4.6.3 Return Fields . . . 18

5 Implementation of Emilia **20**

5.1 Overview . . . 20

5.2 Interceptor . . . 21

5.2.1 Stack hash extraction . . . 21

5.2.2 Return value mutation . . . 22

5.3 Controller . . . 23

5.4 Value extractor . . . 24

6 Vulnerability Analysis **25**
6.1 Applications examined 25
6.2 Classification 25
6.3 Vulnerabilities found 27
6.4 Mitigating Iago vulnerabilities 31
6.5 Forwarding OFL Analysis 32

7 Evaluation **34**
7.1 Comparing Target Selection Methods 34
7.2 Effects of Valid Values 36

8 Conclusion and Future Work **37**
8.1 Future Work 37

Bibliography **39**

List of Tables

5.1 Stack trace overhead (ms) 22

6.1 Legacy applications analyzed 26
6.2 Detected Iago vulnerabilities 27

7.1 Stateful fuzz vs. Fuzz-all 35
7.2 Effects of the valid value set 35

List of Figures

2.1 Syscall forwarding model 5

4.1 High-level Overview of Reverse Syscall Fuzzing Emilia 12

5.1 Components and workflow of Emilia 20

List of Algorithms

1 Stateful fuzzing loop 17

Listings

1.1 Iago attack example . . . 2
4.1 An example of a new syscall invocation introduced by fuzzing . . . 12
4.2 An example of varying syscall arguments . . . 14
4.3 An example of early termination with fuzz-all method . . . 16
4.4 An example of missing vulnerability with fuzz-all method . . . 16
4.5 An example of extracted valid values . . . 18
6.1 An example (accept) of the Static semantics in Redis . . . 28
6.2 An example (readlink) of the local semantics in OpenSSH . . . 28
6.3 An example (epoll_wait) of the stateful semantics in Lighttpd . . . 29
6.4 An example (fstat) of the external semantics in glibc . . . 30

Chapter 1

Introduction

Operating system(OS) is the most important software component in common computing infrastructure. Its core code (kernel code) runs in a privileged supervisor mode (also known as kernel mode) to manage computer resources and provide services for applications. Due to its high privilege and overly permissive interface, a compromised OS or privileged application can read and modify the execution state (e.g. memory content and register values) of any program running in the same system. For example, a compromised OS could extract the encryption key from a web server's memory and steal sensitive information that is supposed to be crypto-protected such as financial and health data. It is challenging to make all privileged processes secure. Therefore, running secure-sensitive applications in a Trusted Execution Environment (TEE) and protecting them from vulnerable components is an attractive alternative. In recent years, many hardware-based [17, 11, 1] and hypervisor-based [6, 5, 35] TEEs have been proposed. Applications implemented in those environments are isolated from the large, legacy trusted computing base (TCB, set of components critical to the computer system's security) of commodity systems, which includes the OS, drivers, and all privileged applications on a system. Thus, the computing infrastructure is separated into the normal/untrusted world controlled by the untrusted OS and the secure/trusted world in which the protected code is executing. Privileged parties in the untrusted world are no longer allowed to access sensitive information stored in the protected application's memory.

However, to retain the benefits of the isolation that TEEs provide, such "trusted" applications themselves should remain small so that they can have a higher level of assurance. As a result, there exists a tension between rewriting code from scratch to keep it small, and re-using legacy application code, which reduces effort in implementing trusted applications. Both new code and legacy code may have vulnerabilities and need to be audited. In addition, using legacy code creates an extra challenge—it assumes the presence of an operating system. Normally, the OS provides services (e.g. accessing hardware) to applications through the system call (abbreviated to syscall hereinafter) interfaces. Thus, supporting syscalls is necessary for running legacy code. One way to achieve that is to implement some OS's functionalities and handle syscalls internally in the trusted environment. However, handling syscalls involves communication with hardware components. Some hardware-based TEEs such as Software Guard eXtension (SGX) do not have this ability. Code inside SGX always executes in the lowest privilege level (ring 3 / user mode), which is restricted from performing I/O. Besides, communicating with file systems and other processes in the normal world still requires interaction with the untrusted OS. Therefore, another option is to delegate those service requests back to the untrusted OS by forwarding syscalls through an OS-forwarding layer (OFL). When the protected application makes a syscall, the OFL will forward the requests with its arguments to the untrusted OS, and return the results to the application after receiving them.

One issue raised by the use of such OFL is that legacy code often inherently trusts the OS that it makes syscalls to. Once legacy code is moved into a TEE, such trustworthiness can no longer be assumed. When an untrusted OS that compromises an application in a TEE due to this misplaced trust, this attack is known as an Iago attack. For example, a legacy application might assume the returned size from the `read` syscall would not exceed the max length specified in the syscall's argument. In listing 1.1, the application assumes `len <= MAX_LEN-1` after `read` syscall returns, it then uses `len` to send buffer content to the network with `write` syscall. A malicious `len` that violates the assumption will trick the application to write content beyond the `buf`. As a result, the value of `secret` will be exposed to the attacker.

```c
char buf[MAX_LEN];
int secret = 1234;
size_t len = read(fd, buf, MAX_LEN-1);
size_t ret = write(socket_fd, buf, len);
```

Listing 1.1: Iago attack example

Such attacks were first identified in [5] and [30], and then eventually named Iago attacks in [4]. While most legacy code implicitly trusts the OS, this does not automatically mean that all legacy code is vulnerable to Iago attacks. For code to be vulnerable, it must a) neglect to sanitize the return values of a syscall and b) use the return values in an unsafe way. Thus, for legacy code to be vulnerable to an Iago attack, it must have an Iago vulnerability that meets these two criteria.

Iago attacks are real-world threats because many projects still propose using OFLs to enable legacy code to run in a TEE. Such projects include TaLoS [12] and SGX_SQLite [29]. They port legacy LibreSSL and SQLite correspondingly into SGX enclaves. Syscalls are forwarded to the untrusted OS without verifying the return values. Therefore, Iago vulnerabilities in the legacy codes may be exploited when those projects are deployed.

We present Emilia[1], a reverse syscall fuzzer that is designed to find and detect Iago vulnerabilities in legacy code. Emilia fuzzes applications from the syscall interface by replacing legitimate OS's syscall return values with fuzz values designed to find and trigger Iago vulnerabilities. We run Emilia on 17 popular applications and find a total of 50 Iago vulnerabilities, which were categorized into 4 basic types. We also detect two Iago vulnerabilities in Google's Asylo [10] system, an OFL that has been specially designed to protect applications against Iago vulnerabilities. Both vulnerabilities have been confirmed and fixed by the Asylo team. Our main result is that Iago vulnerabilities are wide-spread—almost every application we examined had at least one vulnerability. Fortunately, many vulnerabilities are stateless and could be easily mitigated by minor modifications to an OFL.

In summary, we make the following **contributions**:

- We present Emilia, a tool that detects Iago vulnerabilities using reverse syscall fuzzing.
- We use Emilia to measure the frequency of the Iago vulnerabilities in real-world applications, and have identified a total of 50 memory corruption vulnerabilities in 17 popular legacy applications and

[1]Emilia was the wife of Iago who eventually reveals Iago's treachery in Shakespeare's tragedy, Othello.

glibc. From the statistics, we can see the actual impact of the Iago vulnerabilities. Furthermore, we have also found similar vulnerabilities in one of the OFL implementations, Google Asylo [10], which is supposed to treat the OS as untrusted.

- We identify some of the underlying causes of the Iago vulnerabilities by characterizing the syscall return values. Our analytics sheds some light on how legacy applications can be better ported to the OFL's protection.

Chapter 2

Background

2.1 Isolation techniques

Recently, the security provided by commodity operating systems is often insufficient because privileged components include not only kernel but also device drivers and services daemons that run as root in the system. Those components expose broad attack surfaces that are frequently vulnerable to bugs or misconfigurations. Once they have been compromised, the attacker could gain access to any sensitive data on the system. To solve this problem, researchers have built isolation techniques to protect the execution of secure-sensitive applications even when the underlying OS is compromised.

In general, there are hypervisor-based and hardware-based solutions. A hypervisor creates virtual machines for operating systems to run. The OS running in the virtual machine is called guest OS. The hypervisor provides a mapping from the guest OS's memory to the actual physical memory. That means a hypervisor could restrict or intercept the guest OS's memory access. Hypervisor-based solutions use this ability to create an environment that is not accessible from an untrusted guest OS. ProxOS [35] runs the protected application in another virtual machine paralleled to the untrusted OS. A unikernel is combined with the application to support basic functionalities and keep the TCB small. Overshadow [5] provides multiple views of the guest's memory. It presents the application with cleartext of its page, and the OS with an encrypted view.

Hardware-based solutions utilize designated hardware. One example is Intel's Software Guard eXtentions (SGX) [17], which provides processor support to run secure-sensitive code on a computer owned and maintained by an untrusted party. In SGX, the protected code is running in a secure container called *enclave*. While loading the enclave, the code's initial integrity and its private data are checked through software attestation. A user could refuse to interact with the service running in an enclave, of which content hash does not match the expected value. The integrity and confidentiality of the execution inside an enclave are protected from the outside environment, including the OS, the hypervisor, and hardware devices attached to the system bus. It is achieved by storing the enclave's code and data into a Processor Reserved Memory (PRM), which cannot be directly accessed by other software or peripherals. This memory protection is provided by a Memory Encryption Engine(MEE) connect to the memory controller.

2.1.1 Handling system calls

As mentioned in Chapter 1, handling syscalls invoked by the legacy application is an essential task for those isolation techniques. They usually have an OFL to proxy requested services. For Overshadow, the OFL consists of both trusted and untrusted shims. The VMM is responsible for saving the execution state and transferring control among shims and kernel by setting instruction and stack pointers. Both shims act as trampolines to perform hypercall to the VMM for context switch. Syscall arguments and outputs are stored in the memory region of untrusted shim. Both untrusted kernel and trusted shim can access this region. The trusted shim is responsible for writing and retrieving those data. For ProxOS, trusted syscalls can be handled internally by the unikernal in the private Virtual Machine(VM). Untrusted ones are forwarded to the commodity OS through inter-VM remote procedure calls (RPCs) and a shared communication buffer.

In SGX enclaves, `INT` and `SYSCALL` instructions are disallowed for security. However, the enclave can use ECALL and OCALL, an RPC-like mechanism, to enter an enclave function or temporarily leave the enclave and invoke an untrusted function in the normal world. Many SGX-based isolation techniques utilize this mechanism to redirect syscalls.

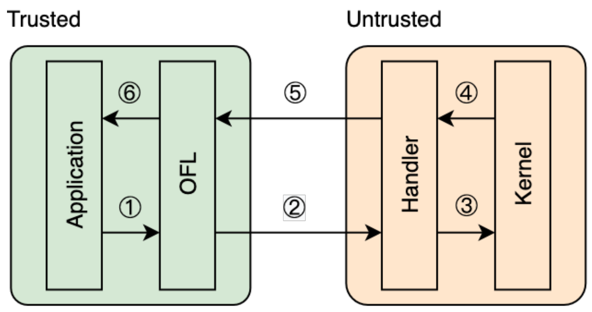

Figure 2.1: Syscall forwarding model

In general, syscall forwarding techniques share a similar model as shown in Figure 2.1. The OFL sits inside the trusted environment intercept application's syscall request in step 1. Then, it passes the syscall arguments to an untrusted handler in step 2. In steps 3 and 4, the handler invokes the actual syscall to the commodity OS. All the syscall outputs are returned to the handler's address space. The OFL copies those outputs into the trusted environment in step 5. Finally, values are returned to the application in step 6.

2.1.2 Protect trusted files

Applications sometimes rely on the OS to provide access control for protecting trusted files' confidentiality and integrity. When the trust shift, it is isolation techniques' responsibility to ensure the application can retrieve the correct file content from `read`-related syscalls. For example, ProxOS [35] encrypt and decrypt trusted data write to and read from the untrusted OS. It also stores hashes of all trusted files into a private block device directly available from the underlying VMM to verify the file's integrity. Graphene-SGX [37] also stores secure hashes of trusted files into each application's manifest. The manifest is measured as part of enclave initialization. InkTag [14] translates file I/O in trusted files into operations memory-mapped files. Then it ensures privacy and integrity by hashing and encrypting in-memory file data in response to accesses by the untrusted OS. All of those protection mechanisms happen transparently without the requirement of code modification. Users only need to specify which files are trusted through the configuration stage.

2.2 Iago Attacks

Porting a legacy application to an isolated environment means the ported application a) still needs the OS's support to access services through syscalls and b) still retains any trust it has in that OS. The application could trust syscall return values to follow the semantics described in the syscall's specification. Iago attack [4] is an attack that A malicious kernel can mount by manipulating syscall return values and forcing a protected process to act against its interests. Its threat model assumes that the malicious kernel can not directly read or modify the protected application's execution state. The kernel still handles syscalls invoked by the application and can provide arbitrary syscall return values. It assumes both the application and linked system libraries are unmodified and written to trust the OS. The original Iago attack paper demonstrates two Iago exploits. One causes replay attacks on Apache servers with `mod_ssl`, due to the syscall `getpid` being used in part for randomness. The other one even achieves arbitrary code execution because `malloc` (wrapped in c library) could be tricked to modify arbitrary memory by malicious return values of `brk` and `mmap`.

2.2.1 SSL Replay with `getpid`

The SSL replay attack is presented as a warmup example in Iago paper. SSL protocol uses random numbers to derive cryptographic secrets for each session so that the same message crossing different SSL sessions will be encrypted differently. If the random value is reused, the attacker can replay previous packets to repeat the user's one-time action even if the attacker cannot forge arbitrary requests without knowing the secret. For example, a single money transfer could be turned into multiple transactions with the same amount. In the `mod_ssl` extension of the Apache Web server, the entropy pool used to generate random values is initialized and seeded with strong entropy once in the parent process. Every child process inherits an identical entropy pool when forked. To prevent generating the same randomness in each child process, Apache stirs the entropy pool with system time in seconds and process ID obtained from `getpid`, which should be distinct for each child process. However, once the kernel is malicious, it can provide the same process ID and time for all child processes through the return value of `getpid` and `time`. As a result, packets sent to one child process can be replayed by establishing a new connection to another child process because the random value generated by two processes will be the same.

2.2.2 Compromising any program using `malloc`

Consider the following code fragment.

```
p = mmap(NULL, 1024, prot, flags, -1, 0);
read(fd, p, 1024);
```

`mmap` syscall maps a 1024 byte memory region whose start address is stored at pointer `p`. After that, the code reads up to 1024 bytes into the memory region from a file descriptor through the `read` syscall.

Unlike a benign kernel which returns the address of a newly allocated memory region from `mmap`, a malicious kernel could break this semantic and return an arbitrary address. The following `read` will overwrite content on that address with bytes provided by the attacker. If the attacker could guess the memory layout of the application, he could trick the application to overwrite the return address saved on the stack and hijack the execution once the function returns.

This `mmap-read` logic is common in standard I/O functions such as `fread` that reads data from a file. Those functions perform buffering on the target file. The fixed sized buffer to hold one block data is allocated by `mmap` in the `EGLIBC` internal function `_IO_file_doallocate`. Then it will be filled by the `read` syscall in `_IO_new_file_underflow`

The Iago paper further introduces another vulnerability in `malloc` function of `EGLIBC`. `malloc` uses `brk` syscall to allocate memory. `brk(addr)` changes the location of the program break that defines the end of the process's data segment. Increasing the program break has the effect of allocating memory to the process. Every allocated memory region managed by malloc is called a memory chunk. `malloc` writes both the previous chunk's size and current chunk's size at the beginning of each chunk. There's also a special top-most chunk that can grow and shrink as `malloc` requests memory from and returns memory to the system. If it is the first time the process calls `malloc(size)`, `malloc` will invoke several `brk` syscall. In general, it uses the returned address from the first `brk(0)` to get the initial program break `S`, and calculate the start of the initial top chunk based on `S` and requested `size`. Then `malloc` invoked another `brk` to align the requested memory and get the end of new program break `E`. The size of the top chunk (`E-S-size`) is then written to the corresponding metadata region (`S+size+4`). By carefully crafting `S` and `E` returned from `brk` syscalls, a malicious kernel can make `malloc` write a single word of the kernel's choice into an arbitrary address. Given this vulnerability, the attacker could change the saved return address to a standard I/O function and further exploit the `mmap-read` attack mentioned above.

In practice, isolation techniques manage the application's memory internally and do not forward memory-related syscalls such as `mmap` and `brk` to the untrusted OS. Our work shows legacy applications contain other Iago vulnerabilities caused by syscalls that will be directly forwarded.

Chapter 3

Related Work

3.1 Mitigation of Iago vulnerabilities

Isolation techniques usually take Iago attack into account when forwarding an application's service request to the untrusted OS. This section will study their methods of mitigating Iago attacks to understand their limitations. Isolation techniques try to mitigate the attacks while we see to identify vulnerabilities so they can be patched. Therefore, their work is complementary to ours.

Since the mmap-based attacks in the original Iago paper attract significant attention in the community, almost all isolation techniques have included checks to ensure that the returned address of memory management syscalls does not overlap with previously allocated memory. VirtualGhost[8] also introduces a random number generator to defend against the specific attack related to the entropy source. Other isolation techniques that have also addressed the original Iago vulnerabilities include: Trustshadow[13], AppShield[6], Sego[23], ShieldBox[36] and HiddenApp[39]. In the first version of SGX, all memory needed by the enclave is allocated during initialization with no further memory requests allowed. In SGX revision 2 [41], enclave programs can dynamically request EPC pages at runtime. Before accessing newly committed pages, the enclave memory manager must accept the allocation using EACCEPT instruction. This instruction will perform basic validation on newly assigned pages (e.g. non-enclave pages or pre-allocated pages). As a result, It is unlikely that SGX-based solutions will forward memory-related syscalls directly to the untrusted OS, which thwarts a major source of mmap-related attacks.

Most OFLs make some effort to narrow the syscall interface by only implementing certain syscalls. Minibox[25], SGX-Tor[21] and InkTag[14] handle part of system services with special care. InkTag has an application-level library to translate read/write syscalls into operations on memory-mapped files. Minibox divides all syscalls into sensitive and non-sensitive calls. Memory management, thread local storage management, multi-threading management, and file I/O are handled by Minibox internally. Both Minibox and InkTag still directly forward network I/O to the OS. This is because network is originally considered as an untrusted communication channel by the application and cryptographic protocols may be applied to help secure the channel.

Ryoan[16], SeCage[27] and Glamdring[26] claim that some checks are applied in the OFL to validate the return value of syscalls. However, no information is disclosed about what exactly those checks are. Panoply[32] studies the type of syscall return values and categorizes them into zero/error, integer value and structures. Their OFL will validate the returned error code as well as ranges of some integer return values based on POSIX semantics. For syscalls that return structure pointers or function pointers, Panoply states that developer annotations are needed for correct bounds. OpenSGX[18] only supports a limited number of syscalls so that it can carefully consider the potential attack surface on their narrowed interface. It relies on packet encryption, Network Time Protocol (NTP) server, random instruction and monotonically increasing integer to prevent potential Iago attacks.

As a strong form of mitigation, Haven[3], Graphene[37], and SCONE[2] mitigate the Iago vulnerabilities by placing a library OS, which provides OS services in the form of libraries, inside the isolated environment. This method replaces the complex syscall interface with a carefully designed small interface, which makes validation of values returned by the untrusted OS more realistic. For example, in Graphene, the library OS can track the offset of opened files and all `epoll` event data.

3.2 Iago Attack Analysis

There exist other works that analyze and detect Iago vulnerabilities. In this section, we studied the methods they use and compared them with our work.

Symbolic execution technique had been used to study the effect of Iago attacks. In symbolic execution, inputs are represented as symbols rather than concrete values as in normal execution. Thus, operations on symbols become expressions of symbols. Constraints are applied to the symbols according to each possible outcome of conditional branches encountered. Hong Hu et al. [15] combine symbolic execution with data dependency analysis to detect vulnerabilities introduced by privilege separation. Iago attack can fit into this model if the two separated partitions are kernel and application. Hong Hu and his team first run the program binary with given input to record the execution state. From the collected execution state, they extract suspicious read/write instructions whose address or data is controlled by the syscall return value. Then, they use symbolic execution to collect constraints representing the relationship between the interface inputs and the address or data used in the instruction. By analyzing those constraints, they can assess the capability of the attacker. Since they only analyze instructions executed with the given input, paths that reachable with the other program input and interface input are ignored.

In 2019, Jo Van Bulck et al. [38] analyzed responsibilities and attack vectors of a TEE shielding runtime. They generalized Iago attacks from the OS syscall interface to OCALLS in general, and detected Iago vulnerabilities in Graphene-SGX [37] and SGX-LKL [31] similar to the ones we found in Google Asylo. Their work is all manual and does not present a tool or method to automate discovery as we do.

COIN attack [20] performs the symbolic analysis on both inputs of ECALL and return values of OCALL (used to forward syscall for some systems) are studied. The authors of COIN attack paper emulate the execution with concrete values while collecting the path constraints in terms of symbolic variables. They then use a constraint solver to solve the constraints to generate a new seed (input) that can help explore other paths. By comparing the symbolic information with predefined policies, they can detect different types of memory vulnerabilities. Their work aims to detect errors in existing SGX projects, which are aware of the malicious OS. In contrast, we focus on legacy applications, and seek to provide guidelines to port them.

Chapter 4

Design of Emilia

In this chapter, we will first describe the threat model and our assumptions. Then we will discuss our fuzzing technique for detecting Iago vulnerabilities. We will introduce reverse syscall fuzzing and discuss its challenges compared with regular fuzzing. After that, we will define our objective as increasing syscall coverages under a fixed setting of an application (i.e. fixed command-line arguments, configuration). Finally, we will explain our design choices and strategies for achieving our objective.

4.1 Threat Model and Assumptions

Threat Model: We follow a typical threat model for Iago attacks. The following components are untrusted: (1) the OS and other system software, (2) other applications executing under the same OS, (3) syscall handler described in Section 2.1.1. We only trust the hypervisor or the designated hardware which creates an isolated environment. The protected application is benign but may contain vulnerabilities.

Assumptions: We assume the application is correctly protected by the isolation technique so that the malicious OS could not directly access the application's memory or other execution states. We assume the application's syscalls will be forwarded to the malicious OS, and syscall outputs will be copied to the application's address space by the OFL.

Pointer corruption: When the OS mounts Iago attacks, the manipulated return values can be used in a variety of ways. For instance, the return value of `getpid` can be used as an entropy source and the time provided by the untrusted OS can be relied on as timestamps to generate system logs. While such vulnerabilities can take various forms, we consider it the worst when code or data pointer can be corrupted [34]. In this thesis, we focus on Iago vulnerabilities that can result in pointer corruption. Data pointer corruption can lead to memory safety errors. Reads of this pointer could lead to data leakage, and writes can contribute to the corruption of data or other pointers. If the corrupted pointer is a code pointer, then the attacker could execute arbitrary code.

For certain pointer corruptions, the attacker may be unable to retrieve the information directly, but there are possibilities that the illegally accessed data can be revealed to the attacker through other channels. For example, a buffer containing data from out-of-bounds memory read may be written to a file or a network socket later.

Syscall's output argument included: In addition to the simple scalar return values considered in the original Iago attack paper, we also examine the buffers with OS-defined structures filled as the result of the syscall (whose members' value can still be scalar) because they are also in the malicious OS's control. Note that the OS is unable to directly overflow those buffers because the buffers are copied to the application's address space by the OFL as specified in Figure 2.1, and we assume the length of the buffer is known when performing the copy. For example, in `getsockopt`:

```
getsockopt(int sockfd, int level, int optname,void *optval, socklen_t *optlen)
```

`optlen` initially contains the size of the buffer pointed to by `optval`. When forwarding `getsockopt`, the OFL should record the original `optlen` provided by the application and copy at most that amount of bytes to `optval`.

Untrusted payload excluded: Even the trusted OS can return both trusted and untrusted values. For example, some syscalls are intended for data transfer between parties, with the help of the OS. Therefore, what the other party sends to the application does not represent the OS. For example, when the `read` syscall is used to read from a socket, buffer contents are by nature untrusted as they could come from network, but buffer length should be bear the same trust as the OS as it is set by the OS. For file contents, the application may trust the OS to perform access control and thus may assume the payload of `read` syscall is trusted. Fortunately, as we described in Section 2.1.2, isolation techniques transparently ensure the privacy and integrity of trusted file data with a proper configuration. However, the metadata (e.g. file size returned from `fstat` syscall) of the trusted file usually remains unprotected. Therefore, we only define vulnerabilities due to the misplaced trust in the syscall return values as Iago vulnerabilities and exclude untrusted or protected payload from our analysis.

4.2 Reverse Syscall Fuzzing

We use fuzzing to detect Iago vulnerabilities. Fuzzing is a general technique for testing a target application. A fuzzer repeatedly executes the application by feeding it with unexpected or random data. It then monitors the application to detect exceptions, such as crashes, assertions, or potential memory leaks. Compare to static analysis, fuzzing usually does not generate false alarms. For each bug found, fuzzing provides concrete inputs that can be used to reproduce and examine the bug. Emilia shows the ability to apply fuzzing on syscall return values.

Reverse syscall fuzzing is to fuzz the program making the syscall (i.e., applications) as opposed to the program handling syscalls (i.e. the OS). Instead of fuzzing the inputs to the syscalls, Emilia fuzzes the return values that the OS uses to respond to the application making the syscall. Compared to regular syscall fuzzers [19] and application fuzzers, which have been systematically studied [28] and are relatively more mature, reverse syscall fuzzing can be considered to be a new variant.

Overview: Figure 4.1 provides a high-level overview of how Emilia works. Applications running in user space do not call directly into the kernel. Instead, they will call a syscall wrapper function provided by the C library that knows how to invoke the syscall of interest. The wrapper will place the syscall arguments in the right locations and do whatever is necessary to invoke a trap into kernel mode. Thus, there are two places we can intercept a syscall. The first one is the actual syscall interface, where the syscall is invoked through special instruction, and the execution is trapped to the kernel. The other one is the syscall wrapper interface, where the application calls to the C library for constructing real syscall requests. We placed Emilia at the actual syscall interface. By doing so, we can analyze how a

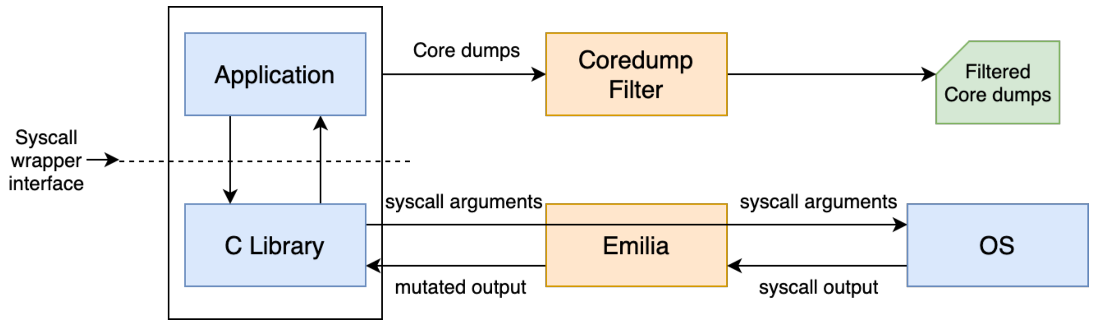

Figure 4.1: High-level Overview of Reverse Syscall Fuzzing Emilia

legacy C library handles syscall return values. Besides, some library functions that are not typical syscall wrappers will also lead to syscall invocations (e.g. `fprintf` will invoke `write` syscall in the C library). Intercepting at the syscall wrapper interface will miss those functions. In addition, isolation techniques such as Graphene [37] and SCONE [2] place the C library inside the trusted world. We believe most of the logic is ported from a legacy C library since it is expensive to write one from scratch. Therefore, it is worth analyzing potential Iago bugs in a legacy C library.

By sitting at the actual syscall interface, Emilia can intercept syscalls invoked by the C Library and mutate any syscall outputs provided by the OS before returning to the application. If a segmentation fault is triggered due to the abnormal return value, a coredump filter will briefly analyze the core dumped to remove duplicates. Therefore, the outputs of our tool are filtered unique core dumps. We define unique core dumps as cores with unique call stacks because using the absolute crash location (program counter) fails to distinguish some cases. For example, all crashes caused by the length argument of `memcpy` may end up crash at the same instruction in the C Library. But how the `memcpy` get called and which syscall's return value affects the length may be different. Notice that core dumps with varying call stacks sometimes may lead to the same vulnerability. For example, the same vulnerability in `fprintf` may result in multiple core dumps because `fprintf` is invoked at different locations in the application. We resorted to manual analysis for understanding the exact cause of each core dumps.

The most obvious difference between reverse syscall fuzzing and traditional fuzzing is their target input. We define target inputs as values a fuzzer is going to modify. For traditional fuzzing, the target inputs are inputs to the application/function. (e.g. arguments, environment variable and configuration files). They can be determined before doing the fuzzing. However, for reverse syscall fuzzing, the target inputs to mutate values on are syscalls' outputs. It has to wait for the application to invoke syscalls passively. When to invoke a syscall, which syscall to invoke heavily depends on the application's behaviour. Furthermore, the syscall sequence can vary due to the fuzzing itself. Considering an error-handling code snippet in Lighttpd [22]:

```
/* after fuzzing the return value of fstat with value other than 0, new write()
    syscall will be triggered */
if (0 != fstat(fd, &st)) { // 0 for success
    log_error_write("..."); // invoke extra write() syscall
    ...
}
```

Listing 4.1: An example of a new syscall invocation introduced by fuzzing

With a proper setup, the `fstat` syscall should succeed and return 0 during the vanilla execution (i.e. the execution with no fuzzed return values). In this case, `fstat` is our only target syscall. However, after fuzzing the return value of `fstat` to be non-zero, `log_error_write` in the error-handling path will be called and will subsequently invoke the `write` syscall that does not occur in the vanilla execution of the program. That means the total number of target syscalls are not deterministic before fuzzing and is affected by the fuzzing process.

Note that although reverse syscall fuzzing shares some commonalities with network protocol fuzzing [9], (e.g., fulfilling requests from the other party), network protocol fuzzing interact the application with protocol-specific semantics as opposed to application-specific. The syscall sequence is much more complicated and less well-defined than the message flow in protocol fuzzing.

4.3 Objective

Since fuzzing on the application's regular input (argument, configuration, e.t.c) has been systematically studied, Emilia will focus on finding Iago vulnerabilities by only mutating the syscall return values with fixed application's inputs. Users can provide different inputs and configurations to explore the path they are interested in.

Considering that the root cause of Iago attacks is the misuse or improper handling of syscall-returned results, Emilia's objective should be to search such code for vulnerabilities. Thus, an Iago vulnerability depends on both the syscall and the code that executes after the syscall. The objective of our reverse syscall fuzzing is to cover as many syscall invocations as possible, as well as the subsequent code paths that operate on the syscall-returned results. In summary, how a syscall return values get used by the application is reflected in two aspects: 1. Different locations in code. If the syscall is invoked in different code locations, the handling of its return values can be naturally considered different. 2. Different context. In the case of the same code, different local/global variables may lead to a different path being taken (e.g., there are two possibilities for `if (a > b)`), and the handling can also be different.

Therefore, compared to the general code coverage in regular fuzzing, we are mainly interested in *syscall coverage*. In this thesis, we define syscall coverage as how many unique syscall invocations we can execute without changing the regular input given to the application being fuzzed. This differentiates syscall coverage from *path coverage*, which is achieved by varying inputs to the application (command-line arguments, for example).

We show in 4.2 that applications may still execute different paths even if inputs are held constant if the return values of syscalls change. While both path coverage and syscall coverage are required to find as many Iago vulnerabilities as possible, reverse syscall fuzzer only focuses on achieving syscall coverage and may be combined with standard fuzzers. In this way, the standard fuzzer achieves path coverage and the reverse syscall fuzzer maximizes syscall coverage for each input generated by the standard fuzzer.

4.4 Measuring syscall coverage

To measure syscall coverage, one must have a notion of what constitutes a unique syscall invocation. Simply counting static syscall locations is insufficient because it does not consider path information leading up to and following the syscall—syscalls are often located in libraries (i.e. `libc`) whose functions may have many incoming and outgoing code paths. Similarly, certain naïve alternatives can be further ruled out. For example, using syscall name with arguments passed in also has significant redundancy because syscall arguments may vary for every invocation (file descriptors, loop iterations, pointer addresses, etc.) but are followed by the same handling logic for the returned results. In listing 4.2, for the same `read` syscall in line 3, the pointer(`buf`) allocated and passed in may vary for every run of the application. Also, the file descriptor is not deterministic since fuzzing on the `open` syscall in line 2 would change its value.

```
1 char* buf = malloc(1024);
2 int fd = open(...);
3 size_t ret = read(fd, buf, 1023);
4 buf[ret] = '\0';
```

Listing 4.2: An example of varying syscall arguments

To effectively identify all execution paths leading to and following from syscalls, one way is to collect the application's control flow directly (e.g., conditional/unconditional direct/indirect branches). One can envision employing application tracing, using either program instrumentation, or efficient hardware such as Intel Processor Trace (PT) [7], which collects such information that can be later retrieved in the form of data packets. However, tracing alone is insufficient because we must still be able to identify which executed paths were the result of syscalls. It means that we must either have debug symbol information or still recompile all code with instrumentation. As mentioned before, many applications invoke syscalls via libraries. Consequently, we must have debug symbol information or instrumentation not only for all application code, but also for all library code as well. Unlike standard fuzzers, which are mainly concerned with code paths in the main application, the libraries containing syscalls can play a role in Iago vulnerabilities, so they must be instrumented as well. To avoid instrumenting and rebuilding multiple libraries for each application and be able to find Iago across the application and libraries, we propose an alternative that does not require instrumentation or debug symbols.

A proxy for the path after a syscall is the path leading up to a syscall. Since the path leading up to a syscall usually explains the purpose of invoking the syscall, how a syscall return values will be used afterwards depends on those purposes. The syscall's call path (i.e. functions in the call stack) is a reasonable approximation of the path leading up to the syscall. We note that if two syscall invocations have different call paths, they must necessarily have different code paths both before and after the syscall, owing to the different caller and callees that must exist if the call paths are different. Moreover, the call stack, which gives us the call path, is easily accessible from the OFL without needing to instrument the application or special tracing hardware. Thus, we formally define a syscall invocation in Emilia as a tuple of syscall name (i.e. `read`, `write`) and its call stack (function name + offset) at the point the syscall is invoked.

4.5 Achieving syscall coverage

In summary, we need to execute as many syscall invocations as possible by using call stack as a means to check for uniqueness and varying syscall return values with constant application's regular input. To achieve "as many", it seems that we should keep trying with different return values as long as the syscall uniqueness still holds. A standard fuzzer can *specify* its target inputs (e.g. command-line arguments or functions in libraries) and try different values on them. Due to the non-determinism of syscall invocations, reverse syscall fuzzing needs a strategy to select syscall invocations for mutating their value.

On the contrary, a reverse syscall fuzzer needs to ensure that the application does not terminate too early. As already shown in Listing 4.1, fuzzing by itself makes subsequent syscall invocations vary, which is a double-edged sword: 1. We lose a fixed basis on which we can keep trying systematically. 2. Only through such variations, can we increase syscall coverage. If each run has the same sequence of syscall invocations, syscall coverage will become constant.

Since each iteration can encounter a varying number of syscall invocations and fuzzing them also affects subsequent invocations. To maximize coverage, we may want to fuzz only a subset of these syscall invocations observed. (We define one *iteration* as the cycle from when the application gets launched for fuzzing to when it crashes due to fuzzing or terminates normally.) We refer to the set of invocations to be fuzzed as *current targets* in each iteration.

Note that syscalls not selected as targets (hence not fuzzed) are still executed as part of their code path. Their OS-provided return values are passed directly to the application without modification by Emilia. The following section will explore the fuzzing strategies to achieve maximal syscall coverage with our tool Emilia.

4.6 Fuzzing Strategies

Our fuzzing strategies with Emilia reflect three important aspects that affect syscall coverage: target selection (whether an encountered syscall should be fuzzed), fuzzing value sets (what values to fuzz with) and return fields (which return fields of one syscall to fuzz).

4.6.1 Target Selection

We have shown that new syscall invocations could appear after fuzzing previous syscalls' return values. We need a method to select which syscall to fuzz for each iteration that can cover the new syscall generated.

Note a simple option of fuzzing one syscall invocation at a time (e.g. the first syscall invocation, then the second) does not work for new syscalls. To fuzz a new syscall, the previous syscall, which creates an execution path for the new syscall, must be fuzzed together.

Both stateless and stateful methods can be used to cover new syscall generated. The fuzzing state represents the information of which syscall's output field we have fuzzed and what value we have used for each field. A target syscall may only appear if preceding syscalls have been fuzzed. This state information can be used to fuzz the preceding syscalls in the same way so that the extra syscall in the mutated execution path can be analyzed.

Fuzz-all

A simple solution to address the new syscall problem is to fuzz all syscall invocations during the execution. In this case, all the syscalls in new paths and old paths will be fuzzed.

By doing so, the application would usually terminate early after fuzzing the first few syscalls due to an error. To keep going and fuzz syscalls afterwards, a variable `skip_count` is introduced to "skip" fuzzing the first `skip_count` syscall invocations. The `skip_count` will be incremented by one each time. For example, in Listing 4.3, the application will terminate after fuzzing the first `stat` syscall with a non-zero return value. In order to reach the following `open` syscall, the `skip_count` will be set to 1 so that the first `stat` syscall will normally return a zero in the next iteration.

```c
int ret = stat(...);
if (ret != 0) exit();
int fd = open(...);
...
```

Listing 4.3: An example of early termination with fuzz-all method

Since this method will fuzz a variable number of syscall invocations for each iteration, we would lose the fuzzing state because we would have no idea of which syscall invocation should we continue mutating its values for the next iteration. Even though we are forced to pick one and continue to change its return values, syscalls fuzzed in the next iteration would also be changed. That means the states we recorded for all the syscalls in the previous iteration may become useless. Therefore, we could only statelessly fuzz all the syscalls by randomly picking values and fields.

We found that this method also fails to trigger some vulnerabilities if there's an extra syscall between the vulnerable syscall invocation and the use of its return values. Thus the fuzz-all method will fuzz both the vulnerable and the extra syscall invocations. If the application terminates itself or changes its execution path due to the extra syscall's return value being fuzzed, the vulnerable syscall's return value will have no chance to be used. `skip_count` does not help because the vulnerable syscall is invoked before the extra syscall.

```c
1 /* the application calls syscall1 and then syscall2 */
2 ret1 = syscall1(); // ret1 is a vulnerable syscall return value
3 ret2 = syscall2(); // in vanilla run, syscall2 returns 0 on success
4 if (ret2 != 0) { // ret2 is less likely to be 0 if syscall2 also get fuzzed
5     return ERROR; // vulnerability in line 7 will be skipped
6 }
7 array[ret1] = var; // memory corruption caused by ret1
```

Listing 4.4: An example of missing vulnerability with fuzz-all method

For example in Listing 4.4, the code calls `syscall1` and `syscall2` in order. Assume the ther return value of `syscall2` is zero (`ret2 = 0`) in vanilla execution. If we only fuzz `syscall1`, a memory corruption error will be trigger in line 7. However, with the fuzz-all method, both `syscall1` and `syscall2` will be fuzzed with random values. A non-zero `ret2` will cause a return in line 5 and the vulenrable code in line 7 will not be executed.

Stateful Fuzz

In order to keep useful fuzzing states, we need to record the relationship of which syscall's return values can lead to new syscall invocations. Those relationships form a tree-like structure. Each node represents a discovered unique syscall invocation. The root of the tree is one of the syscall invocations in the vanilla syscall sequence. A child node is a new syscall invocation caused by fuzzing the parent syscall. We ignore already discovered syscall invocations even they are caused by fuzzing different parent syscalls to prevent cycles in the tree. There are many different methods to traverse the tree. We simply pick the recursive method to perform a depth-first search as it is intuitive. Algorithm 1 shows the pseudo-code of the recursive fuzzing loop.

Algorithm 1 Stateful fuzzing loop

```
 1: $overall\_set \leftarrow \emptyset$
 2: procedure MAIN_LOOP()
 3:     $vanilla\_syscalls \leftarrow extract\_vanilla\_syscalls()$
 4:     $overall\_set.add(vanilla\_syscalls)$
 5:     for syscall in vanilla_syscalls do
 6:         $target \leftarrow (syscall, init\_ref)$
 7:         $references \leftarrow [target]$
 8:         $recursive\_fuzz(references, 0)$

 9: procedure RECURSIVE_FUZZ(references, depth)
10:     if depth > max_depth then
11:         return
12:     $current\_target \leftarrow references[depth]$
13:     do
14:         $new\_syscalls \leftarrow start\_fuzzing(references)$
15:         $overall\_set.add(new\_syscalls)$
16:         for (syscall, hash) in new_syscalls do
17:             $next\_target \leftarrow (syscall, init\_ref)$
18:             $references.append(next\_target)$
19:             $recursive\_fuzz(references, depth + 1)$
20:     while $current\_target.update\_target()$
```

In every iteration, Emilia will store the fuzzing state into a reference list (`references`). Each element of the list contains the identifier for one target syscall (we will discuss this later) and a *value reference* describing which return field should be fuzzed with what value for this syscall. This reference list also serves as a guide for fuzzing. All syscall invocations in the list will be fuzzed accordingly.

The stateful fuzz first extracts a list of unique syscall invocations from the vanilla run and updates the `overall_syscall` set with `vanilla_syscalls` (line 3-4). `overall_syscall` is a global variable that records all the discovered syscalls. Then for every syscall invocation in the `vanilla_syscalls`, it runs the recursive analysis. The `init_ref` is the initial *value reference* of the target syscall invocation, and the content of the value reference will be updated in the do-while block in `recursive_fuzz` each time until it can not be updated further (values exhausted) (line 20). We will describe the update mechanism in the Sections 4.6.2 and 4.6.3. `start_fuzzing`(line 14) will launch the application and fuzz syscall invocations recorded in the reference list. Then Emilia will extract any new syscall invocations that are not found in the `overall_syscall` set. For each newly founded syscall invocation, we will append it to the reference list and go to the next level of recursion (line 17-19).

As a result, a syscall in the reference list is a new syscall invoked in the new execution path caused by fuzzing its previous syscalls. In most cases, the reference list as a stored state will help replay the previous execution by filling the fuzzed syscall return fields with the same values. However, the application could also be affected by the OS-returned values to the unfuzzed syscalls. For future work, we could record the whole execution state or taking a snapshot of the execution.

We use experiments to understand the trade-off between the stateless and stateful methods (see Section 7.1). It shows that the stateful method can cover more syscall invocations than the stateless one while also requiring more time to run.

4.6.2 Fuzzing Value Sets

The values we select to fuzz each syscall return field are important in discovering new syscall invocations and triggering memory corruption bugs. Ideally, we could iteratively flip the bits, store the state and start mutating other bits when finding new paths as what regular guided fuzzing does. However, an application has hundreds of syscall invocations, and we have to do it recursively to handle new syscall invocations systematically. Moreover, the syscall return values without file/network payload do not affect code coverage much compared to program inputs and configurations. So if we have infinite resources, we would rather use regular fuzzing on input/configurations and combine the generated input with our reverse syscall fuzzing.

As a tradeoff, we prepare a finite value set for each return field of each syscall. The `update_target` in the do-while loop in Algorithm 1 will iteratively update the value reference with values inside the value set. A value set consists of a valid value set, an invalid value set and a few random values generated at runtime. The valid value set contains values extracted from static analysis to increase path coverage. For example, in Listing 4.5 , the return value `ret` of `read` syscall is compared with a constant 10 in a conditional branch (line 2). To explore both branches, we would add 10 and 9 into the valid value set of `read`'s return code. An experiment was performed to evaluate the effectiveness of valid values. (see Section 7.2)

```
1 int ret = read(...);
2 if (ret < 10) { // compare the read return vlaue with a constant 10
3     write(...);
4     ...
5     return;
6 }
7 poll(...);
8 ...
```

Listing 4.5: An example of extracted valid values

Both the invalid set and random values aim to trigger the crash once the value is used on pointer arithmetics. Currently, the invalid set contains two bound values: MIN and MAX. The interceptor will fill the target return buffer with little-endian, twos-complement MIN and MAX when seeing them in the value reference. For random values, Emilia randomizes not only the bytes of the output but also the number of bytes to overwrite. In this way, Emilia has a higher possibility of generating value with different orders of magnitude. We did this because some memory corruptions can not be triggered with a too large value. For example, in OpenSSH, the `read` return value will be used first to reallocate a buffer then perform pointer arithmetic on another buffer. If the value is too large, the reallocation will fail, and the program will not go further.

4.6.3 Return Fields

A syscall could have multiple return fields. We need to decide which of them to fuzz in each iteration. A systematic and time-consuming way is to try every combination of the return fields and values. The number of combinations increases exponentially with the number of the return fields. For example, `stat` has 14 return fields when broken down (i.e., the return value + 13 fields in `struct stat`). Assume we only have 3 values for each field to try (including an option to not fuzz this return field). There will be 3^{14}

combinations for one syscall invocation. To make the fuzzing finish in a reasonable amount of time, we choose to fuzz one field at a time, so that the time to fuzz a single syscall invocation grows quadratically with the number of fields and values (i.e. $num\_fields \times num\_values$) as opposed to exponentially. In this way, the `update_target` method will iteratively go through every value of every output field.

Chapter 5

Implementation of Emilia

5.1 Overview

An overview of our reverse syscall fuzzing framework is shown in Figure 5.1. Emilia consists of three components: interceptor, controller and value extractor. We divided the interception of the application and fuzzing loop regulation into an interceptor and a controller. In each iteration, the controller feeds the interceptor with references (described in 4.6.1), then the interceptor will launch the application's binary and actually perform the fuzzing instructed by the references. After the application finishes or crashes, the controller will analyze the syscall invocation data to detect new syscall invocations and prepare references for the next iteration. The controller will also briefly examine the causes of the termination by inspecting the application's exit code and its core dump. The outputs of Emilia are unique core dumps(Section 4.2) filtered by the controller. Emilia requires manual analysis on core dumps generated. A client instance will be launched for continuing the tested server's execution if necessary. The source code of the application will go into the Value Extractor for generating valid values (see Section 4.6.2). The controller will select values from them when constructing the reference list.

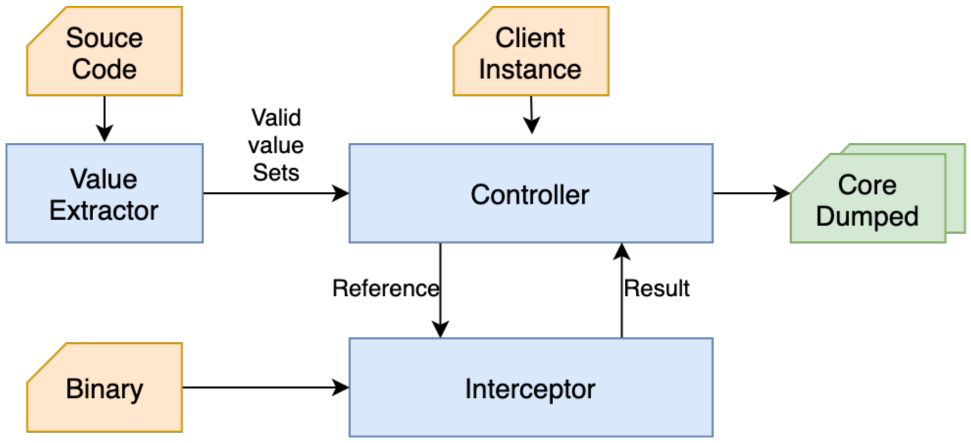

Figure 5.1: Components and workflow of Emilia

5.2 Interceptor

When a syscall is trapped, the interceptor will replace values in the return fields/buffers in the application's address space and registers (e.g. `$rax`) with fuzz values and continue execution. In the case of buffers (i.e., pointer arguments passed to the syscall), the interceptor determines the buffer size either based on its type or from other arguments as mentioned in Chapter 4.1. The interceptor would also calculate a hash of stack trace for each syscall encountered to help identify them for syscall coverage.

We build the interceptor based on *strace* [33]. *strace* is a userspace utility for Linux to monitor and tamper interfaces between processes and the kernel. *strace* make intercepting syscall possible by a feature known as `ptrace`. A tracer process can call `ptrace` to observe and control the execution of another process(tracee). After calling `ptrace` with `PTRACE_SYSCALL`, the tracer can use wait syscalls (e.g. `wait4, waitpid`) to wait for the tracee to stop. The stop will be trigger twice on both the entry and exit of a syscall. The fuzzing of the return values happens at the exit point.

5.2.1 Stack hash extraction

The original *strace* utilize *libunwind* to extract the call stack of each syscall invoked if option `-k` is specified. However, *libunwind* is not designed to be performance sensitive. As table 5.1 shows, stack trace implemented by the original *strace+libunwind* introduce a 6x overhead.

To optimize the stack trace function, we first need to understand where the bottleneck is. In brief, whenever encountering a syscall invocation, the stack trace function will execute a for-loop consist of two parts:

1. Go to the next frame. It walks through the program stack to retrieve the return address of the next frame.
2. Get function name and offset. It uses the return address to recover the function name and offset from the program image.

The number of iterations executed in this for-loop depends on the call stack depths of the syscall. By performing a microbenchmark on the stack trace function, we find that the get function name part takes most of the execution time (16:1). Therefore, we further study the code of this part in *libunwind*.

libunwind first reads and parse the `/proc/pid/maps` file to get the elf image path corresponding to the return address. Then it loads the image file or the debug link file extracted from the image file. For each function symbol in the symbol table, *libunwind* calculates the function's distance to the return address. Finally, it returns the name and offset of the function that closest to the return address. We find that *libunwind* will go through the whole process without caching for each of the return address. That means, for one syscall invocation with `n` call frames, it will reload the map and image file `n` times, although those files are unchanged during the process.

Therefore, we decide to cache the map file and function symbols when they get loaded. Fortunately, we find *strace* itself caches the maps file to ignore some unnecessary lookup. *strace* also keeps the cache fresh by rebuilding it after syscalls that can affect memory mapping. (e.g. `mmap, mprotect, munmap, execve`). We take advantage of it to pass *strace*'s map-cache to the *libunwind*, and refresh the symbol cache along with the map-cache.

As a result, we make the stack trace process approximately 3 times faster than the original *strace and libunwind*. (see Table 5.1).After getting the stack string (function name and offset pairs), we hash it

Application	strace w/o stack trace	stack trace (original)	stack trace (cached)
git clone	548.1125855	3836.559932	1095.884483
openssh	47.34340906	341.8519902	105.715909
lighttpd	47.02815771	676.3854456	179.5955133
memcached	42.62156487	322.8056741	146.6278839

Table 5.1: Stack trace overhead (ms)

to a 32-bit unsigned integer with Murmurhash. The interceptor will output the syscall name and stack hash for all the syscalls invoked. Thus, the controller can figure out if new syscalls appear after fuzzing.

5.2.2 Return value mutation

When intercepting the exit point of a syscall invocation, the interceptor first compares the syscall stack hash with the reference list (see Section 4.6.1). If it finds a match, and this syscall has been fuzzed less than 10 times during this iteration (to prevent infinite retry), the interceptor will start mutating the return value.

We manually write one fuzz function for each supported syscall that returns structures or buffers. We need to specify the size and location of each output field for different structures. For value-result argument/field that describes the buffer's max length, we need to tell the interceptor to use input length recorded at the syscall entry point when filling the buffer. The fuzz function creates a local variable in its address space for each output of the syscall. For example, the fuzz function for `fstat` contains `struct stat fuzz_stat` and `int fuzz_ret`. For buffer output whose length depends on another input argument, (e.g. the max length of `optval` in `getsockopt` should equal to the original `optlen`), we dynamically allocate the buffer. The buffer's length must be retrieved at the entry point of the syscall if it is a value-result argument (`optlen`). Fortunately, *strace* already has the feature to record and retrieve those old values. (*strace* prints `[old -> new]` for value-result argument when tracing syscalls). After filling those local variables according to the value reference, they will be written back to the application's address space. *strace* achieves this with `process_vm_writev` and `ptrace(PTRACE_POKEUSER)` calls.

As described in Section 4.6, the reference for the target syscall contains a field index and a reference value. The field index tells the interceptor which output field to fill, and the reference value specifies its content. There are three types of values: 1. concrete value, 2. random and 3. MIN/MAX.

concrete value: The concrete values are integer values extracted by the value extractor. They will be copied to the corresponding field directly.

MIN and MAX: If the reference value is MIN or MAX, the target field will be filled with a two's Complement MIN/MAX in little-endian. For example, if the length of a target field is 4 bytes, the interceptor will write the value `0x800000` or `0x7FFFFF` for MIN and MAX. If the target field is a signed number, this method can provide both positive and negative values. If the target field is unsigned, it still can provide a large enough value for triggering memory corruption.

random: To replay the previous execution, we need to use the same value to fuzz recorded syscalls (See section 4.6.1). Therefore, we need a method to store the random values in the previous iteration. We can not determine the random value ahead in the controller and treat it the same way as concrete values, because the length of some buffer can not be determined until runtime (again `optval`). As a result, the controller only set a randomly generated file name to the interceptor. The interceptor will generate random bytes to fill the target field and create the file to store the value for the first time. If the file exists, the interceptor will directly fill the field with the file content. The interceptor also extends the file with random bytes if the file size is smaller than the target field's length. Before starting to try out the next value in the value set (update target), the controller removes the previously used random file.

Filling the field with random bytes favours values in large magnitude. However, we find that some crashes will only be triggered if the return value is small. As a result, when generating random bytes, we also randomize the number of bytes. For example, if the target field is 4 bytes, the number of bytes (`n`) will be randomly picked from 1-4. Once `n` is determined, we will fill the first `n` bytes of the target field with random values, and fill the remaining bytes with `0x00` or `0xFF`.

5.3 Controller

The entire fuzzing process is coordinated by a controller, which invokes and feeds the other components with instructions. The role of the controller is threefold: 1. Target selection. The controller regulates the fuzzing loop and selects the fuzzing targets for the interceptor (which syscalls to fuzz with what values). For syscalls that are not fuzzed, we just let them pass through with the OS-returned values. 2. Satisfying external conditions. Sometimes, the application may have external dependencies for continuing its execution. In particular, if the application is a server, the interceptor will send a signal to the controller when the accept syscall (a syscall indicating the server is ready to handle client connections, which could be `accept`, `select`, `epoll_wait`, etc.) is reached. Upon receipt of the signal, the controller will launch the corresponding client to connect to the server application so that fuzzing can continue. 3. Core dump analysis. If a core dump is produced after the application crashes, the controller would also briefly analyze them for duplicates and filter out the ones that are not caused by memory corruptions (e.g. assertion error).

Note that not all syscall invocations will be selected as fuzzing targets by the controller. Some syscalls are not suitable to be directly forwarded to the untrusted OS. 74 syscalls related to threads, memory management and signals (e.g. `fork, mmap, sigaction`) are usually specially handled by the isolation techniques. The untrusted OS is usually not allowed by the isolation technique to directly manage threads and signals because those operations involve manipulating the application's address space. After the publication of the Iago attack, almost all isolation techniques implement their own memory management handlers to address the mmap-based attack. Since they are handled separately, the interfaces might be changed, and careful checks might have already been applied to the interfaces. (see Section 3.1). As a result, we exclude those 74 syscalls from our analysis since the untrusted OS will not handle them through an unverified syscall interface.

The controller part of Emilia is written in python3. The main logic of it is described in Algorithm 1. It uses `subprocess` to launch the interceptor with configurations in each iteration. The interceptor does all the fuzzing. The controller does not interact with the interceptor unless the target application is a server. For sever applications, the controller specifies its `pid` to the interceptor through a configuration option. Once the accept syscall is invoked, the interceptor will send a customized signal to that PID so that the controller can launch a client instance for further processing. We write most of the client instances python functions. For example, a simple client instance for a web server is a python function that sends a URL request with *request* package. The user could define its own client function or launch the client binary with subprocess.

The controller is also responsible for storing dumped cores. During the initialization phase, the controller will set the `core_pattern` kernel configuration so that all the crashed cores will be dumped to a specified temporary directory. After each iteration, the controller will check that directory to see if any new core is generated. For each newly dumped core, the controller launches a gdb script to extract a hash of the core's call stack. Both the core and the fuzzing log will be renamed and moved to a permanent directory if the call stack hash is new. Otherwise, they will be cleared before the next iteration. Further analysis of those cores is performed manually.

5.4 Value extractor

The value extractor generates a valid value set for each output field of syscalls. The value extractor is a coarse-grained LLVM (a open-sourced compiler project [24]) analysis pass. The syscall wrapper looks for any usages of the return value and output arguments for each call instructions to the syscall wrapper. Suppose the return value is used in a binary operation instruction (e.g. add and sub) with an immediate value, the extractor will update the offset and recursively search usage of this binary operation. If the value is used in a store instruction, we will further analyze its corresponding load instruction. If the usage is a comparison instruction (e,g, equal, less than) and the other operand is constant, we will output this relationship for value generation. For example, `ret + 3 < 5` will be represented as `var1 = ret + 3` and `var2 = var1 < 5` by LLVM. When we see the usage of the return value in the first add instruction, we will update the offset to 3 and looking for usages of `var1`. Then in the comparison instruction, we output `ret < 2`. For all the relationships in the form of `ret operator n` (`n` is a constant integer), we simply add `n`, `n-1` or `n+1`(depends on the operator) into the target field's value value set. (e.g `read`'s `ret`, `fstat`'s `ret` and `fstat`'s `stat.st_mode`)

Libc syscall wrappers move the negative part of the syscall return value to a global variable `errno`. So the extractor also searches for comparison instructions with `errno` involved. To find the syscall related to an `errno`, we use a heuristic that `errno` is usually used immediately after the call. Thus, we search forward to find the closest function call instruction. If it is a call to a syscall wrapper, we add the extracted negative value to the valid set of that syscall's `ret`. Otherwise, we simply add the value to all syscall's `ret`.

Chapter 6

Vulnerability Analysis

This chapter starts with the classification of vulnerabilities found by Emilia and provides examples of each. We then quantitatively examine our measurement results to describe how frequent the Iago vulnerabilities arise in legacy code. Then we discuss our insights into why Iago vulnerabilities arise, and at the same time, why they don't arise more often. After that, we summarize some lessons-learned that will provide directions for avoiding Iago vulnerabilities for legacy code in applications. Finally, we describe our extra work on analyzing a commercial OFL implementation and Iago vulnerabilities detected in it.

6.1 Applications examined

We examine 17 applications and libraries, including servers, clients, and utilities. They are summarized in Table 6.1. We choose these applications because they are likely to be treated as targets in different isolation techniques[2, 40, 37]. They are largely I/O-intensive, meaning they tend to make a lot of syscalls, and are all UI-less applications (we test the headless version of Chromium) making them easier to fuzz. We also analyze the C library code invoked during fuzzing since we intercept the actual syscall layer instead of libc wrappers. Our system was running *glibc-2.27*.

6.2 Classification

We consider all the memory corruption bugs as potential Iago vulnerabilities though the severity of those bugs may vary. Memory corruption bugs are caused by a corrupted pointer get dereferenced. The dereference could be either a memory write or memory read. For memory write, if the attacker could control both the write address and content, he could overwrite a stored return address to hijack the execution further. In most cases, the attacker can only control the invalid write operation's address, and the value to overwrite is limited by the code's logic. We believe there's a potential for the attacker to overwrite bytes of some important variables (e.g. a boolean variable decide whether to encrypt network packages or not) to affect its execution. For memory read, if the read content is exposed to the attacker through some channel (e.g. pass the value as a syscall argument), sensitive information could be leaked. In some cases, bytes read are only used internally. We believe there's still a potential for the attacker to learn some knowledge of the bytes read by observing the application's behaviour change if the read

App	Ver.	Description	LOC
openSSH	7.9p1	SSH server and client	91,607
Lighttpd	1.4.51	light-weight web server	49,688
Apache	2.4.37	HTTP Server	184,033
MongoDB	r4.2.4	document-based, distributed database	1,957,478
Redis	5.0.5	key-value database	115,034
Nginx	1.17.0	web server	132,911
Memcached	1.5.20	memory object caching system	18,414
Evolver	2.70	liquid surfaces modelling system	130,104
Charybdis	3.5.5	IRCv3 server	191,478
BOINC	7.14.2	volunteer grid computing system	222,388
Chromium	74.0	web browser	21,140,796
Git	2.18.0	version control system	210,732
wolfSSH	v1.4.3	lightweight SSHv2 server library	22,533
Coreutils	8.31	GNU operating system utilities	62,466
zlib	1.2.11	data compression library	18,334
libreadline	7.0	command lines editing library	21,728
curl	7.72.0	command lines web client	130,833

Table 6.1: Legacy applications analyzed

value is used on some branch conditions. Therefore, we just count all the memory corruption bugs found as potential vulnerabilities. Further work is needed to determine whether they are exploitable and how the attacker could take advantage of them.

We classify the Iago vulnerabilities by the nature of the assumptions they violate. Every syscall has semantics that a correct OS adheres to, so naturally, programs will assume the OS will obey such semantics. We categorize these semantics into four types:

Static: Semantics are independent of syscall arguments and history. For example, certain syscalls return a negative value as the error code, which can be checked against a predefined list, such as the negative value returned by `accept`.

Local: Semantics are only dependent on the arguments that are *local* to the syscall. For example, the returned number of bytes processed (read/written) needs to be less than or equal to the specified buffer length as an argument, as in `read` and `getsockopt`. We also find that some similar syscall return values do not have this restriction. For example, the manual page of `accept` syscall specifies that the returned `addrlen` can be set greater than the provided max length. We do not consider the bug caused those return values as Iago vulnerability as it could happen with a benign OS.

Stateful: Semantics are dependent on the history of previous syscalls. Certain states can only be affected by the application itself, in the case of a well-behaved OS. A representative example is the current read/write pointer of an open file, which is only determined by the previous syscalls the application has invoked (i.e. the return values of `epoll_wait` depend on previous invocations and returns).

External: Semantics depend on information external to the application. Examples include randomness from the Iago paper and time. In theory, without duplicating the corresponding functions within the OFL, it would be infeasible or impossible to verify such semantics.

6.3 Vulnerabilities found

App	Syscall	Count	Type
Redis	accept	1	Static (1, 2.0%)
openSSH	read (27)	2	Local (41, 82.0%)
Apache-httpd		6	
MongoDB		1	
Redis		5	
Nginx		1	
Evolver		1	
BOINC		1	
Chromium		1	
Coreutils		2	
zlib		1	
curl		1	
libreadline		2	
glibc		3	
openSSH	readlink (7)	1	
Redis		1	
libreadline		4	
glibc		1	
openSSH	getsockopt	1	
Lighttpd	getsockname	1	
zlib	write	1	
Redis	epoll_wait	1	
Memcached	recvfrom	1	
glibc	recvmsg	1	
glibc	getdents	1	
Lighttpd	epoll_wait (6)	1	Stateful (6, 12.0%)
Apache-httpd		1	
MongoDB		1	
Redis		1	
Charybdis		1	
Chromium		1	
Git	lseek	1	External (2, 4.0%)
glibc	fstat	1	
Total		50	

Table 6.2: Detected Iago vulnerabilities

We run the stateful fuzzing on 17 real-world applications for about 80 hours in total. We provide a subset of per-application runtimes in Table 7.2 where the measurements were taken from an average of 3 runs (the variance was small in all cases). 50 memory corruption Iago vulnerabilities are discovered in those applications and the involved part of glibc. We manually analyze those memory corruptions to make sure all 50 vulnerabilities are caused by violating one of the syscall semantics. The majority of vulnerabilities are Local (82.0%), followed by Stateful (12.0%). See Table 6.2 for the list and statistics. We will discuss a few examples based on the four types of semantics.

Static: Many of the syscalls will return a positive value on success. The negative return value will be interpreted as an error code, and the C library will move it to `errno` and set the return value of the syscall wrapper function to `-1`. However, `glibc` will not perform this translation if the negative value is less than `-4095` because all valid error codes should fit into this range. In Redis, we find a piece of code that uses a file descriptor returned from `accept` to index a pre-allocated file descriptor array (Listing 6.1). Before performing indexing, Redis checks the returned file descriptor with `-1` in line 2 and then compares it against the array's max size in line 8. Usually, those checks are sufficient to prevent buffer overflow since Redis assumes the negative value returned from the syscall should be a valid error code and be moved to `errno` correctly, `-1` should be the only negative return value from the glibc syscall wrapper. Thus, a crafted negative return value less than `-4095` will skip the translation of glibc, pass all those checks and cause out-of-bounds indexing in line 11. The attacker could make `fe` point to a selected address and uses the following dereference instructions to change memory content around that address. For the syscalls that are not allowed to return any negative values except error codes, the OFL could check the negative part against a predefined valid value list to prevent such vulnerabilities.

```
 1  int fd = accept(...);
 2  if (fd == -1) {
 3      // error handling
 4      return error;
 5  }
 6  ...
 7  /* compare fd with max number of file descriptors tracked (max size of eventLoop
        ->events[]) */
 8  if (fd >= eventLoop->setsize) {
 9      return error;
10  }
11  aeFileEvent *fe = &eventLoop->events[fd]; // use fd to index
12  fe->mask |= mask;
13  if (mask & AE_READABLE) fe->rfileProc = proc;
14  if (mask & AE_WRITABLE) fe->wfileProc = proc;
```

Listing 6.1: An example (accept) of the Static semantics in Redis

Local: Syscalls such as `read` and `getsockopt` will fill a buffer provided by the caller. There is always an input value to specify the buffer's max length so the OS will not overwrite the buffer. Upon completion, the syscall sometimes returns a value to indicate the actual size it has written into the buffer. In most cases, a benign OS should never return a value larger than the specified max length. Listing 6.2 shows a vulnerability caused by an unbounded `readlink` return value in OpenSSH.

After reading the symbolic link's content into the `buf`, the program tries to form a zero-terminated string by adding a zero at the end of the string. Normally, the returned `len` should be equal or less than the input length (`PATH_MAX - 1` in this case) based on the specification of the `readlink` syscall. So the application feels safe to index `buf` with `len` in line 7. However, a crafted `len` which breaks the assumption will let the attacker set any byte beyond `buf` to zero. If the attacker can guess the application's memory layout, he could use this vulnerability to change a boolean variable on the stack to zero and change the application's behaviour.

```
1  char buf[PATH_MAX];
2  if ((len = readlink(path, buf, sizeof(buf) - 1)) == -1)
```

```
    ...
    /* error handling */
else {
    ...
    buf[len] = '\0';
    ...
}
```

Listing 6.2: An example (readlink) of the local semantics in OpenSSH

Stateful: Some state information involved in syscalls is supposed to be exclusively controlled by the application. The application may make assumptions on return values regarding such state information based on its previously invoked syscalls. To verify this type of return values, a stateful OFL that can keep track of all related syscalls is necessary. Syscalls like `epoll_wait` and `epoll_pwait` will return user data corresponding to the polled file descriptor. The user data should contain the same data as was stored in the most recent call to `epoll_ctl`. This user data usually specifies a file descriptor or a pointer. If the application dereferences the pointer returned by a malicious OS, the vulnerability will occur. The common usage of `epoll` will also use the returned file descriptor to index a pre-allocated array to extract the data regarding this file descriptor. Listing 6.3 shows an Iago vulnerability caused by `epoll_wait` in Lighttpd.

```
 1  n = epoll_wait(ev->epoll_fd, ev->epoll_events, ev->maxfds, timeout_ms);
 2  ...
 3  ndx = 0;
 4  do {
 5      ...
 6      fd = ev->epoll_events[ndx].data.fd
 7      handler = fdevent_get_handler(ev, fd);
 8      (*handler)(srv, context, revents);
 9  } while (++ndx < n)
10
11  fdevent_handler fdevent_get_handler(fdevents *ev, int fd) {
12      if (ev->fdarray[fd] == NULL) ERROR();
13      if (ev->fdarray[fd]->fd != fd) ERROR();
14      return ev->fdarray[fd]->handler;
15  }
```

Listing 6.3: An example (epoll_wait) of the stateful semantics in Lighttpd

`ev->epoll_events` is an output buffer of `epoll_wait`. The malicious OS controls its content after the syscall returns. `fd` is an integer value returned in this buffer (line 6). Lighttpd retrieves the corresponding handler function pointer by indexing `(fdnode)ev->fdarray` with the returned `fd` in `fdevent_get_handler` (line 14). Then the function pointer gets called in line 8.

We will show that the attacker could fully control the function pointer if he knows the memory layout. `ev->fdarray` is placed at a lower address of `ev->epoll_events`. By setting `fd = 4100`, the attacker could make `fdarray[fd]` point to the region inside `ev->epoll_events` buffer (`&fdarray[fd] == &epoll_events[1].data.ptr`). Since the content of `epoll_events` buffer is controlled by the attacker, he can then set `epoll_events[1].data.ptr = &(epoll_events[2])`

and craft a valid `fdnode` structure there (set `(fdnode)epoll_events[2].fd = fd` to pass check in line 12 and 13). Finally, the attacker can set `(fdnode)epoll_events[2].handler` to any code address he wants and gain control of the execution.

A similar vulnerability is found in Charybdis, the `data` field of `epoll_event` stores a pointer to a structure which contains a function pointer. If the attacker let the pointer point to a controlled buffer and write a function pointer there, he can make the application call arbitrary functions.

External: Some syscall return values describe a state that can not be maintained by the application, and they do not have clear invariant as Static or Local semantics do. Examples are the file or network content obtained from `read` syscall, time received from `gettimeofday`, and file size received from `stat`. The application may have the ability to affect those values by performing operations like writing to a file, but the external world could also change it.

We find one such example in glibc's code of parsing `ld.so.cache` (Listing 6.4). It uses the file size retrieved from `fstat` to `mmap` the same file in line 5. Then it assumes the file content is written in a specific format and casts the buffer to `struct cache_file` in line 8. If the malicious OS returns a small file size (`cachesize`). glibc would `mmap` less pages to `cache`, and the following parsing based on the file format would eventually access unmapped memory. Although glibc verifies pointers with the macro defined in line 1 before using, the `cache_data_size` itself could be miscalculated. In line 10, if the `cachesize` is smaller than the offset of `cache_data`, `cache_data_size` would be a very large number since it is unsigned.

In `git_config_set_multivar_in_file_gently` function of Git, it tries to modify key-value pairs in the config file by copying file contents to a temporary lock file part by part. It first parses the config file and records each parsed element's position by calling `lseek(fd, 0, SEEK_CUR)`. Then it `mmap` the config file to a buffer named `contents` with the file size read from `fstat`. During this procedure, Git assumes the config file is owned exclusively by itself and uses the lock file to prevent access from other Git processes. Therefore, Git expects the recorded file offset to be smaller or equal than the file size they read from the `fstat` and uses the recorded file offset to index the `content` buffer. In this case, both file size and file offset describe a state that cannot be maintained by the application alone (`lseek` with `SEEK_END` will set the file offset based on file size).

```
1  #define _dl_cache_verify_ptr(ptr) (ptr < cache_data_size)
2  if (fstat(fd, &st) >= 0)
3  {
4      sizep = st.st_size;
5      result = mmap(NULL, sizep, prot, MAP_FILE, fd, 0);
6  }
7  cachesize = sizep;
8  struct cache_file* cache = result;
9  cache_data = &cache->libs[cache->nlibs];
10 uint32_t cache_data_size = (const char *) cache + cachesize - cache_data
11 ...
```

Listing 6.4: An example (fstat) of the external semantics in glibc

A malicious OS could also compromise the application through return values ways other than memory corruption. For example, Apache used `getpid` and `time` as a random source, which was mentioned in the original Iago paper. Those vulnerabilities are ad-hoc and hard to detect automatically. Extra work such as modifying the application logic and adding a trusted random and time source is necessary to mitigate those vulnerabilities.

In summary, we have found at least one vulnerability in every application we examined except wolfSSH. They are listed in Table 6.2. From the table, we can see that 82% of the vulnerabilities are caused by a returned size, which goes beyond the local upper bound. (The `epoll_wait` vulnerability in category Local is caused by a returned number of file descriptors which is larger than the specified `maxevents`.) It is predictable since the returned length is likely to be used to access the buffer sent to the same syscall for some common programming practices. For example, iterating the buffer using the returned number of items, adding a zero to the end of the received data to terminate a string and copying the buffer using a "smaller" size to save space. The `epoll_data` returned from `epoll_wait` caused most of the Stateful vulnerabilities because storing a file descriptor in `epoll_event` and using it to index a file descriptor array to extract more data associated with that file descriptor is also a common practice. On the other side, syscall-returned data that is not related to any buffer position is unlikely to be used for pointer arithmetics. That is why the types of vulnerable syscalls which can trigger memory corruption are limited.

6.4 Mitigating Iago vulnerabilities

An obvious way of mitigating the Iago vulnerabilities is to check if the semantics of syscalls have been violated. This could be done by either the application itself or the OFL. Also, for each type of syscall semantics, the implications and difficulty might be different.

Local and Static: Since this type of semantics can be checked against predefined ranges or other constraints without maintaining a state, the checks can be simply performed by the OFL and they are straightforward and relatively cheap to do. For example, the OFL can check if the returned size is smaller or equal to the maximum length specified in the parameter. The high number of Local vulnerabilities in Table 6.2 suggests that the majority of Iago vulnerabilities can be mitigated in this way, and this would eliminate **84**% of the vulnerabilities for Static and Local. The semantics might need to be manually derived from the OS code or syscall specifications for the OFL check, but this would only be a one-time effort for each OS version.

Stateful: In contrast to Static and Local which are straightforward to check, Stateful Iago vulnerabilities require more complex logic to maintain parallel states with the untrusted OS (e.g., keeping track of the syscall history). However, we note that the main motivation of many user-TEEs is to reduce the TCB of security-sensitive code. Since the OFL is in the TCB, it must also remain small as well. Implementing a stateful OFL will thus increase the TCB, which is antithetical to the philosophy of TEEs. Therefore, instead of purely relying on the OFL, an alternative is to patch the application to free from vulnerabilities. We find that all 6 applications we examine that contain an `epoll_wait` vulnerability can be easily configured to use other polling syscalls such as `poll` and `select` for compatibility reasons.

External: We have argued that the application should not make assumptions on resources that it does not control or keep track of. The root cause of the `fstat` and `lseek` bug in Git and glibc is the assumption that it owns the file exclusively, which is not true even in a common threat model (the OS is not malicious but with other applications running in parallel). External metadata, such as file size, should also be crypto-protected to prevent those vulnerabilities. Ad-hoc vulnerabilities such as mistrusted random sources (causing other application failures) can be mitigated through improved application development.

Our study shows that Emilia can help detect vulnerabilities and help application developers identify where applications need to be rewritten to avoid Stateful Iago vulnerabilities. By detecting the presence of Stateful Iago vulnerabilities, these vulnerabilities can be patched in the application, thus avoiding increases in the TCB of the OFL to perform stateful checks. Adding up Static and Local, **84%** of the vulnerabilities can be checked and mitigated in a straightforward manner.

6.5 Forwarding OFL Analysis

Compared to the threat model inconsistency faced by legacy applications, the Iago vulnerabilities have been included in the OFL development threat model. With that said, we are still interested to see whether or to what extent, commercial OFL implementations can defend against Iago vulnerabilities.

Google Asylo [10] is a two-year-old (first commit on GitHub was May 3, 2018) enclave application development framework with 32 contributors on Github (101,131 LOC). It aims to help developers take advantage of a range of emerging trusted execution environments (TEEs), including both software and hardware isolation technologies.

In Asylo, a subset of POSIX calls made in the enclave will be forwarded to the untrusted side. It shares the same syscall forwarding model described in Section 2.1.1. Trusted OFL performs the forwarding in the enclave while handler functions make actual syscall calls in the untrusted world. For some of the calls, outputs are copied based on predefined rules. For other calls, specific codes are used to parse and copy the returned value.

Instead of directly applying Emilia, we intercept Asylo's untrusted syscall handler which handles and replies syscall requests forwarded by the OFL. The fuzzing loop algorithm we use in Emilia that enumerates all target syscalls is unnecessary when fuzzing OFL since we know the target syscall will only be invoked once by the forwarding interface. We do not use strace to intercept target syscalls because the untrusted handler will also invoke syscalls, while the ones forwarded by the OFL are only a subset. e.g., when doing initialization, logging and sending syscall output back to the enclave, the untrusted handler would also need to rely on syscalls. To be able to use strace, we need to modify the untrusted handler code in a way to tell strace which syscalls should be fuzzed, which might be redundant work as we are already modifying the code of the untrusted handler. Therefore, we decide to fuzz the return values directly in the untrusted handler. Since both the untrusted handler and the OS are under the attacker's control in the threat model, altering the handler's syscall return values before sending them to the trusted part is a valid fuzzing method.

Two vulnerabilities discovered, reported and fixed: We find two memory corruption vulnerabilities according to the `getsockopt` and `recvmsg` syscalls. Asylo uses serialized data to transfer syscall parameters and return values. Most of the syscalls have a pre-defined forwarding rule and are forwarded together (all handled by a set of functions instead of handled separately). For example, the forwarding rule `SYSCALL_DEFINE3(read, unsigned int, fd, \out void * [bound:count], buf, size_t, count)` means that the `buf` parameter is an output of the `read` syscall and its size is bounded by `count` parameter. So the forwarding function will only copy `count` bytes from `buf` to the application. Asylo also handles some syscalls specifically. In `enc_untrusted_getsockopt`, it copies data to the internal `optval` with the length of `opt_received`:

```
memcpy(optval, opt_received.data(), opt_received.size());
```

If the untrusted handler sets the `opt_received` with a size larger than the original size (the original `optlen`) which is used to allocate `optval` inside the enclave, the attacker could overflow `optval`. A similar vulnerability is found in `enc_untrusted_recvmsg` when it tries to copy `msg`→ `msg_name`. In this case, the same could happen to `msg.msg_namelen`. Following the practice of responsible disclosure, we contact Google and these two vulnerabilities are confirmed. They have been patched by comparing the received size with the input parameter `optlen` and `msg_namelen`.

Implications: We have shown that Emilia and the idea of fuzzing can find vulnerabilities in commercial-grade OFLs as well. The vulnerabilities in the application are less severe because of the assumption that the OFL will truncate the returned buffer based on the input max length. For example, if the application does this:

```
char opts[200];
char buf[200];
optlen = 200;
getsockopt(fd, level, name, opts, &optlen);
memcpy(buf, opts, optlen);
```

`buf` might be overflowed. However, since the OFL will truncate the returned `opts` to 200 bytes, the attacker can not control the content he writes beyond 200 bytes. But in Asylo, `opts` is serialized, and Asylo uses `opts.size (optlen)` instead of 200 to perform the truncation and copy (`memcpy(buf, opts.content, opts.size)`, the overflow happens before/during truncation, then the attacker can control the overwritten content.

Chapter 7

Evaluation

We have discussed our strategies for achieving syscall coverage in section 4.6. To evaluate the design decisions we make and understand the trade-off between syscall coverage and runtime, we perform two experiments. The first one compares stateless and stateful methods of handling new syscalls that occurred in new paths caused by fuzzing. Fuzz-all method mutates all encountered syscalls statelessly, and the stateful method can record the relationship between the fuzzed syscall and the newly occurred one. The second experiment is performed to evaluate the effectiveness of valid value sets.

Metrics: We used runtime, number of unique syscall invocations (with call stack) fuzzed, and number of unique core dumps produced, as the basis for comparison. Unique core dumps are core dumps with unique call stacks. This number does not equal to number of vulnerabilities we listed in Table 6.2. As we explained in Section 5.1, multiple core dumps with different call stack could represent the same vulnerability, and those core dumps also include common vulnerabilities in glibc. Manual analysis on dumped cores is needed to categorize them further. *#syscall(vanilla)* in Table 7.1 represents the number of unique syscalls in the vanilla run. This number is the same across all design settings. We run the experiments 3 times for each setting since randomness is involved in these experiments and the results are average values over 3 times. For the fuzz-all method, the runtime is calculated by turning off the stack trace since the stateless fuzz-all does not need it to identify syscall invocations. We also run 3 extra times for fuzz-all with stack trace turned on to collect the unique syscalls fuzzed by this method.

Experimental Setup: We test with 5 applications from our analyzed application list in section 6.1: OpenSSH, Lighttpd, Memcached, Redis and Curl. All experiments run on a machine equipped with 8 Intel 2.20GHz Xeon cores and 4GB RAM. The software environment is Ubuntu-18.04 with glibc-2.27.

7.1 Comparing Target Selection Methods

Table 7.1 shows the result of comparing two methods of handling new syscall invocations. *Fuzz-all(rnd)* statelessly fuzzes all syscalls encountered, and the stateful method feeds the interceptor with syscall references list recursively. Since stateless fuzzing can not identify unique syscall invocations, it can not iterate through every value of every return field. To make the two methods comprable, the fuzzer mutates all return fields with random values for every syscall in both *Fuzz-all(rnd)* and *Stateful(rnd)* settings.

Application	Setting	Runtime	#Syscall (Vanilla)	#Syscall (Fuzzed)	#Core dumps
Openssh	Stateful (rnd)	01:41:40	389	1098	23
	Fuzz-all (rnd)	01:23:12		820	20
Lighttpd	Stateful (rnd)	00:50:48	194	742	21
	Fuzz-all (rnd)	00:22:38		497	19
Memcached	Stateful (rnd)	00:53:25	152	434	37
	Fuzz-all (rnd)	00:42:36		315	5
Redis	Stateful (rnd)	00:32:36	94	537	11
	Fuzz-all (rnd)	00:11:53		348	8
Curl	Stateful (rnd)	00:05:31	59	107	6
	Fuzz-all (rnd)	00:04:02		94	4

Table 7.1: Stateful fuzz vs. Fuzz-all

Application	Setting	Runtime	#Syscall (Fuzzed)	#Core dumps
OpenSSH	Stateful (inv+rnd+valid)	14:19:45	1736	29
	Stateful (inv+rnd)	02:28:58	1173	24
Lighttpd	Stateful (inv+rnd+valid)	05:07:05	1077	23
	Stateful (inv+rnd)	01:04:06	784	21
Memcached	Stateful (inv+rnd+valid)	03:17:38	626	61
	Stateful (inv+rnd)	01:14:29	457	47
Redis	Stateful (inv+rnd+valid)	02:54:20	823	19
	Stateful (inv+rnd)	01:06:52	598	17
Curl	Stateful (inv+rnd+valid)	00:22:22	117	7
	Stateful (inv+rnd)	00:08:28	107	6

Table 7.2: Effects of the valid value set

The data in Table 7.1 shows that the stateful method always covers more syscall invocations and produces more core dumps than the stateless fuzz-all method. This is expected since Stateful has a higher chance of fuzzing new syscall invocations multiple times by replaying the mutation of previously fuzzed syscalls. In contrast, Fuzz-all depends on randomness to reach the new syscall invocations (and very likely only once for each). Also, because of its ability to distinguish syscall invocations, Stateful fuzz can be extended to become more systematic by iterating through different values and fields as *Statefull(inv+rnd+valid)* in Table 7.2 shows.

Furthermore, the Fuzz-all method is unable to trigger some crashes by design, as mentioned in section 4.6.1. This is the reason why the number of core dumps drops a lot for *Fuzz-all(rnd)* in Memcached. It fails to trigger the memory corruption in `fprintf`, which forms the majority of core dumps in *Stateful(rnd)* with different call stacks. Fuzz-all actually fuzzed the vulnerable `write` syscall in `fprintf`. However, there is another extra `write` syscall before the misuse of the return value of the previous `write`. If the extra `write` does not succeed (returned size equals the input size), the misuse by the vulnerable instruction will not be reached. As Fuzz-all fuzzes both the vulnerable and the extra `write`s, the possibility of making the extra `write` success is very low during fuzzing.

Obviously, the stateful method is slower than the stateless Fuzz-all because the stateful method analyzes more syscall invocations, and the overhead of stack trace is not negligible. However, the runtime of *Fuzz-all(rnd)* and *Stateful(rnd)* are close for OpenSSH and Memcached. That is because Fuzz-all would more likely enter an infinite retry loop and timeout by design. As mentioned in section 4.6.1, Stateful only fuzzes the first 10 occurrences of the target syscall invocation to prevent infinite retry. Fuzz-all can not do the same because it can not identify unique syscall invocations. Also, Stateful will only face this situation when the retrying syscall invocation is in the current reference list, while Fuzz-all will fuzz the retry syscall as long as it is reachable. For example, assume S1, S2, S3, S4, and S5 is a syscall sequence in the vanilla run. None of them except S4 affect the execution path. S4 resides in an infinite retry loop unless the return value is zero and the default return value of a normal OS is zero. When fuzzing the first 3 syscalls, the stateful fuzz method will only fuzz the target syscall invocation and end the iteration smoothly. However, Fuzz-all will fuzz all the syscalls including S4 when `skip_count` < 4. So Fuzz-all will always enter the infinite retry and time out until `skip_count` equals 4.

7.2 Effects of Valid Values

We perform the second experiment to analyze the effectiveness of valid values generated from the value extractor. In Table 7.2, both *Stateful(inv+rnd+valid)* and *Stateful(inv+rnd)* uses the stateful method to handle new syscalls, and the fuzzer iteratively tries out every value in the value set for every return field of the target syscall. For *Stateful(inv+rnd+valid)*, the value set of each return field contains valid values from the value extractor, invalid values (MIN and MAX) and 3 random values. For *Stateful(inv+rnd)*, the value set only includes invalid values and 3 random values.

The result shows that the valid value set could lead to more syscall coverage and core dumps. For example, in OpenSSH, a `poll` syscall will never be reached unless the previous `read` syscall returns `EAGAIN` or `EWOULDBLOCK`. The `EAGAIN` in the valid set of `read`'s return value will help trigger this syscall.

The runtime also increases dramatically when using valid values because there are more syscall invocations to fuzz and more values to try out for each syscall invocation. It seems not efficient since the number of fuzzed syscall invocations and the number of core dumps are not increased in the same order of magnitude. This is because not all valid set values are useful for finding new syscall invocations due to our coarse-grained static value extractor. We do not relate the syscalls in static analysis with unique syscall invocations in stateful fuzzing. We try the valid values for all the syscall invocations with the same syscall number, although the valid value only applies to some of them. It is worse when we can not relate an `errno` with a specific syscall number. We will add the value to the valid set of all syscalls. This can be improved by having a fine-grained value extractor and labelling each syscall invocation only to use the valid value if the label matches during fuzzing.

In summary, the experiments demonstrate, to a certain extent, that the design decisions we made on target selection and fuzzing value sets indeed help increase syscall coverage and produce more core dumps.

Chapter 8

Conclusion and Future Work

In an effort to detect memory corruption Iago attacks in legacy applications, we explore the technique of reverse syscall fuzzing, which passively waits for syscall invocations and replace their return values. We choose to fuzz legacy applications (i.e., applications developed prior to the advent of mechanisms protecting them from an untrusted OS) with the observation that such legacy code is still largely reused or ported as opposed to rewritten.

We study the difficulties of fuzzing syscall return values. One of the most challenging problems is that the syscall sequence would change during fuzzing so that we do not have fixed targets to fuzz. We design Emilia to fuzz newly appeared syscalls recursively by replaying the fuzzing values in previous iterations. We also build a valid value extractor to find values that can help explore different application branches. Experiments have been performed to justify the effectiveness of our design.

We have detected 50 Iago vulnerabilities in 17 applications using Emilia. We further study the cause of the detected vulnerabilities by categorizing the semantics of syscall return values into static, local, stateful and external.

The majority of the detected vulnerabilities pertaining to static and local categories could be easily fixed by stateless verification in the OFL. The remaining vulnerabilities may need code modification or extra resources. We hope Emilia can shed some light on how Iago vulnerabilities can be mitigated when legacy applications are ported to the isolated environment of the protection mechanisms.

8.1 Future Work

Modifying syscall return values along has less impact on the path coverage than mutating application's regular inputs such as environment variables and configurations. We let the users provide fixed inputs to analyze paths they are interested in for now. Ideally, we could use fuzzing or other techniques to generate regular inputs that can trigger different paths automatically.

Instead of utilizing stack trace to identify unique syscalls, we could use the fine-grained execution path leading to the syscall. The path can be extracted by instrumenting the application and all its libraries to track branches taken in the run time. By doing so, the performance of Emilia would also be improved since computing the stack trace involving syscalls to access tracee's stack and load images.

The current implementation of Emilia is synchronous. We could run multiple fuzzing tasks in parallel to improve the performance further.

The valid value extractor could be optimized to generate values for a specific syscall invocation. Currently, if the return value of one `read` syscall is used on a branch condition, the extracted valid value will be added to all `read`'s value set. Trying this value on other `read` syscalls usually not trigger any new paths, and iterations are wasted. We could mark each syscall invocation with an identifier in the application and only use valid values on the syscall invocation where the value is generated from.

Bibliography

[1] ARM Ltd. *Security technology building a secure system using trustzone technology (white paper)*, 2009.

[2] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O'keeffe, Mark L Stillwell, et al. SCONE: Secure linux containers with intel SGX. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 689–703, 2016.

[3] Andrew Baumann, Marcus Peinado, and Galen Hunt. Shielding applications from an untrusted cloud with haven. *ACM Transactions on Computer Systems (TOCS)*, 33(3):1–26, 2015.

[4] Stephen Checkoway and Hovav Shacham. Iago attacks: Why the system call API is a bad untrusted RPC interface. *SIGPLAN Not.*, 48(4):253–264, March 2013. Available at http://doi.acm.org/10.1145/2499368.2451145 [Accessed July 30, 2020].

[5] Xiaoxin Chen, Tal Garfinkel, E. Christopher Lewis, Pratap Subrahmanyam, Carl A. Waldspurger, Dan Boneh, Jeffrey Dwoskin, and Dan R.K. Ports. Overshadow: A virtualization-based approach to retrofitting protection in commodity operating systems. *SIGPLAN Not.*, 43(3):2–13, March 2008.

[6] Yueqiang Cheng, Xuhua Ding, and Robert Deng. Appshield: Protecting applications against untrusted operating system. *Singaport Management University Technical Report, SMU-SIS-13*, 101, 2013.

[7] Intel Corporation. *Intel Processor Trace.* Available at https://software.intel.com/content/www/us/en/develop/blogs/processor-tracing.html [Accessed July 30, 2020].

[8] John Criswell, Nathan Dautenhahn, and Vikram Adve. Virtual ghost: Protecting applications from hostile operating systems. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, pages 81–96, New York, NY, USA, 2014. ACM. Available at http://doi.acm.org/10.1145/2541940.2541986 [Accessed July 30, 2020].

[9] Joeri de Ruiter and Erik Poll. Protocol state fuzzing of TLS implementations. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 193–206, Washington, D.C., August 2015.

[10] Google. *Asylo.* Available at https://asylo.dev [Accessed July 30, 2020].

[11] James Greene. *Intel trusted execution technology, white paper.* Intel, 2012.

[12] Large-Scale Data & Systems (LSDS) Group. *TaLoS*, 2019. Available at https://github.com/lsds/TaLoS [Accessed July 30, 2020].

[13] Le Guan, Peng Liu, Xinyu Xing, Xinyang Ge, Shengzhi Zhang, Meng Yu, and Trent Jaeger. Trustshadow: Secure execution of unmodified applications with arm trustzone. In *Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services*, pages 488–501, 2017.

[14] Owen S. Hofmann, Sangman Kim, Alan M. Dunn, Michael Z. Lee, and Emmett Witchel. Inktag: Secure applications on an untrusted operating system. *SIGPLAN Not.*, 48(4):265–278, March 2013. Available at http://doi.acm.org/10.1145/2499368.2451146 [Accessed July 30, 2020].

[15] Hong Hu, Zheng Leong Chua, Zhenkai Liang, and Prateek Saxena. Identifying arbitrary memory access vulnerabilities in privilege-separated software. In Günther Pernul, Peter Y A Ryan, and Edgar Weippl, editors, *Computer Security – ESORICS 2015*, pages 312–331, Cham, 2015. Springer International Publishing.

[16] Tyler Hunt, Zhiting Zhu, Yuanzhong Xu, Simon Peter, and Emmett Witchel. Ryoan: A distributed sandbox for untrusted computation on secret data. *ACM Transactions on Computer Systems (TOCS)*, 35(4):1–32, 2018.

[17] Intel. *Intel Software Guard Extensions SDK for Linux OS: Developer Reference*, 2016.

[18] Prerit Jain, Soham Jayesh Desai, Ming-Wei Shih, Taesoo Kim, Seong Min Kim, Jae-Hyuk Lee, Changho Choi, Youjung Shin, Brent Byunghoon Kang, and Dongsu Han. OpenSGX: An open platform for SGX research. In *NDSS*, 2016.

[19] Dave Jones. *Trinity: Linux system call fuzzer*. Available at https://github.com/kernelslacker/trinity [Accessed July 30, 2020].

[20] Mustakimur Rahman Khandaker, Yueqiang Cheng, Zhi Wang, and Tao 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, page 971–985, New York, NY, USA, 2020. Association for Computing Machinery. Available at https://doi.org/10.1145/3373376.3378486 [Accessed July 30, 2020].

[21] Seongmin Kim, Juhyeng Han, Jaehyeong Ha, Taesoo Kim, and Dongsu Han. SGX-Tor: A secure and practical tor anonymity network with SGX enclaves. *IEEE/ACM Transactions on Networking*, 26(5):2174–2187, 2018.

[22] Jan Kneschke. *Lighttpd*, 2003. Available at https://www.lighttpd.net/ [Accessed July 30, 2020].

[23] Youngjin Kwon, Alan M Dunn, Michael Z Lee, Owen S Hofmann, Yuanzhong Xu, and Emmett Witchel. Sego: Pervasive trusted metadata for efficiently verified untrusted system services. *ACM SIGARCH Computer Architecture News*, 44(2):277–290, 2016.

[24] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, pages 75–86. IEEE, 2004.

[25] Yanlin Li, Jonathan McCune, James Newsome, Adrian Perrig, Brandon Baker, and Will Drewry. Minibox: A two-way sandbox for x86 native code. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 409–420, 2014.

[26] Joshua Lind, Christian Priebe, Divya Muthukumaran, Dan O'Keeffe, Pierre-Louis Aublin, Florian Kelbert, Tobias Reiher, David Goltzsche, David Eyers, Rüdiger Kapitza, et al. Glamdring: Automatic application partitioning for Intel SGX. In *USENIX Annual Technical Conference (USENIX ATC 17)*, pages 285–298, 2017.

[27] Yutao Liu, Tianyu Zhou, Kexin Chen, Haibo Chen, and Yubin Xia. Thwarting memory disclosure with efficient hypervisor-enforced intra-domain isolation. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 1607–1619, 2015.

[28] V. J. M. Manès, H. Han, C. Han, S. K. Cha, M. Egele, E. J. Schwartz, and M. Woo. The art, science, and engineering of fuzzing: A survey. *IEEE Transactions on Software Engineering*, pages 1–1, 2019.

[29] Yerzhan Mazhkenov. *SGX-SQLite*, 2019. Available at https://github.com/yerzhan7/SGX_SQLite [Accessed July 30, 2020].

[30] Dan RK Ports and Tal Garfinkel. Towards application security on untrusted operating systems. In *HotSec*, 2008.

[31] Christian Priebe, Divya Muthukumaran, Joshua Lind, Huanzhou Zhu, Shujie Cui, Vasily A. Sartakov, and Peter Pietzuch. SGX-LKL: Securing the host os interface for trusted execution, 2019.

[32] Shweta Shinde, Dat Le Tien, Shruti Tople, and Prateek Saxena. Panoply: Low-TCB linux applications with SGX enclaves. In *NDSS*, 2017.

[33] strace. *strace: linux syscall tracer*. Available at https://strace.io/ [Accessed July 30, 2020].

[34] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. Sok: Eternal war in memory. In *2013 IEEE Symposium on Security and Privacy*, pages 48–62. IEEE, 2013.

[35] Richard Ta-Min, Lionel Litty, and David Lie. Splitting interfaces: Making trust between applications and operating systems configurable. pages 279–292, November 2006.

[36] Bohdan Trach, Alfred Krohmer, Franz Gregor, Sergei Arnautov, Pramod Bhatotia, and Christof Fetzer. Shieldbox: Secure middleboxes using shielded execution. In *Proceedings of the Symposium on SDN Research*, pages 1–14, 2018.

[37] Chia-Che Tsai, Donald E Porter, and Mona Vij. Graphene-SGX: A practical library OS for unmodified applications on SGX. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 645–658, 2017.

[38] Jo Van Bulck, David Oswald, Eduard Marin, Abdulla Aldoseri, Flavio D Garcia, and Frank Piessens. A tale of two worlds: Assessing the vulnerability of enclave shielding runtimes. In *Proceedings of the 2019 ACM Conference on Computer and Communications Security*, pages 1741–1758, 2019.

[39] Veronica Velciu, Florin Stancu, and Mihai Chiroiu. Hiddenapp-securing linux applications using ARM TrustZone. In *International Conference on Security for Information Technology and Communications*, pages 41–52. Springer, 2018.

[40] Huibo Wang, Erick Bauman, Vishal Karande, Zhiqiang Lin, Yueqiang Cheng, and Yinqian Zhang. Running language interpreters inside SGX: A lightweight, legacy-compatible script code hardening approach. In *Proceedings of the ACM Asia Conference on Computer and Communications Security*, pages 114–121, 2019.

[41] Bin Cedric Xing, Mark Shanahan, and Rebekah Leslie-Hurd. Intel® software guard extensions (intel® sgx) software support for dynamic memory allocation inside an enclave. In *Proceedings of the Hardware and Architectural Support for Security and Privacy 2016*, pages 1–9. 2016.