Emilia: Catching Iago in Legacy Code

Rongzhen Cui
University of Toronto
gavin.cui@mail.utoronto.ca

Lianying Zhao
Carleton University
lianying.zhao@carleton.ca

David Lie
University of Toronto
lie@eecg.toronto.edu

***Abstract*—There has been interest in mechanisms that enable the secure use of legacy code to implement trusted code in a Trusted Execution Environment (TEE), 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 observe that not all legacy code is vulnerable to Iago attacks and that legacy code must use return values from system calls in an unsafe way to have Iago vulnerabilities.**

Based on this observation, we develop Emilia, which automatically detects Iago vulnerabilities in legacy applications by fuzzing applications using system call return values. We use Emilia to discover 51 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 (82.4%) 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.

I. Introduction

To protect security-sensitive code from the large trusted computing base (TCB) of commodity systems, many hardware-based [15], [10], [1] and hypervisor-based [6], [5], [38] trusted execution environments (TEEs) have been proposed. User-level TEEs, such as SGX, isolate "trusted" applications from the large, legacy TCB of commodity systems, which includes the OS, drivers as well as all privileged applications on a system. In general, there are two ways these trusted applications can be implemented—they can either be implemented from scratch, or one can port a legacy application that runs on a normal OS into the TEE. While the former is more secure, it also requires more effort. In contrast, the latter requires less effort, but faces a significant drawback—legacy code assumes the presence of an operating system (OS), but the TEE isolates the trusted application from the OS. To service system calls (abbreviated to *syscall* hereinafter) to the OS, legacy applications running in TEEs may use an OS-forwarding layer (OFL) that intercepts syscalls made by the legacy application and forwards them to an untrusted OS running outside the TEE. Unfortunately, while the TEE provides isolation for the application's runtime state and memory, this syscall interface still represents a significant attack surface for the trusted application.

Network and Distributed Systems Security (NDSS) Symposium 2021
21-24 February 2021
ISBN 1-891562-66-5
https://dx.doi.org/10.14722/ndss.2021.24328
www.ndss-symposium.org

Legacy code inherently trusts the OS that it makes syscalls to. However, if the legacy code is inside a TEE, and the commodity OS is outside the TEE, this makes the commodity OS *untrusted*, which raises the possibility of the OS executing an *Iago* attack. An Iago attack is one where the untrusted OS abuses the trust the legacy application places in the syscall interface to return maliciously crafted syscall return values. Such attacks were first identified in [5] and [32], 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.

Many trusted applications use a custom OFL to enable legacy code to run in TEEs such as SGX by forwarding syscalls to the untrusted OS, as exemplified by TaLoS [26], SGX_SQLite [31] and Intel-SGX-SSL [17]. In addition, legacy code can also be ported to work with general-purpose TEE-secured containers with OFL, e.g., TensorSCONE [24] integrates TensorFlow with SCONE [2] (Intel SGX) to enable secure execution of machine learning computations, or TEE-secured language interpreters like ScriptShield [43]. A number of OFLs try to mitigate Iago vulnerabilities by performing syscall return value sanitization, as exemplified by Panoply[35] and Glamdring[28], or by narrowing the syscall interface as proposed in Graphene-SGX[40] and SCONE[2]. However, such sanitization is often ad-hoc and may be incomplete, leaving applications still vulnerable to Iago attacks.

In this paper, we 1) attempt to measure the base rate of Iago vulnerabilities in a wide range of legacy code; and 2) analyze how well a sample of current OFLs can shield legacy code from Iago vulnerabilities. To do this, we design and implement Emilia[1], a syscall fuzzer that finds Iago vulnerabilities in legacy code. Compared to regular syscall fuzzers, Emilia fuzzes applications from the syscall interface, by replacing legitimate OS syscall return values with fuzz values, designed to find and trigger Iago vulnerabilities. Previous work [13] has used binary-level symbolic execution and taint-tracking to detect unsafe information flows from syscalls to sensitive uses, but their approach suffers from path explosion, whereas fuzzing does not. We run Emilia on 17 popular applications and find a total of 51 Iago vulnerabilities, which we categorize into 5 basic types. We also found two Iago vulnerabilities in Google's Asylo [9] system, an OFL that has been specially designed to protect applications against Iago vulnerabilities. Both

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

vulnerabilities have been confirmed and fixed by the Asylo team. Our main result is that Iago vulnerabilities are widespread—nearly every application we examined had at least one vulnerability. Moreover, we find that many vulnerabilities are stateless, which should be possible for OFLs to mitigate efficiently, but a number of current OFLs and SGX applications actually only partially mitigate these vulnerabilities or do not mitigate them at all.

In summary, this paper makes the following contributions:

- We present Emilia, a tool that finds and detects Iago vulnerabilities by fuzzing syscall return values.
- We use Emilia to measure the frequency of the Iago vulnerabilities in real-world applications, and have identified a total of 51 vulnerabilities involving memory corruption in 17 popular legacy applications and glibc. We find that while Iago vulnerabilities are widespread, the vast majority are easily-mitigatable Local or Static vulnerabilities.
- We perform an analysis of 6 OFLs and SGX applications and find that the majority neglect to mitigate some or all Iago vulnerabilities. Only one, Graphene-SGX, catches all Local and Stateful Iago vulnerabilities. While Emilia is intended to fuzz unmodified legacy applications, we ported it to fuzz Google Asylo [9] and discovered two Iago vulnerabilities which have been reported and fixed.
- 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.

We begin by providing background and describing our Iago vulnerability model in Section II. We then describe the design and implementation of our Iago fuzzer, Emilia in Section III, followed by an analysis of the Iago vulnerabilities found in a corpus of legacy applications in Section IV, where we categorize the vulnerabilities into Static, Local, Stateful, External and Channel. We analyze the security and Iago mitigation ability of current OFLs in Section V and then evaluate the different Emilia fuzzing strategies in Section VI. Finally, we discuss limitations, related work and conclude in Sections VII, VIII and IX.

II. Background and Vulnerability Model

For legacy applications running in a TEE with an OFL, the syscall interface represents the attack surface for a malicious OS. A malicious OS kernel can cause unexpected and undesirable application behavior by generating unexpected or illegal inputs to applications. Applications generally interact with the OS kernel via the syscall interface. A benign OS adheres to a well-known set of behavior when generating responses for sysalls, which may be specified in documentation such as that found in a syscall's "man pages". However, in this work, we model a malicious OS that is free to arbitrarily deviate from the specification and return any values it wishes in its responses.

The possibility of syscall manipulation by a malicious OS was identified by Ports and Garfinkel in their Overshadow system [32]. However, it was not until the analysis performed by Checkoway and Shacham that these attacks were formally named Iago attacks [4]. Their proposed Iago attacks consisted of only scalar syscall return values, and two attacks were identified in their work: one caused replay attacks on Apache servers with mod_ssl, due to the syscall `getpid` being used in part for randomness; and the other one even achieved arbitrary code execution because `malloc` (wrapped in `glibc`) could be tricked to modify arbitrary memory by malicious return values of `brk` and `mmap`. However, a malicious OS is free to return both corrupted scalar and buffer values. For example, in the syscall `getsockopt(int sockfd, int level, int optname, void *optval, socklen_t *optlen)` the OS may fill `optval` with arbitrary values. We do note that in general, the contents of such buffers are copied from the OS' address space into SGX enclave memory by the OFL, which will only copy as much data as it has allocated space for. Thus, while a malicious OS can return a longer buffer, the entire buffer may not necessarily be passed to the trusted application depending on the implementation of the OFL. For example, in the `getsockopt` syscall, a proper OFL will only copy at most the number of bytes specified by the application in the `optlen` argument. We exclude arguments that could be set by a malicious party outside of the OS from the Iago attack surface. For example, the value returned in the `buf` argument of the `read(int fd, void *buf, size_t count)` syscall could have been set by a malicious attacker with access to the file being read from. We restrict Iago attacks to only modify values that are exclusively under the control of the OS.

Iago attacks require an exploitable Iago vulnerability to be successful. We define an Iago vulnerability in an application as a section of code in the application that uses a syscall return value in an unsafe way, which leads to unexpected or undesirable behavior in the application. These vulnerabilities occur because legacy applications inherently trust the OS and thus do not perform validation on the syscall values returned by the OS.

In this paper, we focus on Iago vulnerabilities that can result in pointer corruption. While malicious syscall return values can be used in a variety of unsafe ways, e.g., the return value of `getpid` can be used as an entropy source, or the time provided by the untrusted OS can be relied on as timestamps to generate system logs, the most egregious misuse of untrusted data is when it can result in code or data pointer corruption [37]. Data pointer corruption can lead to memory safety errors, which can lead to information leakage or further memory corruption. Code pointer corruption can result in arbitrary code execution. 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 later written to a file, network socket or side-channel.

III. Design and Implementation of Emilia

A. Objective

Since Iago vulnerabilities result from the misuse or improper handling of syscall-returned results, Emilia's objective is to trigger as many syscalls and as much code that executes after a syscall as possible to search for vulnerabilities. When

```
1 /* after fuzzing the return value of fstat with
     value other than 0, new write() syscall will
     be triggered */
2 if (0 != fstat(fd, &st)) { // 0 for success
3   log_error_write("..."); // invoke write()
4   ...
5 }
```

Listing 1: An example of a new syscall invocation introduced by fuzzing

searching this code, we must take two aspects into account: 1) Static code locations. Naturally, we wish to find and execute as many syscall invocations as possible. 2) Context. Even the same static location may be executed under different contexts (i.e. different local/global variables values), which may lead to different arguments being passed to the syscall, as well as different code paths after the syscall.

While Emilia is technically a fuzzer, as it dynamically generates test cases in an effort to find vulnerabilities, it differs from standard fuzzers in two key aspects. First, standard fuzzers generally aim to maximize path coverage by fuzzing program inputs. In contrast, Emilia aims to maximize *syscall coverage*, which is to maximize both the number of static syscall invocations and the contexts under which they are executed by fuzzing syscall return values.

It is possible to increase syscall coverage by only mutating syscall return values and not program inputs. To see why, consider Listing 1, which is taken from Lighttpd [23]. If Emilia injects a non-zero return value for `fstat`, then `log_error_write` in the error handling path will be invoked and will subsequently invoke the `write` syscall, which does not occur in the vanilla execution of the program (i.e., the execution with no fuzzed return values). From this we see that fuzzing some syscall return values may cause new syscall invocations, which subsequently can also be iteratively fuzzed (and which may go on to cause more syscall invocations). Although mutating both program inputs and syscall return values is required to find as many Iago vulnerabilities as possible, our current implementation of Emilia focuses only on the latter—trying to maximize syscall coverage while covering only a single path—Emilia does not fuzz inputs, only syscall return values. We envision that Emilia can be combined with standard application fuzzers in a straightforward way—each input that triggers a new path can be given to Emilia as a starting point for achieving syscall coverage. We leave the exploration of this for future work.

The other key difference compared to standard fuzzers is that as opposed to generating and invoking an application with the fuzz inputs, Emilia responds to syscall invocations from the application with fuzz return values. As a result, Emilia is necessarily *passive*, i.e., the fuzzer can only respond to syscalls that the application has made, because the goal is to fuzz the application from the point of view of a malicious kernel. This is the reverse of kernel fuzzers [19], [30], which attempt to fuzz the kernel from the point of view of a malicious application. Emilia bears some similarity with network protocol fuzzing [8] in that it sends fuzz inputs inside responses to requests.

Measuring syscall coverage: Since Emilia's objective is to attain syscall coverage, we first define here, how we intend to measure syscall coverage. Simply counting static invocation of syscalls is insufficient as it doesn't take into account the path or context leading up to and following the syscall—syscalls are often located in libraries (such as `libc` and whose functions may have many incoming and outgoing code paths). Moreover, since Iago vulnerabilities are necessarily a result of unsafe syscall result handling after the syscall invocation, it is important that our measurement take into account different code paths after a syscall invocation.

To effectively identify all execution paths leading to and following from syscalls, one way is to directly collect the application's control flow (e.g., conditional/unconditional direct/indirect branches). One can envision employing application tracing, using efficient hardware such as Intel Processor Trace (PT) [16], which collects such information that can be later retrieved in the form of data packets. An alternative that does not require specialized hardware is to instrument the application. However, because many applications invoke syscalls via libraries, this would necessitate instrumenting not only the application but all libraries as well.

In light of these drawbacks for hardware-tracing or instrumentation, we observe that a proxy for the path after a syscall is the path leading up to a syscall, which can be approximated by the call path (i.e., functions in the call stack) leading up to the syscall—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 at the point the syscall is invoked, and measure syscall coverage as the number of unique syscall invocations that are executed.

B. Design of Emilia

An overview of Emilia's architecture is shown in Figure 1. Emilia consists of Interceptor, Controller and Value Extractor, with the application's source code, binary and optionally a client binary as inputs. Core dumps generated from crashes are the output.

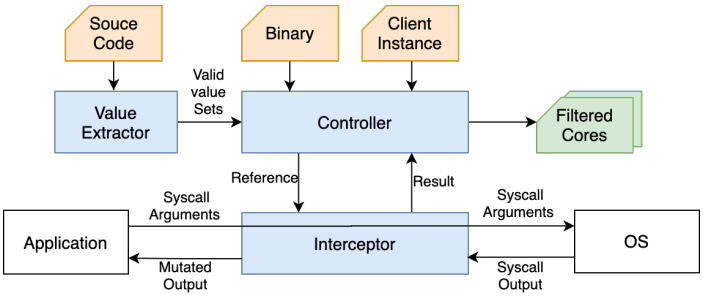


Figure 1: Components and workflow of Emilia

Interceptor: Rather than replacing the kernel, Emilia intercepts system calls made between the application being fuzzed and a standard OS kernel. Interception is performed

using an interceptor adapted from strace [36] to capture and handle the syscalls made by the application. We use strace to intercept syscalls instead of a libc wrapper to intercept libc calls to enable Emilia to also detect Iago vulnerabilities in libc implementations. Isolation techniques such as Graphene-SGX [40] and SCONE [2] place the C library inside the trusted world, and in many cases, this code is ported from a legacy C library to maximize compatibility and minimize engineering effort.

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. For example, `int getsockopt(int sockfd, int level, int optname,void *optval, socklen_t *optlen)` has three return fields: the return value (`ret`), `optval` and `optlen`. The sizes of `ret` and `optlen` are `sizeof(int)` and `sizeof(socklen_t)`. For `optval`, the maximum length is given by the original `optlen` passed to the syscall. By filling the buffer with the known max length, strace would not directly overflow the buffer during writeback if the application properly allocates the pointer and provides a correct length argument. The interceptor would also calculate a hash of stack trace for each syscall encountered to help identify them for syscall coverage. The stack trace is produced by `libunwind` embedded in `strace`.

Controller: The entire fuzzing process is coordinated by a python script, which invokes and feeds the other components with instructions. We define one *iteration* as the cycle from when the controller starts the application for fuzzing to when it crashes due to fuzzing or terminates normally.

The role of the controller is threefold: 1) Target selection. The controller regulates the fuzzing loop and selects the syscalls to be fuzzed, called *targets*, for the interceptor. The return values of syscalls that are not targets are passed on without modification. 2) Satisfying external conditions. Sometimes, the application may have external dependencies for continuous 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 deduplication and filter out the ones that are not caused by memory corruptions (e.g., assertion error).

Value extractor: The fuzz values used by Emilia can also affect syscall coverage. Similar to the approach by Shastry et al. [34], we also perform a coarse-grained static analysis on the application's source code to generate values that may help increase syscall coverage, which is done by the value extractor. The value extractor extracts constant values against which syscall return values are compared in branch conditions and adds values that will exercise both sides of the branch to the valid value set of corresponding syscall's output (see Section III-C2). For example, if we find `if (ret < 10)` and `ret` is the return value of `read`, we will add both 10 and 9 to the valid value set for `read`. We call them valid values because the application assumes they are possible return values of the syscall and may take different actions based on them. The extractor also checks the usage of `errno` in the application since it is likely set by libc based on the negative part of the original syscall return values. If we can not associate an `errno` with a specific syscall, we add this value to the valid value set for all syscalls.

The output of Emilia is core dumps with unique call stacks after deduplication by the controller. However, further manual analysis is still needed to understand the exact cause of each crash. Note that multiple core dumps with different call stacks could be caused by the same Iago vulnerability. For example, we detected a crash in `fprintf` in glibc when fuzzing the `write` syscall. The application could call `fprintf` in different locations with different call stacks, but they are essentially the same Iago bug. Also, we could not simply look at the exact crash location (last frame of the call stack). For example, a buffer overflow caused by `memcpy` would always dump a core with the last frame in glibc. But the `memcpy` could be used differently in different parts of the application. So we resort to manual analysis to determine the actual Iago bugs.

For Emilia to work, the user should first provide the source code of the application to Emilia's value extractor, which extracts the valid value set. The user should then provide the application's binary, a set of command-line arguments and the valid value set to Emilia, which then fuzzes the binary and produces a set of deduplicated core dumps and corresponding fuzzing logs. Finally, the user needs to analyze the core dumps and logs to determine the root cause of the crash. In the case where the application is a server that needs a client to send requests for it to continue execution, Emilia needs both a client to send the request and a signal to indicate when the client should be invoked. This is done by running the interceptor provided by Emilia on the server, which exports a list of unique *syscall+stackhash* pairs, of which the user must identify the one that indicates the server is ready for the client request. Most often, this is an accept call which causes the server to block or wait for a client request.

C. Fuzzing Strategies

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

1) Target Selection: During execution, an application may make a number of syscalls. Rather than fuzz all of them, Emilia selectively fuzzes a subset to elicit different responses from the application. We refer to the set of invocations to be fuzzed as targets. Syscalls that are not selected as targets, and hence are not fuzzed, are still executed and the OS-provided return values are passed directly to the application without modification by Emilia.

There are several strategies Emilia could use in selecting which syscall invocations to target to maximize syscall cover-

age. We begin by defining the sequence of syscalls made by the application when no syscalls are targeted as the *vanilla* syscall sequence. A naïve strategy would be to target one syscall at a time (e.g., the first syscall in the vanilla syscall sequence, then the second, etc.). However, as demonstrated in Listing 1, fuzzing a syscall can modify the sequence of syscalls that the application makes subsequently. If Emilia only targets one syscall at a time, it would miss these newly discovered syscalls. Thus, maximizing syscall coverage requires strategies where Emilia targets more than one syscall at a time. As a result, Emilia uses two strategies to attain syscall coverage:

Fuzz-all: Instead of targeting one syscall at a time, an alternative is to target all syscalls encountered during execution. This will thus capture any new syscalls generated by fuzzed inputs. However, by doing this, 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 (i.e., not target) the first `skip_count` syscall invocations. The `skip_count` will be incremented by one on each iteration.

While this approach is simple, a disadvantage is that it cannot systematically target every syscall that the application can generate. For example, if Emilia targets s_1, which results in a new syscall s_2, Fuzz-all will cause Emilia to also target syscall s_2. If Emilia had only targeted s_1 and not targeted s_2, it is possible that a new syscall s_3, might have been found, but because Fuzz-all is stateless, it can only target all syscalls after `skip_count`. Fuzz-all can also fail to trigger vulnerabilities if there's an extra syscall between the vulnerable syscall invocation and the use of its return values. If the application terminates due to the extra syscall being fuzzed, the vulnerable syscall's return value will not have a chance to be used in an unsafe way. To address these shortcomings, Emilia also supports a more complex, Stateful fuzzing strategy.

Stateful: Emilia can target syscall invocations statefully and recursively. When a new syscall invocation is identified, Emilia saves the current fuzzing state (syscalls already fuzzed and which fields with what values) and targets the newly found syscall invocations. Once the iteration is finished, it will go back and continue from the saved state. Algorithm 1 shows the pseudo-code of the fuzzing loop.

In every iteration, the controller provides the interceptor with a reference list (`references`). Each element of the list contains a target syscall name, syscall stack hash and a value reference. The value reference describes which return field should be fuzzed and with what value for the target syscall invocation. The interceptor will fuzz all the syscall invocations with matching syscall name and stack hash in the reference list. A syscall invocation with a unique stack hash could be invoked multiple times in a loop. In some cases, an application may have a loop that will keep retrying a syscall until it succeeds. Because Stateful fuzzing tracks each syscall by its call stack, it is able to identify these and will terminate the fuzzing iteration if it seems a syscall invocation occur more than 10 times in one iteration.

The Stateful strategy first extracts a list of unique syscall invocations from the vanilla sequence and updates the `overall_syscall` set with `vanilla_syscalls`. Then for every syscall invocation in the `vanilla_syscalls`,

Algorithm 1 Stateful fuzzing loop

```
1:  overall_set ← ∅
2:  procedure MAIN_LOOP()
3:      vanilla_syscalls ← extract_vanilla_syscalls()
4:      overall_set.add(vanilla_syscalls)
5:      for syscall in vanilla_syscalls do
6:          target ← (syscall, hash, init_ref)
7:          references ← [target]
8:          recursive_fuzz(references, 0)

9:  procedure RECURSIVE_FUZZ(references, depth)
10:     if depth > max_depth then
11:         return
12:     current_target ← references[depth]
13:     do
14:         new_syscalls ← run_interceptor(references)
15:         overall_set.add(new_syscalls)
16:         for (syscall, hash) in new_syscalls do
17:             next_target ← (syscall, hash, init_ref)
18:             references.append(next_target)
19:             recursive_fuzz(references, depth + 1)
20:     while current_target.update_target()
```

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). We will describe the update mechanism in the Sections III-C2 and III-C3. `run_interceptor` will launch the interceptor with the reference list and connect the client if the target application is a server. If there exist new syscall invocations which are not found in the `overall_syscall` set, the controller will update `overall_syscall` set, append the new syscall invocations to the reference list and go to the next level of recursion. 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. As future work, we could record the whole execution state or take a snapshot of the execution.

Stateful fuzzing is still not comprehensive. Due to limited resources, we only fuzz syscall invocations with this "fuzz-then-appear" (syscalls in the reference list) relationship together. With more resources, it may be possible to also fuzz every possible combination of syscall invocations at the same time.

2) Fuzzing Value Sets: Once Emilia targets a syscall, it will need to modify the return values with fuzz values. There are three possibilities of how the fuzz values can be set: random values, invalid values and valid values. Random values are drawn randomly from the range defined by the type of return value (i.e., a random 32-bit value for a returned `long`). Invalid values are values known to be invalid for the returned value. Currently, this is set to the MAX and MIN values for the returned type. With more effort, one could also examine the semantics of each syscall and create syscall-specific sets

of invalid values, but currently Emilia does not do this. For example, many syscalls will only return a limited set of negative-valued error codes, so any negative value not in the set would be a potentially invalid value. Finally, valid values are those derived by the Emilia's value extractor, described in Section III-B. Both invalid and random values aim to trigger crashes if the syscall return value is used for pointer arithmetic. For random values, Emilia not only randomizes the bytes of the output but also the number of bytes to overwrite. In this way, Emilia has a higher possibility of generating values with different orders of magnitude. This helps detect more vulnerabilities because some memory corruptions can not be triggered with too large a 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.

3) Return Fields: A syscall could have multiple return fields. In which order should we fuzz them? For example, `stat` has 14 return fields when broken down (i.e., the return value + 13 fields in `struct stat`). A systematic, but expensive strategy would be to try every combination of the return fields and values. The number of combinations increases exponentially with the number of return fields. Assume we have num_values for each field to try including an option to not fuzz this return field. There will be num_values^{14} combinations for one `stat` syscall invocation. To make the fuzzing finish in a reasonable amount of time, we instead design Emilia to fuzz one field at a time, so that the time grows linearly with the number of fields (i.e., $O(num\_fields \times num\_values)$). As a result, `update_target` method will try every value only once on every return field.

D. OS-specific optimizations

The efficiency of Emilia is largely determined by the number of syscall invocations processed. Therefore, if we can eliminate certain syscalls from consideration, it will speed up the analysis. To this end, one can apply domain-specific analysis to eliminate syscalls. Since we will evaluate Emilia on Linux, we conducted a manual audit of the Linux syscall interface, and narrow down the scope of syscalls that Emilia will consider as targets.

Linux syscalls can be categorized based on the purpose and how they are handled by the OFL:

I. Special-purpose syscalls that are discouraged for regular applications such as `kexec_load` and `query_module`, syscalls that have no corresponding libc wrapper (e.g., `io_getevents`) and those that do not exist for certain Linux versions (e.g., `getcpu`). 40 syscalls fall in this category. They are unlikely to be invoked by regular applications protected by the isolation technique.

II. Syscalls that are usually specially handled by the isolation technique and unlikely to be directly forwarded to the untrusted OS. 74 syscalls fall in this category. They are related to threads, memory management, and signals. 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.

III. 194 remaining syscalls including but not limited to file, network and time operations such as `read`, `epoll` and `gettimeofday`. Syscalls in this category are common in applications and are more likely to be forwarded by the OFL.

Based on the categorization, Emilia currently only targets the 194 syscalls that we expect to be vulnerable to Iago attacks.

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 1: Legacy applications analyzed

IV. Vulnerability Analysis

In this section, we begin by classifying the vulnerabilities found by Emilia and provide examples of each. We then quantitatively examine the results of our measurement to describe how frequently 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. Finally, we summarize some lessons-learned that will provide directions for avoiding Iago vulnerabilities for legacy code in applications.

A. Applications examined

One of the primary motivations for developing Emilia was to measure the base rate of Iago vulnerabilities in a wide range of legacy applications. To this end, we applied Emilia to 17 applications and libraries, including servers, clients, and utilities, which are summarized in Table 1. Since we intercepted the actual syscall layer instead of libc wrappers, we also analyzed the C library code invoked during fuzzing. Our system was running glibc-2.27.

Naturally, some applications are more likely to be ported to SGX than others. As a result, we also classify our applications into groups based on their functionality and how likely they are to be used in SGX, as shown in Table 2. The first group consists of basic *Utility* programs, and also includes common libraries. We believe these are the most likely to find their way into SGX enclaves, and some already have implementations in SGX[2]. The next group consists of programs that are largely *Computational* and perform little I/O. These programs have a low attack surface and we envision that users may wish to use SGX to protect these computations and any private data they operate on when run on the public cloud. The third class consists of *Server & Network* applications that we may wish to secure from a malicious OS, but represent a greater challenge due to the amount of I/O they conduct as well as their larger code bases. Finally, we include Chromium, an *Interactive* application for completeness, though we think it is unlikely that large and highly interactive applications such as web browsers will be ported into SGX.

B. Classification of vulnerabilities

We classify the Iago vulnerabilities found 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 five 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`.

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. Example: the return values of `epoll_wait` depend on previous invocations and returns.

Unauthenticated channel: In the case of a multi-component trusted application, sometimes inter-component communication uses syscalls, e.g., `read/write` to transmit data via pipes between processes/threads. When no authentication is performed, an Iago may result. Notice that this is to be distinguished with untrusted payload as both endpoints of the communication here are part of the trusted application.

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.

C. Vulnerabilities found

We now describe the vulnerabilities we found by running Emilia on our corpus of 17 applications. We ran Emilia using the Stateful fuzzing strategy and random, invalid and valid values. In total, we ran Emilia for 80 hours across all applications. We focus on the vulnerabilities found by Emilia here first, and give more detailed measurements of Emilia's performance at generating syscall coverage and finding vulnerabilities under constrained resources in Section VI.

App	Class	Static	Local	Stateful	External	Channel	Total	
OpenSSH	Utility		4				4	21
WolfSSH							0	
zlib			2				2	
libreadline			6				6	
glibc			6		1		7	
Coreutils			2				2	
BOINC	Computational		1				1	2
Evolver			1				1	
Lightthpd	Network		1	1			2	26
Apache			6	1			7	
MongoDB			1	1			2	
Redis		1	7	1			9	
Nginx			1				1	
Memcached				1		1	2	
git					1		1	
curl			1				1	
Charybdis				1			1	
Chromium	Interactive		1	1			2	2
Total								51

Table 2: Classification of tested applications by functionality and type of vulnerabilities found

In total, Emilia discovered 51 memory corruption Iago vulnerabilities in our application corpus (including glibc). Every application had at least one vulnerability except WolfSSH, and vulnerabilities were also found in every class of applications. Table 2 breaks down the vulnerabilities by type and application class. We can see that the largest density of vulnerabilities (21 in 6 applications) was in the Utility class of applications, which we felt are also the most likely to be ported into SGX. In particular, many vulnerabilities were found in key libraries, such as glibc and libreadline, which are likely to be compiled into many programs. The next class with the highest density of vulnerabilities were the Network applications. These results serve also as a word of caution as simply porting networking applications directly into SGX is likely very risky, even though there exist frameworks designed to do just that.[3]

Table 3 lists the vulnerabilities by syscall and vulnerability type. We see that the majority of vulnerabilities are Local (80.39%), followed by Stateful (11.76%). Static and local vulnerabilities, which, as discussed in Section IV-D can be easily mitigated by an OFL, account for 82.4% of vulnerabilities, suggesting that good design of OFLs will be critical to allowing easier porting of legacy applications into SGX. On the other hand, the fair number of Stateful vulnerabilities suggests that some amount of porting is needed, particularly in complex network and interactive applications. Our results

[2]For OpenSSH see https://github.com/mfriedl/sk-sgx.

[3]For example https://github.com/lsds/sgx-lkl.

App	Syscall	Count	Type
Redis	accept	1	Static (1, 1.96%)
openSSH	read (27)	2	Local (41, 80.39%)
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, 11.76%)
Apache-httpd		1	
MongoDB		1	
Redis		1	
Charybdis		1	
Chromium		1	
Git	lseek	1	External (2, 3.92%)
glibc	fstat	1	
Memcached	read	1	Channel (1, 1.96%)
Total		51	

Table 3: Detected Iago vulnerabilities

suggest that fuzzers such as Emilia can be valuable in helping developers find and fix Iago vulnerabilities in such code.

We describe in detail some of the vulnerabilities that Emilia discovered:

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 found a piece of code that uses a file descriptor returned from `accept` to index a pre-allocated file descriptor array. Before performing indexing, Redis checks the returned file descriptor with `-1`, and compares it against the max size of the array. 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. 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.

Local: Syscalls such as `read` and `getsockopt` will fill

```
1 char buf[PATH_MAX];
2 if ((len = readlink(path, buf, sizeof(buf) - 1))
      == -1)
3     ...
4     /* error handling */
5 else {
6     ...
7     buf[len] = '\0';
8     ...
9 }
```

Listing 2: An example (readlink) of the local semantics in OpenSSH

a buffer provided by the caller. There always exists an input value to specify the max length of the buffer 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 2 shows a vulnerability caused by an unbounded `readlink` return value in OpenSSH.

After reading the content of the symbolic link 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 to 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 large `len` which breaks the assumption will let the attacker set any byte beyond `buf` to zero.

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 the syscalls it has invoked previously. 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 3 shows an Iago vulnerability caused by `epoll_wait` 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.

```
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 3: An example (epoll_wait) of the stateful semantics in Lighttpd

```
1  ret = read(fd, &timeout_fd, sizeof(timeout_fd);
2  ...
3  conn_close_idle(conns[timeout_fd]);
```

Listing 4: An example (read) of unauthenticated channels in Memcached

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 the 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 was found in Charybdis. The `data` field of `epoll_event` stores a pointer to a structure which contains a function pointer. If the attacker lets the pointer point to a controlled buffer and writes a function pointer there, he can make the application call arbitrary functions.

Unauthenticated channel: In our threat model, we exclude payloads from an untrusted source such as network content. However, a legacy application which trusts the OS may assume the communication channel established among different component of the application is reliable. Listing 4 shows an Iago bug we found in Memcached. It reads a `timeout_fd` from the libevent wakeup pipe. Since the `timeout_fd` is written by another thread of the application, Memcached feels safe to use it to index an array in line 3. A malicious OS could change the `timeout_fd` to cause invalid memory access.

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 include local protected files and state in the exclusive control of the OS such as time received from `gettimeofday` and

```
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 5: An example (fstat) of the external semantics in glibc

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 found one such example in glibc's code of parsing `ld.so.cache` (Listing 5). 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` 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 than or equal to 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).

A malicious OS could also compromise the application through those return values in other ways. 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 or adding a trusted random and time source is necessary to mitigate those vulnerabilities.

In summary, from Table 3, we can see that 80% of the vulnerabilities are caused by a returned size, which goes beyond the local upper bound (e.g., the `epoll_wait` vulnerability in category Local is caused by a returned number of file descriptors which is larger than the specified `maxevents`). This is not entirely unexpected since it is a common programming

practice to use the returned length of a syscall to access the buffer used in the syscall. Examples include iterating a 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. Similarly, we also found it a common practice to store a file descriptor in the `epoll_data` field of `epoll_event` struct returned by the `epoll_wait` syscall and use it to index into a file descriptor array. The applications that did these often failed to check the validity of the syscall returned value because they assume the OS is trustworthy and correct.

D. 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 the syscall semantics, the implications and difficulty might be different, which we will discuss in the following.

Local and Static: As these types 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 3 suggests that the majority of Iago vulnerabilities can be mitigated in this way, and this would eliminate 82.4% 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 to check, but this would only be a one-time effort for each OS version.

Stateful: In contrast to Static and Local, while also straightforward to check, Stateful Iago vulnerabilities require more complex logic to maintain parallel state 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 and 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 so that it is no longer vulnerable. We found that all 6 applications we examined that contain an `epoll_wait` vulnerability can be easily fixed by replacing `epoll_wait` with other polling syscalls such as `poll` and `select` for compatibility reasons.

Unauthenticated channel: As with any network communication assuming an insecure channel, the unauthorized channel vulnerabilities may be solved using cryptography to secure certain trusted channel within the trusted application components. We also consider this type as straightforward to address.

External: As 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` bugs 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.

V. OFL Analysis

Further to the Iago vulnerabilities identified by Emilia from legacy code, we are also interested to see to what extent state-of-the-art OFLs and SGX applications mitigate Iago vulnerabilities. However, we note that Emilia is not suited to fuzzing OFLs the way it fuzzes legacy applications. There are two reasons for this. First, OFLs typically have a trusted portion that runs in the SGX enclave and an untrusted portion that runs as a process on the untrusted OS, which makes the actual syscalls. Fuzzing the syscall return values is likely to find Iago vulnerabilities in the untrusted portion, which is of no consequence to the trusted portion. Second, to properly fuzz the trusted portion, Emilia needs to be ported to each OFL to fuzz the return values that each OFLs untrusted component returns to the trusted component. As a result, we port Emilia to fuzz one particular OFL, the Google Asylo project, and perform a manual analysis of several other popular OFLs and SGX applications by examining their documentation and performing code reviews.

A. Documentation-based Analysis

We survey 17 recent OFLs and SGX applications to examine the types of Iago vulnerabilities they defend against.

Mmap and randomness Iago vulnerabilities: Mmap and randomness Iago vulnerabilities were first identified in the original Iago paper [4] and thus the most well-known. Almost all OFLs have included checks to ensure that the returned address of memory management syscalls does not overlap with previously allocated memory. Virtual Ghost [7] also introduces a random number generator to defend against an OS that provides bad randomness. Other isolation techniques that have also addressed these two vulnerabilities include: Trustshadow [11], AppShield [6], Sego [25], ShieldBox [39] and HiddenApp [42].

Other Iago vulnerabilities: Section IV-A identifies several other types of Iago vulnerabilities, which have not been systematically documented in the literature. As a result, only some OFLs make explicit mention of mitigating these other vulnerabilities, while many do not mention them at all. Most OFLs make some effort to narrow the syscall interface by only implementing certain syscalls. Minibox [27], SGX-Tor [21] and InkTag [12] 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 leave network I/O directly forwarded to the OS for the reason that network was originally considered as an untrusted communication channel by the application and cryptographic protocols may be applied to help secure the channel. However, we have shown that in addition to the content of network traffic, metadata like size, descriptors or pointers returned by network syscalls can also be attack vectors.

Ryoan [14], SeCage [29] and Glamdring [28] claim that they would apply some checks in the OFL to validate the return value of syscalls, but no information is disclosed about what exactly those checks are. Panoply [35] studied the types of syscall return values and categorized 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. OpenSGX [18] also carefully considers the potential attack surface on their OS interface and has a list of corresponding checks they could apply on the OFL. However, none of them perform experiments on real applications to prove the existence of such vulnerabilities. Moreover, the Stateful and External examples we have presented indicate that without implementing some part of the OS functionality, such as `epoll`, to get a global view of the managed content, a stateless OFL check alone is insufficient to detect the inconsistency between actual syscalls' behavior and the application's assumptions.

As a strong form of mitigation, Haven [3], Graphene-SGX [40], and SCONE [2] place a library OS 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-SGX, the library OS can track the offset of opened files and all `epoll` event data. In other words, the library OS acts as a stateful OFL to address the Stateful type of vulnerabilities we have identified. However, Van Bulck et al. [41] found the return value from `read` was used to copy the return buffer in the Graphene-SGX OFL itself, which lead to memory corruption. This vulnerability was patched in April 2019 (hence rated as No for OFL vulnerable in Table 4).

B. Code-based Analysis

To further delve into the state of OFL defenses against Iago attacks, we conduct an analysis of 6 OFLs through code review, in addition to examining publicized information. Our analysis includes both general-purpose isolation frameworks (Graphene-SGX, Asylo and Virtual Ghost) and TEE-secured applications that use their own custom OFL (SGX-SQLite, SGX-Tor and mbedtls-SGX). We selected these code-bases based on the type of OFL (3 general and 3 application-specific) and based on the availability, apparent maturity and completeness of their code bases. We note that because the following findings are made via manual code review, they are lower bound on the true number of Iago vulnerabilities in these OFLs. We analyze whether the OFL mitigates Static, Local and Stateful vulnerabilities in legacy applications, and as well as whether the OFL itself is vulnerable to Iago attacks. The results are tabulated in Table 4.

The main purpose of our analysis is to find out what countermeasures the OFLs take to mitigate Iago vulnerabilities when forwarding syscall return values to the application, based on the Static, Local and Stateful vulnerability types that can be mitigated by OFLs. For each of the vulnerability types, we classify the level of mitigation into three levels: The worst case is an OFL that forwards syscalls and does not have mitigations on any of the syscalls. Next, an OFL may forward syscalls and have mitigations on some of the syscalls but not others, making it incomplete. Finally, in the ideal case, the OFL either doesn't forward any syscalls (and is thus not vulnerable), or it has mitigations for all forwarded syscalls, allowing it to provide complete protection for that class of Iago vulnerability to a legacy application.

OFL	Mitigation for apps			Application Vuln.	OFL Vuln.
	Static	Local	Stateful		
Graphene-SGX	○	●	●	-	No
Google Asylo	○	◐	○	-	Yes (patched)
Virtual Ghost	○	○	●	-	Yes
SGX-SQLite	○	○	●	Yes	No
SGX-Tor	○	○	○	Yes	No
mbedtls-SGX	○	○	●	No	No

○: Syscalls forwarded and no mitigations for legacy applications
◐: Syscalls forwarded and incomplete mitigations for legacy applications
●: Either no syscalls forwarded or complete mitigations for all forwaded syscalls

Table 4: OFLs analyzed for Iago attack mitigation. The columns of Static, Local and Stateful indicate whether the OFL code has checks for violation of the corresponding semantics. The last column is used to note if we have found the OFL itself to contain Iago vulnerabilities, as opposed to just not checking for the protected application code

Table 4 shows that most OFLs do not provide complete mitigation for Iago vulnerabilities, and many provide no mitigation at all for the most numerous class of Local vulnerabilities. We also list whether the lack of mitigation resulted in Iago vulnerabilities in the underlying applications (only applicable to the SGX applications), as well as whether there were any vulnerabilities a malicious OS could trigger in the OFL code itself. We detail our analysis below.

Static: We were unable to find any checks in the six OFLs for syscall return values that can be statically verified, e.g., we did not see any verification for negative error codes. That means the `accept` vulnerability (Static) in the legacy Redis may still be exposed to the attacker. Fortunately, our findings in Section IV-C also show that the frequency of Static vulnerabilities in legacy code tends to be low (though non-zero).

Local: With the help of the library OS, Graphene-SGX narrows the untrusted interface to only 37 OCALLs, and is thus able to secure that narrow interface with comprehensive verification. They define an `sgx_copy_to_enclave(ptr, maxsize, uptr, usize)` function which compares the trusted and untrusted buffer lengths before copying any untrusted buffer. It also ensures both trusted and untrusted buffers completely reside in the corresponding memory region (inside/outside enclave). Even though Virtual Ghost instruments the protected application to ensure pointers passed into or returned by the untrusted OS do not point into the protected memory region, we did not find any verification in its OFL. In particular, we were unable to find any checks applied on the syscall return values from `read`, even though it is described as an example in the paper.

SGX-SQLite, SGX-Tor and mbedtls-SGX are three SGX-secured applications ported from legacy code. Instead of using an existing isolation framework such as Asylo or Graphene-SGX, they develop custom OCALL interfaces to only forward

necessary syscalls. However, they still forward the syscalls that involve the local Iago semantics. The lack of mitigation allowed us to find two unmitigated Local Iago vulnerabilities in SQLite's handling of `read` and `readlink` syscall return values in SGX-SQLite.

Stateful: Due to the peculiarity of this type, such vulnerabilities are often mitigated by just not forwarding the corresponding syscalls. As mentioned above, Graphene-SGX forwards no syscalls whose return values involve stateful semantics. Therefore, Graphene-SGX can successfully defend legacy code against both stateful (by not forwarding) and local vulnerabilities. Virtual Ghost, SGX-SQLite and mbedtls-SGX are the same case. However, we found one unmitigated Stateful vulnerability caused by `epoll_wait` (which is forwarded) in SGX-Tor.

Vulnerable OFLs: Compared to the threat model inconsistency faced by legacy code, Iago vulnerabilities have been included in the threat model of the OFL development. Still, we see Iago vulnerabilities in the code of certain OFLs. For example, the Virtual Ghost OFL is vulnerable to the local semantic violation (e.g., in `read` and `readlink`). It uses the untrusted buffer length to `memcpy` content from the shared memory to the private memory for almost all syscalls that return the number of bytes processed. We separate vulnerabilities inside the OFL from insufficient mitigations because those vulnerabilities can be exploited without having Iago vulnerabilities in the protected code. We also found confirmed vulnerabilities in the OFL of Asylo, which we discuss next.

C. Google Asylo

Motivated by the identified vulnerable OFLs through code review, we take Google Asylo as a typical example of commercial OFL implementations, and fuzz its OFL for Iago vulnerabilities. Google Asylo [9] is a recent (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 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 by the OFL. The forwarding is performed by wrappers in the enclave which send syscall parameters and copy back outputs, and handler functions that make actual 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.

We modify Emilia to intercept Asylo's untrusted syscall handler, which handles and replies syscall requests forwarded by the OFL. The fuzzing loop algorithm we used 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. The other reason why we do not use strace to intercept target syscalls is that the untrusted handler will also invoke syscalls, of which 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 decided 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 syscall return values in the handler before sending them to the trusted part is a valid fuzzing method.

Two vulnerabilities discovered, reported and fixed: We found two memory corruption vulnerabilities involving 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 was 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 contacted Google and these two vulnerabilities were confirmed. They have been patched by comparing the received size with the input parameter `optlen` and `msg_namelen`.

In Section II, we assume the OFL will only copy as much data as it has allocated space for. If the OFL fails to do this, some vulnerabilities could have more severe consequences than we originally expected. Consider the following application code:

```
1 optlen = 200;
2 char* opts = malloc(optlen);
3 char* buf = malloc(optlen);
4 getsockopt(fd, level, name, opts, &optlen);
5 memcpy(buf, opts, optlen);
```

In `getsockopt`, both `opts` and `optlen` are untrusted return values provided by the malicious OS. If we assume the OFL copies at most 200 bytes to `opts`, then the malicious OS can only overwrite `buf` with uncertain data. However, Asylo does not truncate the `opts` buffer based on the allocated size of 200 in this case, causing the `memcpy` to copy all of the malicious payload provided by the OS instead of just the first 200 bytes.

Lack of Iago mitigation for applications: The patches Google Asylo added to fix the two vulnerabilities prevent the malicious value-result argument from being further returned to the application. However, in addition to the above mitigation, there remain other missing Iago mitigations. While Asylo

developers also added the check for the `read` syscall in September 2020, Ayslo currently still misses checks for other syscalls such as `readlink` and `write`. In addition, Asylo does not perform any stateful check for the `epoll_wait` syscall to protect code that is vulnerable to the epoll attack. Because these are missing mitigations, they are only vulnerabilities if the legacy application using Asylo misuses the return values, so they are not considered vulnerabilities in Asylo. However, we still plan to report them to developers after properly documenting them.

VI. Evaluation

We evaluate Emilia's performance and efficiency at finding Iago vulnerabilities under the different target- and value-search strategies outlined in Section III-C across different applications. We are not aware of any other fuzzers that will fuzz syscall return values to find Iago vulnerabilities. Thus, as a Baseline, we use Emilia in its simplest configuration, which uses the Fuzz-all target selection strategy and a value set composed of only random and invalid values (i.e., without valid values from the value extractor). We compare this against Emilia using the Stateful target-selection strategy but still without valid values, and finally Stateful with valid values included to evaluate the benefits of the Value Extractor.

Metrics: We measure the number of unique core dumps produced (Table 5) and syscall coverage (Table 6) and use these as the basis for comparison.

Unique core dumps, defined as a core dump with a unique program counter and unique call stack. We note that under the Iago vulnerability model, any sequence of fuzzed return values that Emilia uses to trigger a core dump is a legitimate sequence that could be returned by a malicious OS to generate the same core dump. Thus, in principle all core dumps counted here are real crashes with no false positives. However, we also point out that the number of core dumps listed in this section does not match the vulnerabilities found in Table 3, as those have been manually analyzed and deduplicated, while a program counter invocation and call stack do not necessarily mean the root cause of the crash is unique. In addition, some of the core dumps attributed here to applications may actually occur in libraries (like glibc), which we don't attribute to the application itself in Table 3. As recommended by [22], we run each experiment 30 times and apply the Mann Whitney U-test for statistical hypothesis testing to verify the significance of the results.

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

A. Evaluation of Fuzzing Strategies

One property of the Stateful strategy is that it is able to selectively target syscalls that depend on a preceding syscall. In general, this allows the Stateful strategy to target more syscalls than the Fuzz-all strategy, which can only target syscalls (and all following syscalls) that appear in the vanilla sequence. As a result, the Fuzz-all strategy may run out of targets before the Stateful strategy does. To make the strategies comparable, we execute the Stateful strategy until it exhausts all targets, and then execute the Fuzz-all strategy for the same amount of time by resetting `skip_count` so that it will return to the beginning of the vanilla sequence. While this causes it to fuzz previously fuzzed targets, we note it fuzzes them with different random values. Since the number of targets is application dependent, each application is fuzzed for a different length of time.

We tabulate both the number of core dumps and syscalls generated by Emilia's different fuzzing strategies in Tables 5 and 6 respectively. We also show the number of core dumps found as a function of time in Figure 2. The tables report the Min, Max, Median, Mean, 95% upper and lower confidence intervals, Variance and p-values over 30 runs. Table 5 gives the length of time the application was executed for in all configurations and Table 6 gives the number of syscall invocations in the vanilla sequence. All p-values are calculated by comparing the configuration against the Stateful (without valid values) configuration. All p-values in our experiments were small, indicating that the results are statistically significant and conclusions can be drawn.

We first compare the Stateful target-selection strategy against the Fuzz-all strategy used in the Baseline. We find that the Stateful (w/o valid) strategy always achieves more syscall coverage and produces more core dumps than the Baseline. This is because Stateful method is able to explicitly target newly found syscalls by replaying the previously fuzzed syscall return values that the new syscall depends on. This is partially attributed to the fact that the Fuzz-all strategy used in the baseline is unable to trigger some crashes due to the limitations discussed in Section III-C1. For example, in Memcached, it fails to trigger memory corruption in `fprintf`, because an "extra" syscall between the target syscall and the unsafe use of the fuzzed return values causes the application to terminate prematurely. The Stateful strategy is able to target the appropriate syscall but leave the extra syscall unfuzzed and thus find the vulnerability.

We also observe that the Fuzz-all strategy wastes time as it is more likely to trigger infinite loops in the application, as explained in Section III-C1. To see why, consider a sequence of syscalls $S_1, S_2, ...S_n$, where S_n causes an infinite loop and timeout when fuzzed. Since Stateful targets individual syscalls for fuzzing and can detect which ones cause an infinite loop, it identifies S_n as generating an infinite loop after 10 invocations and exits. Fuzz-all on the other hand can't detect the infinite loop and must wait for a timeout to expire. This is further compounded by the fact that S_n will generate an infinite loop any time Fuzz-all targets a syscall $< S_n$, thus causing multiple timeouts.

However, we find that for short runs, the Stateful strategy may find fewer vulnerabilities than the baseline method. As demonstrated by Openssh, Redis and Lighttpd in Figure 2, the Fuzz-all strategy is initially able to find more vulnerabilities than the Stateful strategy because it is able to quickly target all syscalls in the vanilla sequence. However, once the Stateful strategy is able to discover and fuzz new syscalls that the simpler Fuzz-all is not able to, it will catch up and overtake Fuzz-all. This demonstrates a trade-off in fuzzing speed and completeness between the two strategies.

Application	Method	Min	Max	Median	Mean	95% CI lower	95% CI upper	Var	p-value
OpenSSH (20 hours)	Baseline	13	26	17	17.50	15	19	9.98	1.70E-06
	Stateful (w/o valid)	17	25	21	21.30	20	22	3.61	N/A
	Stateful	22	30	26	26.07	25	27	3.26	2.09E-10
Redis (4 hours)	Baseline	7	11	9	9.13	9	9	0.65	5.22E-12
	Stateful (w/o valid)	15	20	17	17.26	17	18	1.13	N/A
	Stateful	16	24	20	19.80	19	21	3.69	2.13E-07
Curl (4 hours)	Baseline	2	4	3	3.23	3	4	0.38	2.40E-09
	Stateful (w/o valid)	4	6	4	4.60	4	5	0.44	N/A
	Stateful	3	7	5	4.90	4	5	1.36	1.26E-01
Lighttpd (6 hours)	Baseline	16	25	18	18.73	18	19	4.06	1.63E-08
	Stateful (w/o valid)	19	25	22	22.60	22	24	2.97	N/A
	Stateful	20	26	24	23.67	23	24	2.02	9.13E-03
Memcached (4 hours)	Baseline	4	7	6	5.56	5	6	0.65	9.24E-12
	Stateful (w/o valid)	36	51	40	41.20	38	43	13.89	N/A
	Stateful	47	60	55	54.80	54	57	11.03	2.31E-11

Table 5: Number of core dumps. p-values are calculated by comparing with the Stateful (w/o valid) setting. We also tabulate the range, averages, confidence intervals and variance

Application	Method	Min	Max	Median	Mean	95% CI lower	95% CI upper	Var	p-value
OpenSSH (389)	Baseline	772	1156	790	843.40	786	797	13490.17	1.42E-11
	Stateful (w/o valid)	1482	1559	1527	1525.93	1515	1536	361.13	N/A
	Stateful	2400	2671	2447	2468.70	2436	2464	4813.61	1.42E-11
Redis (94)	Baseline	500	590	540	539.87	532	548	334.12	1.42E-11
	Stateful (w/o valid)	822	929	846	847.90	842	853	368.76	N/A
	Stateful	899	1024	925	930.60	917	937	633.91	6.26E-11
Curl (59)	Baseline	165	178	170	179.53	169	171	8.50	1.26E-11
	Stateful (w/o valid)	189	205	193	194.17	190	198	23.50	N/A
	Stateful	210	230	225	223.86	222	225	16.44	1.27E-11
Lighttpd (194)	Baseline	543	803	580	617.70	563	646	5687.81	1.43E-11
	Stateful (w/o valid)	1344	1562	1371	1381.80	1364	1382	1549.76	N/A
	Stateful	1978	2080	2037	2034.90	2019	2056	807.22	1.42E-11
Memcached (152)	Baseline	589	627	614	611.93	608	618	104.86	1.42E-11
	Stateful (w/o valid)	759	829	798	797.93	794	806	226.66	N/A
	Stateful	1003	1040	1026	1024.10	1021	1028	93.76	1.41E-11

Table 6: Syscall coverage. The number of unique syscalls in the vanilla sequence are given in the "Application" column. p-values are calculated by comparing with the Stateful (w/o valid) setting. We tabulate the range, averages, confidence intervals and variance

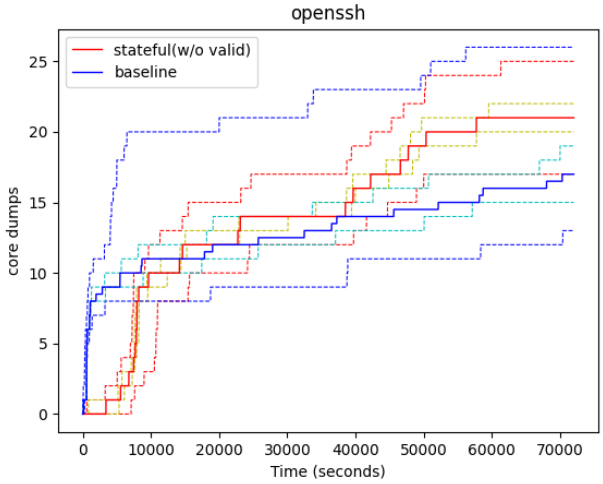
B. Effects of Valid Values

Here, we compare the Stateful strategy with and without valid values from the Value Extractor. Our experiments show that the inclusion of values in the value set also leads to greater syscall coverage and core dumps, though in general not to the extent that using Stateful as opposed to Fuzz-all target selection does. 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.

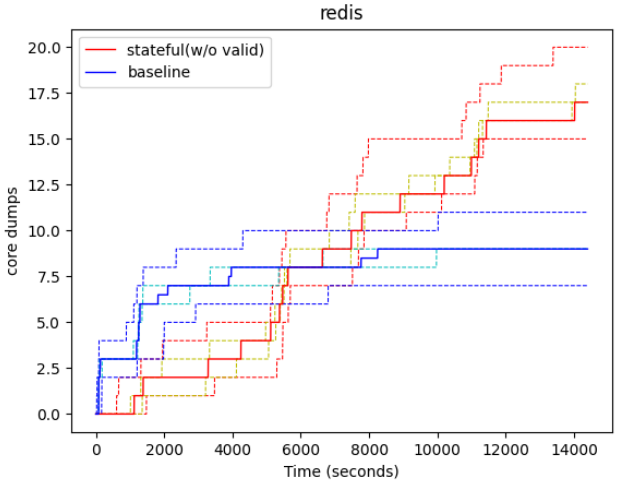
In summary, our evaluation shows that both Value Extraction and Stateful target-selection contribute to increased syscall coverage and more core dumps being found. While they require a longer time to achieve results due to the larger number of targets, they are better able to make use of additional computational resources, on average increasing the syscall coverage by 2.1× and number of core dumps found by 2.4×.

VII. Limitations

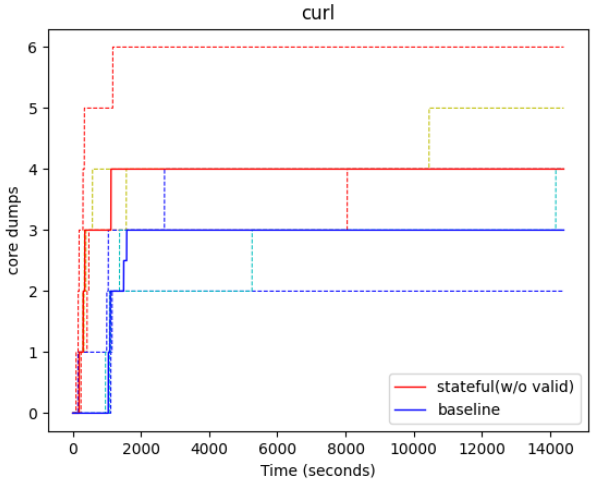
As a fuzzing tool, Emilia has some limitations. First, it only fuzzes the syscall return values but not inputs to the application, which limits the code Emilia can cover. We believe it would be straightforward to combine Emilia with a standard input fuzzer to achieve both code coverage and syscall coverage. Second, Emilia requires source code to generate the valid value sets, and the number of unique crashes found does not increase a lot by adding valid values (Table 5). This is because not all values in the valid set are useful for finding new syscall invocations due to our coarse-grained static value extractor. Emilia currently also does not associate values from the Value Extractor with specific invocations, but only with the syscall type (i.e., syscall number). When we cannot associate a value with a specific syscall type, Emilia adds the value to the valid set of all syscalls. Third, Emilia only uses the stack hash to identify syscall invocations but not the arguments. As a result, if return value handling is different based on different syscall arguments, Emilia may miss some Iago vulnerabilities.



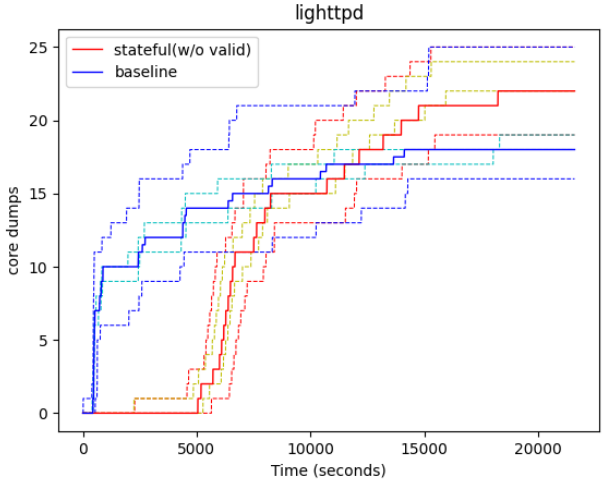
(a) Openssh



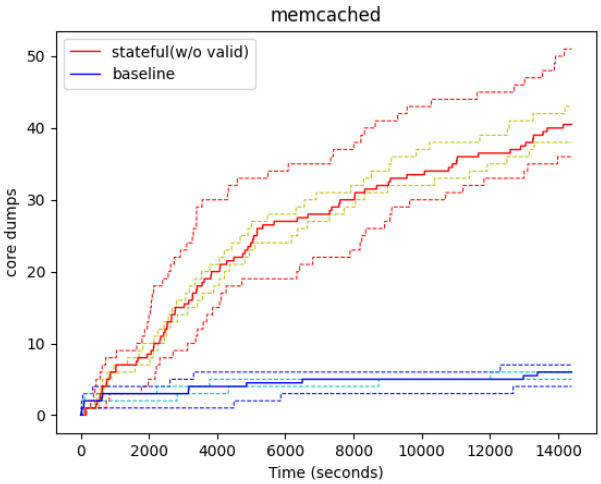
(b) Redis



(c) Curl



(d) Lighttpd



(e) Memcached

Figure 2: Crashes found over time. Solid lines are medians; dashed lines are max/min; Yellow and cyan lines are confidence intervals for stateful (w/o valid) and baseline settings

VIII. Related Work

A review of previous OFLs is given in Section V. Here, we focus on prior work in detecting and analyzing systems for Iago vulnerabilities.

Hong Hu et al. [13] studied the memory access vulnerabilities leading to arbitrary code execution despite privilege separation mechanisms that divide software into trusted and untrusted partitions. If the OS kernel is considered as the untrusted one, then it matches the Iago attack model. They use binary level symbolic execution and dynamic taint analysis to detect invalid memory access introduced by data received through the untrusted interface. Their fine-grained analysis could also analyze the capability of the attacker for each vulnerability found. However, symbolic execution suffers from path explosion, and their work was only evaluated on simple programs or function level. In 2019, Jo Van Bulck et al. [41] 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 [40] and SGX-LKL [33] similar to the one we found in Google Asylo. Their work is more like a guideline, and all the analysis was done manually. COIN attacks [20] describe Iago attacks as a subset of input manipulation against the SGX enclave's untrusted interfaces. They use symbolic execution and several policies to identify vulnerabilities caused by OCALL return values. 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 for porting them. Moreover, we are the first to use fuzzing to detect Iago attacks, which can apply to large applications.

IX. Conclusion

Using Emilia, developed as part of this work, we were able to ascertain a base rate of Iago vulnerabilities over a set of 17 diverse legacy applications and libraries. We find and detect 51 Iago vulnerabilities, and note that they are widespread, with nearly every application or library having at least one vulnerability. Categorizing the vulnerabilities into Static, Local, Stateful, Unauthenticated Channel and External classes, we find that 82.4% are Static and Local vulnerabilities, which can be easily mitigated by an OFL using simple, stateless checks. Our analysis of current, state-of-the-art OFLs and SGX applications shows that the majority do not completely mitigate all Static and Local vulnerabilities in legacy applications, suggesting that OFLs may benefit from research into how to systematically check for and detect attacks against these vulnerabilities. Finally, our results show that using Stateful target-selection and Value Extraction Emilia is able to achieve syscall coverage significantly better than a baseline fuzzer that does not use these features.

Acknowledgments

We thank the anonymous reviewers for their insightful suggestions and comments. We would also like to thank Sibin Mohan, Raskesh Bobba, Somesh Jha, Tom Reps, Austin Kuo, Wei Huang, Shengjie Xu and He Shuang for their suggestions and feedback, which helped improve our research. This research was supported by ONR Award N00014-17-1-2889 and NSERC Discovery Grant RGPIN-2018-05931.

References

[1] *ARM Security technology: Building a secure system using TrustZone technology (white paper)*, ARM Ltd, 2009.

[2] S. Arnautov, B. Trach, F. Gregor, T. Knauth, A. Martin, C. Priebe, J. Lind, D. Muthukumaran, D. O'keeffe, M. L. Stillwell *et al.*, "SCONE: Secure linux containers with Intel SGX," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI'16)*, Savannah, GA, USA, 2016, pp. 689–703.

[3] A. Baumann, M. Peinado, and G. Hunt, "Shielding applications from an untrusted cloud with haven," *ACM Transactions on Computer Systems (TOCS)*, vol. 33, no. 3, pp. 1–26, 2015.

[4] S. Checkoway and H. Shacham, "Iago attacks: Why the system call API is a bad untrusted RPC interface," *SIGPLAN Not.*, vol. 48, no. 4, pp. 253–264, Mar. 2013.

[5] X. Chen, T. Garfinkel, E. C. Lewis, P. Subrahmanyam, C. A. Waldspurger, D. Boneh, J. Dwoskin, and D. R. Ports, "Overshadow: A virtualization-based approach to retrofitting protection in commodity operating systems," *SIGPLAN Not.*, vol. 43, no. 3, pp. 2–13, Mar. 2008.

[6] Y. Cheng, X. Ding, and R. Deng, "Appshield: Protecting applications against untrusted operating system," *Singaport Management University Technical Report, SMU-SIS-13*, vol. 101, 2013.

[7] J. Criswell, N. Dautenhahn, and V. 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)*, Salt Lake City, UT, USA, 2014, pp. 81–96.

[8] J. de Ruiter and E. Poll, "Protocol state fuzzing of TLS implementations," in *24th USENIX Security Symposium (USENIX Security'15)*, Washington, D.C., USA, 2015, pp. 193–206.

[9] *Asylo*, Google, available at https://asylo.dev [Accessed December 30, 2020].

[10] J. Greene, *Intel Trusted Execution Technology, white paper*, Intel Corporation, 2012.

[11] L. Guan, P. Liu, X. Xing, X. Ge, S. Zhang, M. Yu, and T. Jaeger, "Trustshadow: Secure execution of unmodified applications with ARM TrustZone," in *15th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys'17)*, Niagara Falls, NY, USA, 2017, pp. 488–501.

[12] O. S. Hofmann, S. Kim, A. M. Dunn, M. Z. Lee, and E. Witchel, "Inktag: Secure applications on an untrusted operating system," *SIGPLAN Not.*, vol. 48, no. 4, pp. 265–278, Mar. 2013.

[13] H. Hu, Z. L. Chua, Z. Liang, and P. Saxena, "Identifying arbitrary memory access vulnerabilities in privilege-separated software," in *20th European Symposium on Research in Computer Security (ESORICS'15)*, ser. Lecture Notes in Computer Science, G. Pernul, P. Y. A. Ryan, and E. R. Weippl, Eds., vol. 9327, Vienna, Austria, 2015, pp. 312–331.

[14] T. Hunt, Z. Zhu, Y. Xu, S. Peter, and E. Witchel, "Ryoan: A distributed sandbox for untrusted computation on secret data," *ACM Transactions on Computer Systems (TOCS)*, vol. 35, no. 4, pp. 1–32, 2018.

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

[16] *Processor Tracing*, Intel Corporation, available at https://software.intel.com/content/www/us/en/develop/blogs/processor-tracing.html [Accessed December 30, 2020].

[17] *Intel SGX SSL*, Intel Corporation, 2019, available at https://github.com/intel/intel-sgx-ssl [Accessed December 30, 2020].

[18] P. Jain, S. J. Desai, M. Shih, T. Kim, S. M. Kim, J. Lee, C. Choi, Y. Shin, B. B. Kang, and D. Han, "Opensgx: An open platform for SGX research," in *23rd Annual Network and Distributed System Security Symposium (NDSS'16)*, San Diego, CA, USA, 2016.

[19] D. Jones, *Trinity: Linux system call fuzzer*, available at https://github.com/kernelslacker/trinity [Accessed December 30, 2020].

[20] M. R. Khandaker, Y. Cheng, Z. Wang, and T. Wei, "COIN attacks: On insecurity of enclave untrusted interfaces in SGX," in *25th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'20)*, Lausanne, Switzerland, 2020, p. 971–985.

[21] S. Kim, J. Han, J. Ha, T. Kim, and D. Han, "SGX-Tor: A secure

and practical tor anonymity network with SGX enclaves," *IEEE/ACM Transactions on Networking*, vol. 26, no. 5, pp. 2174–2187, 2018.

[22] G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks, "Evaluating fuzz testing," in *ACM SIGSAC Conference on Computer and Communications Security (CCS'18)*, Toronto, Canada, 2018, p. 2123–2138.

[23] J. Kneschke, *Lighttpd*, 2003, available at https://www.lighttpd.net/ [Accessed December 30, 2020].

[24] R. Kunkel, D. L. Quoc, F. Gregor, S. Arnautov, P. Bhatotia, and C. Fetzer, "TensorSCONE: A secure tensorflow framework using intel SGX," *CoRR*, vol. abs/1902.04413, 2019.

[25] Y. Kwon, A. M. Dunn, M. Z. Lee, O. S. Hofmann, Y. Xu, and E. Witchel, "Sego: Pervasive trusted metadata for efficiently verified untrusted system services," *ACM SIGARCH Computer Architecture News*, vol. 44, no. 2, pp. 277–290, 2016.

[26] Large-Scale Data & Systems (LSDS) Group, *TaLoS: Efficient TLS Termination Inside SGX Enclaves for Existing Applications*, 2019, available at https://github.com/lsds/TaLoS [Accessed December 30, 2020].

[27] Y. Li, J. McCune, J. Newsome, A. Perrig, B. Baker, and W. Drewry, "Minibox: A two-way sandbox for x86 native code," in *USENIX Annual Technical Conference (USENIX ATC'14)*, Philadelphia, PA, USA, 2014, pp. 409–420.

[28] J. Lind, C. Priebe, D. Muthukumaran, D. O'Keeffe, P.-L. Aublin, F. Kelbert, T. Reiher, D. Goltzsche, D. Eyers, R. Kapitza *et al.*, "Glamdring: Automatic application partitioning for Intel SGX," in *USENIX Annual Technical Conference (USENIX ATC'17)*, Santa Clara, CA, USA, 2017, pp. 285–298.

[29] Y. Liu, T. Zhou, K. Chen, H. Chen, and Y. Xia, "Thwarting memory disclosure with efficient hypervisor-enforced intra-domain isolation," in *22nd ACM SIGSAC Conference on Computer and Communications Security (CCS'15)*, Denver, CO, USA, 2015, pp. 1607–1619.

[30] 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*, pp. 1–1, 2019.

[31] Y. Mazhkenov, *SGX-SQLite*, 2019, available at https://github.com/yerzhan7/SGX_SQLite [Accessed December 30, 2020].

[32] D. R. K. Ports and T. Garfinkel, "Towards application security on untrusted operating systems," in *3rd USENIX Workshop on Hot Topics in Security (HotSec'08)*, San Jose, CA, USA, 2008.

[33] C. Priebe, D. Muthukumaran, J. Lind, H. Zhu, S. Cui, V. A. Sartakov, and P. R. Pietzuch, "SGX-LKL: securing the host OS interface for trusted execution," *CoRR*, vol. abs/1908.11143, 2019.

[34] B. Shastry, M. Leutner, T. Fiebig, K. Thimmaraju, F. Yamaguchi, K. Rieck, S. Schmid, J.-P. Seifert, and A. Feldmann, "Static program analysis as a fuzzing aid," in *Research in Attacks, Intrusions, and Defenses*, M. Dacier, M. Bailey, M. Polychronakis, and M. Antonakakis, Eds. Springer International Publishing, 2017, pp. 26–47.

[35] S. Shinde, D. L. Tien, S. Tople, and P. Saxena, "Panoply: Low-TCB linux applications with SGX enclaves," in *24th Annual Network and Distributed System Security Symposium (NDSS'17)*, San Diego, CA, USA, 2017.

[36] *strace: linux syscall tracer*, strace, available at https://strace.io/ [Accessed December 30, 2020].

[37] L. Szekeres, M. Payer, T. Wei, and D. Song, "SoK: Eternal war in memory," in *IEEE Symposium on Security and Privacy (S&P'13)*, San Francisco, CA, USA, 2013, pp. 48–62.

[38] R. Ta-Min, L. Litty, and D. Lie, "Splitting interfaces: Making trust between applications and operating systems configurable," in *7th Symposium on Operating Systems Design and Implementation (OSDI'06)*, Seattle, WA, USA, 2006, pp. 279–292.

[39] B. Trach, A. Krohmer, F. Gregor, S. Arnautov, P. Bhatotia, and C. Fetzer, "Shieldbox: Secure middleboxes using shielded execution," in *Proceedings of the Symposium on SDN Research (SOSR'18)*, Los Angeles, CA, USA, 2018, pp. 1–14.

[40] C.-C. Tsai, D. E. Porter, and M. Vij, "Graphene-SGX: A practical library OS for unmodified applications on SGX," in *USENIX Annual Technical Conference (USENIX ATC'17)*, Santa Clara, CA, USA, 2017, pp. 645–658.

[41] J. Van Bulck, D. Oswald, E. Marin, A. Aldoseri, F. D. Garcia, and F. Piessens, "A tale of two worlds: Assessing the vulnerability of enclave shielding runtimes," in *ACM Conference on Computer and Communications Security (CCS'19)*, London, UK, 2019, pp. 1741–1758.

[42] V. Velciu, F. Stancu, and M. Chiroiu, "Hiddenapp-securing linux applications using ARM TrustZone," in *International Conference on Security for Information Technology and Communications (SECITC'18)*, Bucharest, Romania, 2018, pp. 41–52.

[43] H. Wang, E. Bauman, V. Karande, Z. Lin, Y. Cheng, and Y. Zhang, "Running language interpreters inside SGX: A lightweight, legacy-compatible script code hardening approach," in *ACM Asia Conference on Computer and Communications Security (ASIACCS'19)*, Auckland, New Zeland, 2019, pp. 114–121.