GRAMINER: Fuzz Testing Gramine LibOS to Harden the Trusted Computing Base

Jaewon Hur
Seoul National University
hurjaewon@snu.ac.kr

Byoungyoung Lee
Seoul National University
byoungyoung@snu.ac.kr

1 Introduction

Intel SGX [1] enables a variety of valuable use cases (e.g., secure data sharing [13]) by protecting an application from all other untrusted parties (e.g., host kernel). However, incorporating Intel SGX in the conventional software development introduces additional requirements. New interface between the application running in an SGX enclave and the host kernel (i.e., `ecall` and `ocall` [1]) is one of the requirements.

In this respect, legacy applications should be intrusively modified to run in the enclave as they were not implemented with Intel SGX in mind. Thus, many researches have been proposed to run the applications in the enclave without any modification [6, 10, 11]. Currently, Gramine LibOS [2] has become the major option as it is officially maintained by the Intel. However, Gramine LibOS suffers from large trusted computing base (TCB) while there is no efficient way to test it until now.

Thus, we propose GRAMINER, a full-fledged Gramine LibOS fuzzer, to help the developers quickly detect the bugs and fix them. To be specific, GRAMINER employs the powerful fuzzing technique [3, 12] to automatically and efficiently test Gramine LibOS while covering the large input space. For that, we modified syzkaller [3] to execute syscall traces and detect kernel panics in Gramine LibOS. While the current prototype of GRAMINER runs without any coverage guidance [12] and adress sanitizer (ASAN) [9], it clearly shows the potential by finding 6 new bugs within 12 CPU hours. We open-source GRAMINER under https://github.com/JaewonHur/graminer.git, so we hope GRAMINER will be used as a baseline for the following researches.

2 Background

In this section, we introduce Gramine LibOS (§2.1), and fuzzing (§2.2).

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

SysTEX '23, May 8, 2023, Rome, Italy

© 2023 Copyright held by the owner/author(s).

ACM ISBN 979-8-4007-0087-3/23/05.
https://doi.org/10.1145/3578359.3593036

2.1 Gramine LibOS

Gramine LibOS [2] is a library OS that allows legacy applications to run in the enclave protected by Intel SGX [1]. In order to protect a user-level application from an untrusted kernel, Intel SGX defines a new interface, `ecall` and `ocall`, which should be used instead of the syscalls [1]. However, as the legacy applications were not implemented with the Intel SGX in mind, they should be modified intrusively. To this end, Gramine LibOS seamlessly interposes in between the user-level application and the `ecall/ocall` interface so as to run the application without any modification.

Gramine LibOS runs by intercepting the syscalls invoked from the application and handling it through the `ecall/ocall` interface. Thus, Gramine LibOS implements most of the syscall handling logics inside it, resulting in a large trusted computing base (TCB). However, the large TCB can cause a security problem as it may introduce lots of attack vectors to the untrusted host kernel (e.g., unexpected memory bug triggered by Iago attacks [7]). Furthermore, the usability can be degraded as a bug can be triggered while running a normal application on it, breaking down the libOS.

2.2 Fuzzing

Fuzzing is one of the most successful software testing techniques that has found many bugs so far [3, 8, 12]. Briefly speaking, given a target software, a fuzzer randomly generates inputs to the software and tests it while detecting the bugs. Especially, syzkaller [3] targets kernel software and generates random syscall sequences to trigger a kernel panic. Based on the random input generation, coverage-guided fuzzer generates even advanced inputs by employing a coverage metric that measures how much the software is tested [12].

While Gramine LibOS has been implemented with security in mind and many unit tests [2], there is no technique currently to exhaustively test it. To this end, we design GRAMINER to apply the powerful fuzzing technique for testing Gramine LibOS.

3 Design of GRAMINER

Overview. GRAMINER is a full-fledged fuzzer that finds implementation bugs in Gramine LibOS [2]. We design GRAMINER to be seamlessly integrated into syzkaller [3] so that GRAMINER can reuse the features of syzkaller (e.g.,

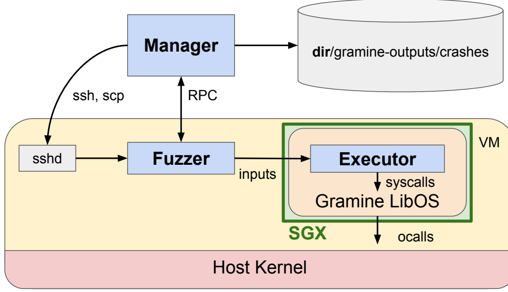

Figure 1. Design of GRAMINER

random syscall sequence generation). To be specific, the workflow of GRAMINER is shown in Figure 1. In order to run the fuzzer, GRAMINER needs a virtual machine that runs the Gramine LibOS inside. Same as the syzkaller, GRAMINER is composed of i) the manager, ii) the fuzzer, and iii) the executor. The manager continuously boots a VM which runs the testcases inside (on the Gramine LibOS), and monitors the status of the VM while saving newly found bugs. Inside the VM, the fuzzer randomly generates and provides a testcase (i.e., a random sequence of syscalls) to the executor. Finally, the executor running on the Gramine LibOS executes the received testcase so that the syscall handling implemented in Gramine LibOS can be tested.

3.1 Running Testcases on Gramine LibOS

In order to test Gramine LibOS, GRAMINER runs the executor directly on the Gramine LibOS. For that, GRAMINER first configures the runtime context (i.e., manifest file for Gramine LibOS [2]), and runs the executor on top of it. Then, the executor receives a randomly generated syscall sequence through pipe, and invokes the syscalls.

3.2 Detecting Bugs in Gramine LibOS

GRAMINER detects the bugs triggered in Gramine LibOS directly inside the VM. Specifically, the fuzzer running in the VM monitors the exit status of the executor, which invokes the random syscalls on Gramine LibOS. As the Gramine LibOS substitutes the kernel panic inside it as the program exit with a SIGPWR status [4] (i.e., power failure), the fuzzer determines the bug by comparing the status against SIGPWR. While the current implementation of GRAMINER can only detect the kernel panic (e.g., assertion failure), it can be easily extended to detect potential memory bugs by incorporating Adress Sanitizer (ASAN) [9].

Once a bug is triggered, the fuzzer compiles (and saves) the bug triggering testcase (i.e., syscall sequence) into a standalone binary, thus the bug can be easily reproduced later. Then, the manager periodically collects the bug triggering testcases into a global working directory (outside the VM) so that they are preserved across the VM power cycles.

4 Evaluation

In order to evaluate GRAMINER, we fuzzed Gramine LibOS for 12 CPU hours and found 6 assertion and memory bugs as shown in Figure 1. The bugs were disclosed to and confirmed by the developers, and some of them are already fixed [5]. While most of the bugs were related to syscall argument checking, the results clearly illustrates that GRAMINER can quickly test Gramine LibOS.

5 Discussions

Incorporating Coverage-guidance. While current implementation of GRAMINER does not employ a coverage-guidance, AFL-like coverage-guidance can also be incorporated. In order to incorporate coverage-guidance, Gramine LibOS [2] should be compiled with AFL [12] coverage instrumentation. In this case, Gramine LibOS should be modified to expose the coverage measurement to the fuzzer. For example, we can add a synthetic file which summarizes the coverage status to be read after each fuzzing iteration.

Invoking Real Security Bugs. Gramine LibOS has two input dimensions in the security perspective: i) syscall interface from the application to the libOS, and ii) `ecall`/`ocall` interface from the host kernel to the libOS. While GRAMINER currently fuzzes only the syscall interface, the `ecall`/`ocall` interface should be fuzzed simultaneously to find a security bug, which may be exploited by the host kernel. This is because the threat model of Gramine LibOS assumes only the host kernel is malicious but the application is trusted. For that, new input format should be defined, which interleaves the syscalls generated from the application and the return value of the `ocalls` generated from the host kernel.

6 Conclusion

In this paper, we propose GRAMINER, a full-fledged fuzzer to find implementation bugs in Gramine LibOS [2]. GRAMINER generates random syscalls to be invoked on top of the Gramine LibOS and finds implementation bugs including assertion failures and memory bugs. We implemented GRAMINER on syzkaller [3] and GRAMINER its practical impacts by finding 6 bugs within 12 CPU hours. We open-sourced GRAMINER under https://github.com/JaewonHur/graminer.git, so we hope it will be used as a baseline for the future researches.

References

[1] https://www.intel.com/content/www/us/en/architecture-and-technology/software-guard-extensions.html, title = Intel Software Guard Extensions (Intel SGX).
[2] https://github.com/gramineproject/gramine, title = Gramine Library OS with Intel SGX Support.
[3] https://github.com/google/syzkaller, title = syzkaller - kernel fuzzer.
[4] https://man7.org/linux/man-pages/man7/signal.7.html, title = signal(7) - Linux manual page.
[5] https://github.com/gramineproject/gramine/pull/1211.

Table 1. Disclosure of the found bugs and their status

ID	Found bugs	Status
1	Illegal instruction during Gramine internal execution at 0x7fffffee9879 (die_or_inf_loop at cpu.h)	`fixed`
2	Internal memory fault at 0x00000000 (libos_syscall_fchdir at libos_getcwd.c)	`fixed`
3	Assert failed ../libos/include/libos_flags_conv.h:25 WITHIN_MASK(prot, PROT_NONE | PROT_READ | PROT_WRITE | ...)	`fixed`
4	Assert failed ../libos/src/arch/x86_64/libos_context.c:113 IS_ALIGNED_PTR(xstate, LIBOS_XSTATE_ALIGN)	`confirmed`
5	Error: Internal memory fault with VMA at 0xffffffffff600000 (libc.so.6+0x14a7d9)	`confirmed`
6	Internal memory fault at 0x21000000 (libos_syscall_writev at libos_wrappers.c)	`fixed`

[6] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O'keeffe, Mark Stillwell, et al. Scone: Secure linux containers with intel sgx. In *OSDI*, volume 16, pages 689–703, 2016.

[7] Stephen Checkoway and Hovav Shacham. Iago attacks: Why the system call api is a bad untrusted rpc interface. *ACM SIGARCH Computer Architecture News*, 41(1):253–264, 2013.

[8] Cheolwoo Myung, Gwangmu Lee, and Byoungyoung Lee. Mundofuzz: Hypervisor fuzzing with statistical coverage testing and grammar inference. In *Proceedings of the 31th USENIX Security Symposium (Security)*, Boston, MA, August 2022.

[9] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. Addresssanitizer: A fast address sanity checker. In *Proceedings of the 2012 USENIX Annual Technical Conference (ATC)*, Boston, MA, June 2012.

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

[11] Chia-Che Tsai, Donald E Porter, and Mona Vij. Graphene-sgx: A practical library os for unmodified applications on sgx. In *USENIX Annual Technical Conference*, pages 645–658, 2017.

[12] M. Zalewski. American fuzzy lop. http://lcamtuf.coredump.cx/afl/.

[13] Wenting Zheng, Ankur Dave, Jethro G Beekman, Raluca Ada Popa, Joseph E Gonzalez, and Ion Stoica. Opaque: An oblivious and encrypted distributed analytics platform. In *Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, Boston, MA, March 2017.