

基于全系统模拟的 OP-TEE 内核模糊测试方法

王丽娜^{1,2}, 谢辉华^{1,2}, 余荣威^{1,2}, 张桐^{1,2}, 赵敬昌^{1,2}

¹ 武汉大学空天信息安全与可信计算教育部重点实验室, 武汉大学网络安全学院 武汉 中国 430072

² 武汉大学网络安全学院 武汉 中国 430072

摘要 OP-TEE(Open Portable Trusted Execution Environment)是运行于基于 TrustZone 的可信执行环境(Trusted Execution Environment, TEE)中的开源可信操作系统。OP-TEE 虽然运行于 TEE 侧,但仍存在漏洞从而遭受来自于富执行环境(Rich Execution Environment, REE)的攻击。模糊测试是一种常用的漏洞发现方法,但由于 TEE 与 REE 的高度隔离,REE 侧的模糊测试工具难以直接测试 OP-TEE,且现有基于 OP-TEE 源码插桩的模糊测试方法存在依赖源码和专业领域知识且崩溃容忍度低的问题。本文基于全系统模拟,模拟 OP-TEE 依赖的环境,提出了对 OP-TEE 内核模糊测试的方法。该方法将 OP-TEE 托管在模拟环境中并追踪其执行过程,模糊测试工具在模拟环境外观测执行过程并以此生成测试用例。该方法通过设计实现模拟环境内外通信组件,将模拟环境内 OP-TEE 的系统调用暴露给模拟环境外的模糊测试工具,使得模糊测试工具能够对 OP-TEE 内核进行模糊测试。同时针对模糊测试过程中单个用例测试耗时较长的问题,设计实现了预翻译优化机制以减少测试过程中的耗时。实验验证了方案可行性,评测了预翻译优化的效果,并评估了方案的漏洞发现能力,同时对比现有方案 OP-TEE Fuzzer 进行了性能测试。实验结果表明,本文方案具有检出崩溃以及发现潜在漏洞的能力,预翻译优化机制能平均减少 19.05% 执行耗时,且实际性能优于 OP-TEE Fuzzer,其中吞吐量与 OP-TEE Fuzzer 相比提高了 104%。

关键词 可信执行环境; OP-TEE; 全系统模拟; 模糊测试

中图法分类号 TP309.1 **DOI 号** 10.19363/J.cnki.cn10-1380/tn.2023.07.06

OP-TEE Kernel Fuzzy Testing Method Based on Whole System Emulation

WANG Lina^{1,2}, XIE Huihua^{1,2}, YU Rongwei^{1,2}, ZHANG Tong^{1,2}, ZHAO Jingchang^{1,2}

¹ Key Laboratory of Aerospace Information Security and Trusted Computing, Ministry of Education, School of Cyber Science and Engineering, Wuhan University, Wuhan 430072, China

² School of Cyber Science and Engineering, Wuhan University, Wuhan 430072, China

Abstract OP-TEE (Open Portable Trusted Execution Environment) is an open-source trusted operating system running in Trusted Execution Environment (Trusted Execution Environment, TEE) based on TrustZone. Although OP-TEE runs on the TEE side, it remains hidden vulnerabilities and suffers from attacks from the Rich Execution Environment (REE). Fuzzing is a commonly used method of vulnerability discovery, but due to the high isolation between TEE and REE, it is difficult for a fuzzing tool on the REE side to directly fuzz OP-TEE. Besides the state-of-art fuzzing method based on OP-TEE source code instrumentation has problems that rely on source code and professional domain knowledge and have low crash tolerance. Based on the whole system emulation to emulate the environment that OP-TEE relies upon, this paper proposes a method of fuzzing the OP-TEE kernel. This method hosts the OP-TEE in an emulated environment and tracks its execution process. The fuzzing tool observes the execution process outside the emulated environment and generates test cases based on the execution process. This method is designed to realize the communication components inside and outside the emulation environment, exposing the OP-TEE system calls in the emulation environment to the fuzzing tool outside the emulation environment so that the fuzzing tool can fuzz the OP-TEE core. Simultaneously, aiming at the problem that single case testing takes a long time in the fuzzing process, a pre-translation optimization mechanism is designed and implemented to reduce time consumption in the fuzzing process. Experiments are designed to verify the feasibility of the method, evaluate the effect of pre-translation optimization, and evaluate the vulnerability discovery ability of the method. Besides, an experiment is designed to compare the efficiency with the existing method OP-TEE Fuzzer. Results of the experiments show that the proposed method has the ability to find potential vulnerabilities and the pre-translation optimization mechanism can reduce the fuzzing time by 19.05% on average. Besides the actual efficiency of our method is better than OP-TEE Fuzzer. Especially, the throughput of our method is increased by 104% compared with OP-TEE Fuzzer.

通讯作者: 王丽娜, 博士, 教授, Email: lnwang@whu.edu.cn。

本课题得到国家自然科学基金项目(No. U1836112); 国家重点研发计划(No. 2020YFB1805400); 国家自然科学基金项目(No. 61876134)资助。

收稿日期: 2021-12-22; 修改日期: 2022-02-22; 定稿日期: 2023-04-17

Key words trusted execution environment; OP-TEE; whole system emulation; fuzzy test

1 引言

可信执行环境(Trusted Execution Environment, TEE)是与通用执行环境相隔离的、抗篡改的执行环境,它能保证在其内运行代码的机密性和完整性^[1]。ARM 架构的芯片通过 TrustZone^[2]机制实现 TEE。TrustZone 机制将 CPU 和内存等硬件资源划分成为两个相互隔离的执行环境,分别是富执行环境(Rich Execution Environment, REE)和 TEE, TEE 侧程序向 REE 侧程序提供安全服务。两个不同的执行环境都有各自的操作系统,REE 的操作系统为通用操作系统如 Linux, TEE 的操作系统称为可信操作系统(Trusted OS, TOS),例如 Qualcomm 的 QSEE^[3]、Samsung 的 TEEgris^[4]、Trustonic 的 Kinibi^[5]、开源的 OP-TEE^[6]。可信操作系统运行于 TEE 环境中,受到 TrustZone 机制的保护,与 REE 通过硬件的方式隔离。若可信操作系统存在漏洞,攻击者仍有可能在 REE 侧对 TEE 侧的可信应用或可信操作系统发起攻击获得更高的权限、泄露秘密信息。QSEE 就被发现存在高危漏洞^[7],攻击者利用该漏洞成功提权。除此之外 Kinibi 以及 OP-TEE 也曾被发现存在可用于越权在 TEE 内核态执行代码的高危漏洞^[8-9]。

因此,针对可信操作系统的漏洞发现引起研究者的关注,目前已有相关研究工作,例如 PartEmu^[10]、文献[11]、OP-TEE Fuzzer^[12]。

PartEmu 提出利用模拟执行技术对现实世界的 ARM TEE 侧软件进行动态测试的方法。该文献分析了不同可信操作系统和可信应用(Trusted Application, TA)与 TEE Driver(Trusted Execution Environment Driver)、BootLoader、Secure Monitor 等组件耦合的松紧程度,通过模拟松耦合组件、复用紧耦合组件的方式,模拟出可信操作系统和 TA 的运行环境,并使用该方法对 QSEE 进行了模糊测试,简略地分析了测试结果。但该文献注重的是可信操作系统和 TA 运行环境的模拟,缺乏对可信操作系统模糊测试的详细方案设计和结果分析。

文献[11]提出了一种对 OP-TEE 内核单个系统调用进行黑盒模糊测试的方案。该文献根据对 OP-TEE 了解的专业知识,设计实现了畸形 SMC(Secure Monitor Call)发生器模块以产生一系列畸形的系统调用用于模糊测试,并设计实现临时数据包记录日志模块记录最近一次执行的畸形 smc 指令^[13]的相关信息用于获得发生崩溃时最后执行的系统调用,另

外设计实现了控制器客户端、控制器服务端两个模块以控制测试流程和测试的速率。该方案能实现对 OP-TEE 内核系统调用的黑盒模糊测试,但存在如下局限性:(1) 依赖专业领域知识。畸形 SMC 发生器依赖对 OP-TEE 系统调用的了解程度。(2) 崩溃容忍度低。该方案设计实现的畸形 SMC 发生器运行在 REE 侧,导致宕机的测试用例会直接结束模糊测试过程。(3) 反馈信息有限。该方案中反馈信息仅有执行过的系统调用记录,且这些信息不能直接地、自动化地指导畸形 SMC 发生器。

OP-TEE Fuzzer 是一个可行的对 OP-TEE 进行灰盒模糊测试的完整方案。OP-TEE Fuzzer 将 OP-TEE 的系统调用视作系统调用号与其参数组合的定长结构体,使得系统调用序列能够以结构化数据的形式存储和解析,这使得传统的模糊测试用例生成方法能够用于生成 OP-TEE 系统调用的测试用例。此外,OP-TEE Fuzzer 采用源码插桩的方式,对 OP-TEE 在源码层级进行大量的插桩以获取测试用例的执行路径和路径覆盖情况,并用这些信息结合模糊测试工具 AFL(American Fuzzy Lop)^[14]生成测试用例,解决了反馈信息有限且难于自动化指导测试用例生成的问题,最终达到有效地对 OP-TEE 内核进行测试的目的。但该方法存在以下局限性:(1) 依赖 OP-TEE 源码,且需要相关领域的专业知识以在源码层级对其进行手动插桩。(2) 崩溃容忍度低,模糊测试工具运行在 REE 环境,导致宕机的测试用例会直接终止模糊测试过程。

为了解决上述方案存在的问题,本文提出一种基于全系统模拟针对 OP-TEE 的模糊测试方法。(1) 针对依赖源码和相关领域专业知识的问题,使用全系统模拟,在模拟平台翻译执行的过程中,通过动态插桩的方式记录 TEE 侧代码的执行路径,在无源码的情况下也能记录 OP-TEE 内核代码的执行路径。(2) 针对宕机测试用例导致模糊测试终止的问题,本文方案将模糊测试工具和模拟平台运行在同一层级,模拟平台记录的执行路径作为模糊测试的反馈信息传递给模糊测试工具。

本文通过借助 QEMU(Quick Emulator)^[15]实现全系统模拟,并结合模糊测试工具 AFL 搭建针对 OP-TEE 的全系统模拟模糊测试系统。通过实验验证了方案可行性,并以现有 OP-TEE 模糊测试方案 OP-TEE Fuzzer 作为评估对比方案,比较两者模糊测试的测试效率。同时还评估了本文方案的漏洞发现

能力。实验结果表明, 本文方案具有发现漏洞的能力, 且吞吐量与 OP-TEE Fuzzer 的相比提高了 104%。

本文的贡献主要有三点:

- 设计实现基于全系统模拟的 OP-TEE 内核模糊测试方案, 解决了现有方案 OP-TEE Fuzzer 依赖源码、崩溃容忍度低的问题。
- 设计实现模拟环境内外通信组件, 将模拟环境内 OP-TEE 的系统调用暴露给模拟环境外的模糊测试工具, 使得模拟环境外的模糊测试工具能够对处于模拟环境内 TEE 侧的 OP-TEE 进行模糊测试。
- 通过实验验证了本文方案的可行性和漏洞发现能力, 并与 OP-TEE Fuzzer 方案对比评估了实际性能。

2 相关工作和背景知识

2.1 TrustZone 解决方案

TrustZone 是由 ARM 提出的硬件强制隔离的安全解决方案^[16]。TrustZone 通过拓展 CPU 架构给 CPU 赋予硬件强制隔离机制, 实现了一种系统范围的、高效的安全方法。

TrustZone 通过硬件强制隔离技术, 从软硬件层面实现两个互相隔离的域: Normal World(NW)和 Secure World(SW), 分别对应于 ARM 的两个执行环境 REE 和 TEE。两个不同的执行环境运行各自的操作系统。在 REE 侧运行常见的操作系统如 Linux。在 TEE 侧运行一个小而简单的轻量操作系统。TEE 侧的操作系统的主要功能是向 TEE 的上层应用以及 REE 侧的操作系统和应用提供关键的可信服务, 同时负责管理机密数据, 包括对机密数据的加解密、存储、访问控制等。TrustZone 的隔离机制保障 TEE 侧执行的代码是可信的, 同时保障了机密数据无法直接被 REE 侧软件所访问, 极大减少了恶意软件的攻击面。

TrustZone 隔离机制的实现主要包括了处理器和内存系统。

ARM 架构处理器有四种异常等级(Exception Level, EL), 分别是 EL0、EL1、EL2 和 EL3。对于处于异常等级为 EL0、EL1 和 EL2 的处理器还存在安全状态和非安全状态两种状态, 分别表示为 NS.EL0、NS.EL1、NS.EL2 和 S.EL0、S.EL1、S.EL2, 其中 S.EL2 在 ARMv8.4 后增加。异常等级 EL3 只有安全状态。安全状态的切换通过调用 smc 指令实现, 此时运行于 EL3 的 Secure Monitor 会保存寄存器, 并修改系统寄存器 SCR 的安全状态位。

在 TrustZone 保护下, ARM 架构系统的物理地址空间会被划分为安全区域和非安全区域两个区域。在非安全状态的 CPU 中运行的进程, 不能直接访问安全状态下的地址空间, 且其虚拟地址只会转换为非安全物理地址。TLB 和 cache 同样受到了 TrustZone 的保护, 通过给它们增加安全状态位 NS Bit, 系统可以得知该缓存对应内存单元的安全状态。系统可以将被访问物理地址的安全状态作为内存属性, 控制不同执行环境对内存系统的访问。内存区域的安全属性, 可以通过配置 TrustZone 地址空间控制器(TrustZone Address Space Controller, TZASC)实现对内存区域进行划分。

TrustZone 通过硬件隔离划分出了 TEE, 但是对 TEE 侧资源的管理以及对 REE 侧提供可信服务需要可信应用(Trusted Application, TA)和可信操作系统(Trusted OS, TOS)的支持。TA 和可信操作系统分别运行于 S.EL0 和 S.EL1 异常等级下, 其中 TA 主要功能是向 REE 侧提供可信服务, 可信操作系统则是 TEE 侧的内核, 负责管理 TEE 侧的硬件资源以及向上层的 TA 提供服务, 同时和 TA 一样具有向 REE 侧提供可信服务的功能。要使 REE 的软件能够使用 TEE 侧的服务, 仅仅有 TA 和可信操作系统是不足够的。如图 1 所示, 该图展示了以 OP-TEE 为例的 TrustZone 架构图, 以及 TEE 侧和 REE 侧通信所需要的软件模块。实现 REE 与 TEE 的通信, 需要在 REE 侧的内核态即 NS.EL1 拥有 TEE Driver, 该驱动的功能主要是接收来自上层 NS.EL0 的应用的请求、管理 REE 侧应用与 TA 的会话、向 EL3 发起 SMC(Secure Monitor Call)等。REE 侧和 TEE 侧还需适用于开发者使用的接口库, 如图 1 中的 TEE Client API 和 TEE Internal API。

TrustZone 通过软硬件协同的方式, 实现了执行环境的隔离, 相对于传统的虚拟环境隔离的方法, 隔离能力更强, 安全性更高。此外, 由于 TrustZone 的隔离机制通过硬件添加新的安全状态位实现, 该实现方式引入的性能损耗极少。虽然 TrustZone 技术大大增强了系统的安全性, 但是 TrustZone 机制保护下的系统仍然会存在漏洞。TrustZone 的漏洞主要分为两类: 一类是代码实现不规范导致的安全问题; 另一类是硬件本身存在的安全问题。虽然 TrustZone 理论上可以提供完善的隔离机制和安全架构, 但是由于开发者的人为因素, 其中的可信操作系统、TA、TEE Driver 等软件模块的具体实现可能存在很多漏洞, 包括缓冲区溢出、资源争用、缺乏输入验证等问题。

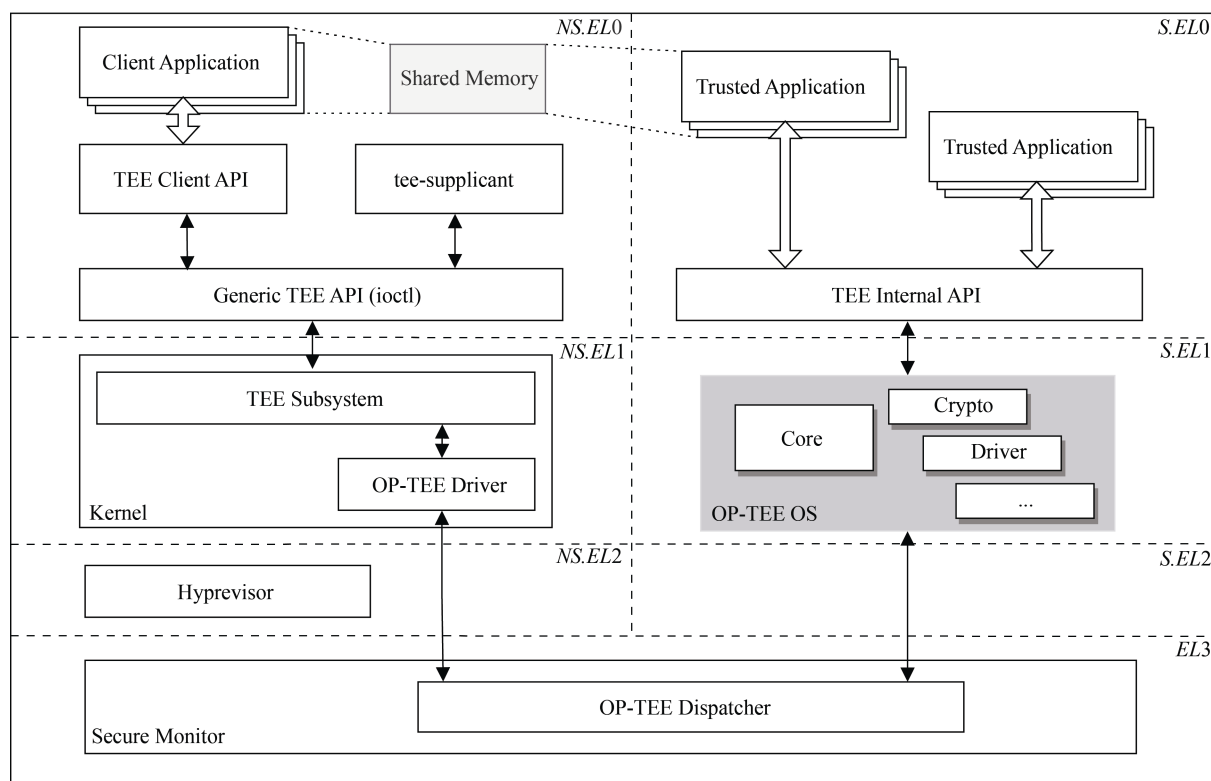


图 1 OP-TEE 作为可信操作系统的 TrustZone 架构
Figure 1 The architecture of TurstZone with OP-TEE as TOS

2.2 基于全系统模拟的模糊测试

全系统模拟是指从硬件平台开始模拟一个完整的计算机系统,与只模拟用户态运行环境的技术相区别,目前常用的全系统模拟平台有 QEMU 的全系统模式、gem5^[17]。

基于模拟执行的模糊测试即将模拟执行技术和模糊测试技术相结合,在模拟环境中运行待测对象并完整记录待测对象的执行过程,同时使用模糊测试技术对待测对象进行安全测试。由于部分软件与硬件耦合紧密,可获取反馈信息有限,且测试效率依赖被测硬件,这导致测试周期可能非常长,因此难以在真实运行环境对该类软件的进行模糊测试。例如对物联网设备裸板程序直接进行测试,需要采用硬件监控的机制才能获取程序的执行信息。这就需要待测对象运行的硬件平台能够提供硬件监控上的支持,若待测对象只能运行在不支持硬件监控的硬件平台上,就难以对其进行测试。因此,需要结合硬件模拟技术,在模拟环境下对软件进行模糊测试。

在用户态模拟模糊测试工作中,目前最为流行的模糊测试工具 AFL 支持基于 QEMU 的用户态模拟模糊测试。由于 QEMU 的用户态模拟是通过将目标程序的指令动态翻译为宿主机的指令来实现,而对于运行于不同平台的目标程序,其所依赖的系统环

境和硬件设备等均有所不同,例如存在宿主机不具备的特殊硬件,这会导致测试对象的部分代码片段不能按照预期执行,模糊测试难以得到正确的结果。因此,为了解决部分软件与硬件耦合紧密的问题,对于这类特殊软件,必须使用全系统模拟的技术进行模糊测试,使用全系统模拟目标软件运行的完整环境,包括目标软件所依赖的硬件环境、CPU 指令集等,然后对模拟环境中的目标软件进行模糊测试工作。此外,目前的模糊测试工作多仅能够在用户态工作,由于不能够轻易应用反馈机制,内核模块很难模糊测试,使用全系统模拟是内核模糊测试的一种解决办法。

TriforceAFL^[18]将 AFL 与 QEMU 全系统模拟结合,实现了对 QEMU 全系统模拟下运行程序的模糊测试,可用于对系统内核进行模糊测试。TriforceAFL 主要通过对 QEMU 的修改,添加 forkserver 和 trace 功能的实现;拓展原有指令集,在原有指令集上增添新的指令,用于完成代理和 AFL 的通信,使得代理可以通知 QEMU 启动 forkserver,获取 AFL 输入文件,并于执行完毕后关闭 forkserver 等功能,最终实现使用 AFL 对全系统模拟下的系统内核进行模糊测试。但该方法存在以下局限性: (1)使用的 QEMU 版本过低,很多最新功能不支持; (2)本身是一个架构,

具体应用需要根据上层待测程序进行适配, 如测试 Linux 内核, 需要编写用于测试的驱动程序; (3)后续没有继续进行维护。

kAFL^[19]是一个 x86-64 平台下高效的内核模糊测试工具, 主要利用了 Intel CPU 提供的 VT-x 虚拟化技术和 PT-Trace 追踪功能来提高模糊测试效率。kAFL 采用模块化的功能设计, 分为 VMM 和 VM 部分。VMM 中包括 3 个模块 KVM、QEMU-PT 和 kAFL, VM 包括目标 kernel 和其上运行的代理。KVM 实现了 PT 追踪功能, 使用 PT 技术收集目标 Kernel 的运行信息; QEMU-PT 作为 KVM 和 kAFL 交互的中间件, 并对 KVM 收集的 PT 数据进行解码; kAFL 类似于 AFL, 进行主要的模糊测试工作。kAFL 的整体架构和 TriforceAFL 有些类似, 但使用 PT 技术实现执行路径追踪, 利用硬件特性加速模糊测试, 整体效率更高。该方法同样存在部分局限: (1)由于使用了 Intel CPU 的硬件特性, 因此不支持其他如 ARM 架构的 CPU; (2)根据测试对象的不同, 需要进行一个适配, 编写少量代码的特定于操作系统的应用程序。

syzkaller^[20]是 Google 2015 年开源的系统内核 fuzz 工具, 一个非监督式、覆盖率引导的内核模糊测试工具, 支持多个系统, 不过支持最全面的还是 Linux 系统。syzkaller 同样是在模拟环境下运行测试对象, 利用系统调用对内核进行模糊测试, 其优点在于会根据内核的代码路径覆盖信息更新变异数据, 达到尽可能大的路径覆盖率。syzkaller 的 manager 通过 ssh 调用 fuzzer, fuzzer 和管 manager 通过 RPC 进行通信, fuzzer 运行于 VM 中, 并将输入传递给 executor, executor 执行系统调用, 然后 fuzzer 从 kernel 中读取代码覆盖率信息。syzkaller 一直在维护, 是目前对内核模糊测试最为流行的工具, 但其同样具有一些局限: (1)根据待测的内核, 需要对待测驱动进行定制修改, 包含比较复杂的结构体嵌套; (2)需要

使用 KCON+KASAN 编译选项编译内核镜像, 即必须有内核源码, 不能进行黑盒测试; (3)具有一定误报率。

由于全系统模拟会带来较大的性能开销, 导致模糊测试的吞吐量降低, 为此, Firm-AFL^[21]将用户态模拟和系统模拟结合, 提出增强进程模拟。通过完整的系统模拟增强用户态模拟, 测试对象大部分时间在用户模拟中运行以提升效率, 只有在执行系统调用时, 程序才会切换到系统模拟中运行, 以保证程序正确执行, 系统调用执行完毕后, 程序会重新切换回用户态模拟。此外, 因为 IOT 很多系统调用与文件系统有关, Firm-AFL 从固件中映射文件系统, 将其作为物理机中的一个目录挂载, 这样用户态模拟就可以访问, 可以直接将文件系统相关的系统调用传递到物理机, 无需重定向到系统模拟, 进一步提升了整体性能。Firm-AFL 基于 AFL 和 Firmadyne^[22]实现, 对物联网固件进行灰盒模糊测试, 相对于全系统模拟的模糊测试, 效率有了较大提高。该方法存在以下局限: (1)仅支持 POSIX 兼容的操作系统; (2)该方法的被测对象大部分执行过程可以在用户态模拟中执行完成, 而小部分系统调用则需在系统模拟中执行。该方法针对用户态模拟的优化能很好的提高测试效率, 而对于大部分执行过程都是系统调用的内核模糊测试而言, 该方案的优化则不再适用; (3)使用 AFL 的 QEMU 模式收集覆盖率信息, 即不收集在系统模拟中运行的代码的覆盖率, 对于大部分执行是系统调用的程序, 无法有效进行模糊测试。

基于全系统模拟的模糊测试目前已有较为成熟的技术和工具, 表 1 从测试效率、崩溃容忍度、操作系统依赖程度、是否依赖源码四个方面对比了上述方案的优缺点。从对比结果可以得出, 基于全系统的模糊测试方案均具有高崩溃容忍度, 且大部分方案对源码没有依赖。

表 1 基于全系统模拟的模糊测试方案对比

Table 1 Comparison of fuzzy test schemes based on full system emulation

	测试效率	崩溃容忍度	操作系统依赖程度	是否源码依赖
TriforceAFL	慢	高	中	否
Syzkaller	快	高	高	是
kAFL	快	高	中	否
Firm-AFL	快	高	高	否

3 方案设计

3.1 方案概述

本文提出的基于全系统模拟的模糊测试方法整

体架构如图 2 所示, 该方法的目标是对 ARM TEE 侧的 OP-TEE 进行模糊测试。

本文方案整体主要由三个模块组成, 分别是 Fuzzer、VM Instance、Instances Manager。

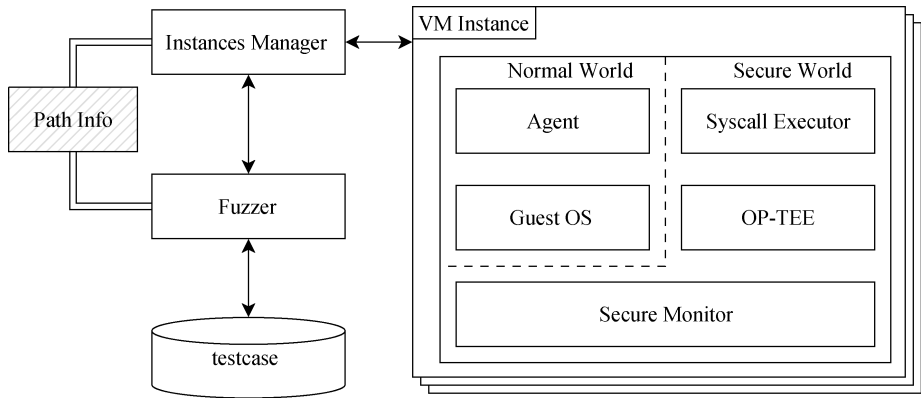


图 2 本文方案架构
Figure 2 The architecture of our solution

Fuzzer 为模糊测试工具, 用于结合程序执行路径等信息生成测试用例。该模块可通过与现有的模糊测试工具结合实现, 如 AFL。

VM Instance 是一个全系统模拟测试实例, 模拟目标硬件平台, 加载和运行一个包含 TEE 和 REE 的完整的运行环境, 并能记录 TEE 侧代码的执行过程。本文模拟目标硬件平台基于现有的模拟工具 QEMU 实现。VM Instance 通过动态插桩的方式记录 OP-TEE 代码段的执行路径, 动态插桩通过改进内核系统调用模糊测试工具 TriforceAFL 来实现。VM Instance 记录的执行路径等测试用例执行过程中产生的信息会在测试用例执行完成后回传给 Instances Manager。VM Instance 的 REE 侧运行 Guest OS 和 Agent, TEE 侧运行 Secure Monitor、Trusted OS 和 Syscall Ex-

ecutor。

Agent 是一个客户端应用(Client Application, CA), 其主要功能是向模拟环境外暴露模拟环境内的 REE 环境, 同时控制单个测试用例的执行过程。

Syscall Executor 是一个可信应用(Trusted Application, TA), 其主要功能是向 REE 侧暴露 OP-TEE 的系统调用接口, 同时接收由 Agent 提供的测试用例并在 TEE 侧实际执行测试用例的内容。Instances Manager 是每个 VM instance 的管理端, 负责将 Fuzzer 模块生成的用例传递给 Agent, 同时管理每个 VM instance 的生命周期, 包括 VM Instance 的创建、初始化和销毁。

本文方案对 OP-TEE 的模糊测试过程分为两个阶段, 如图 3 所示。

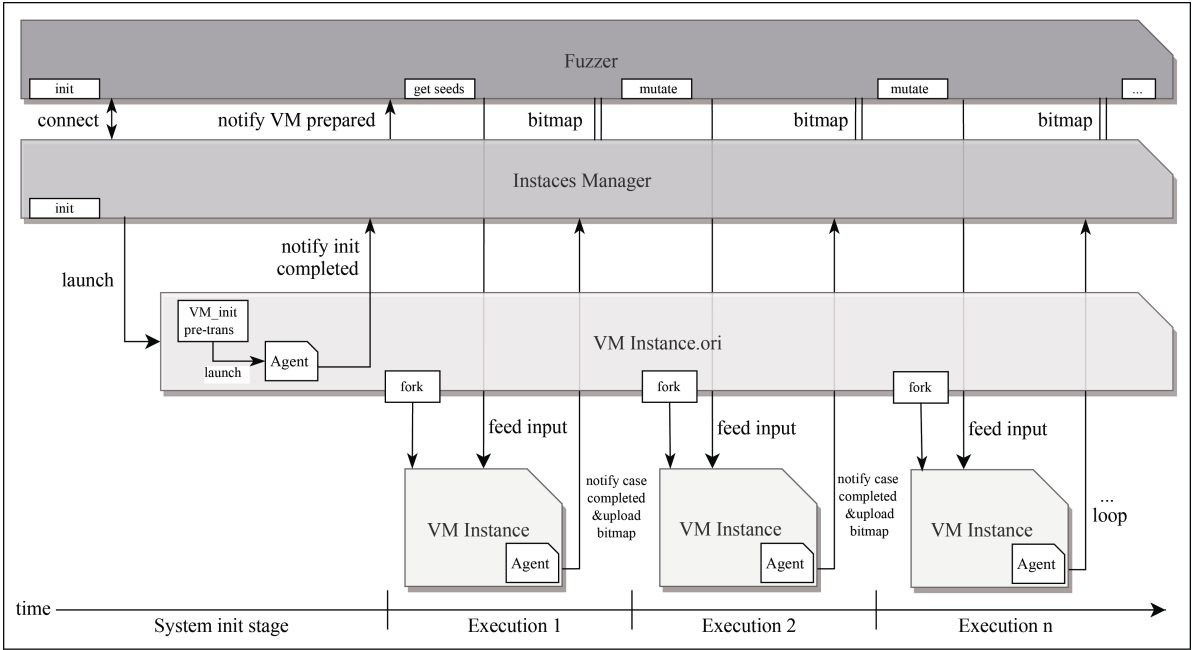


图 3 本文方案各个阶段的模块间交互图
Figure 3 Diagram of interaction between modules at various stages in our solution

第一阶段为初始化过程, 包括 Fuzzer 模块和 Instances Manager 模块的启动和初始化。Instances Manager 初始化过程会创建一个初始 VM Instance(VM Instance.ori), 并初始化该 VM Instance 的 REE 和 TEE 环境。

第二阶段为测试用例生成和执行循环过程, 由 Fuzzer 根据种子不断生成测试用例提供给 Instances Manager。Instances Manager 每获得一个新的测试用例, 就以初始 VM Instance 为原型, 复制出一个新 VM Instance。由新 VM Instance 执行测试用例并在执行完成后销毁。Instances Manager 在每个测试用例执行完成之后获取新 VM Instance 的执行路径, 更新 Fuzzer 的状态, 再进行下一轮测试过程。

3.2 Instances Manager 的设计实现

Instances Manager 主要负责 VM Instance 的创建、测试用例的传递。

Instances Manager 先启动一个实例作为初始 VM Instance, 完成该实例 REE 侧和 TEE 侧的加载和启动。接着 Instances Manager 等待由 Agent 完成初始化后的通知。

Instances Manager 收到 Agent 完成初始化的通知后, Instances Manager 先复制初始 VM Instance 得到新的 VM Instance, Agent 的后续流程将在新 VM Instance 中执行。接着 Instances Manager 等待新 VM Instance 中的 Agent 获取测试用例的通知。

Instances Manager 收到从新 Agent 发送的获取测试用例的通知后, 从 Fuzzer 模块获取新生成的用例, 并将测试用例通过进程间内存共享的方式传递给新 Agent, 同时启动定时器, 等待新 Agent 结束测试的通知或者超时信号。Instances Manager 收到新 Agent 结束通知或超时信号后, 将记录的执行路径通过共享内存的方式传递给 Fuzzer 模块, 并销毁新 VM Instance。

在将执行路径传递给 Fuzzer 后 Instances Manager 就完成了一轮测试中的所有动作, Instances Manager 会从等待 Agent 获取用例通知这一步开始下一轮测试。

Agent 与 Instances Manager 的交互流程如图 4 所示, Instances Manager 复制整个初始 VM Instance 存在较大性能损耗。因此在实现中, 对 VM Instance 的复制是通过初始 VM Instance 调用 fork 系统调用实现。由于 fork 采用了写时复制(Copy on Write, CoW)机制^[23], 新 VM Instance 只有在对内存空间写操作时才会实际复制内存资源, 因此这种实现方式能很快复制出一个新 VM Instance。同时在 Agent 执行测试

用例的过程中, 实际使用的内存资源远小于一个完整的 VM Instance 所占用的内存资源, 使用这种实现方式能极大减少不必要的损耗。实际的流程中, 在 Instances Manager 收到 Agent 的初始化完成以及测试结束的通知后, 初始 VM Instance 将会 fork 出一个新 VM Instance。

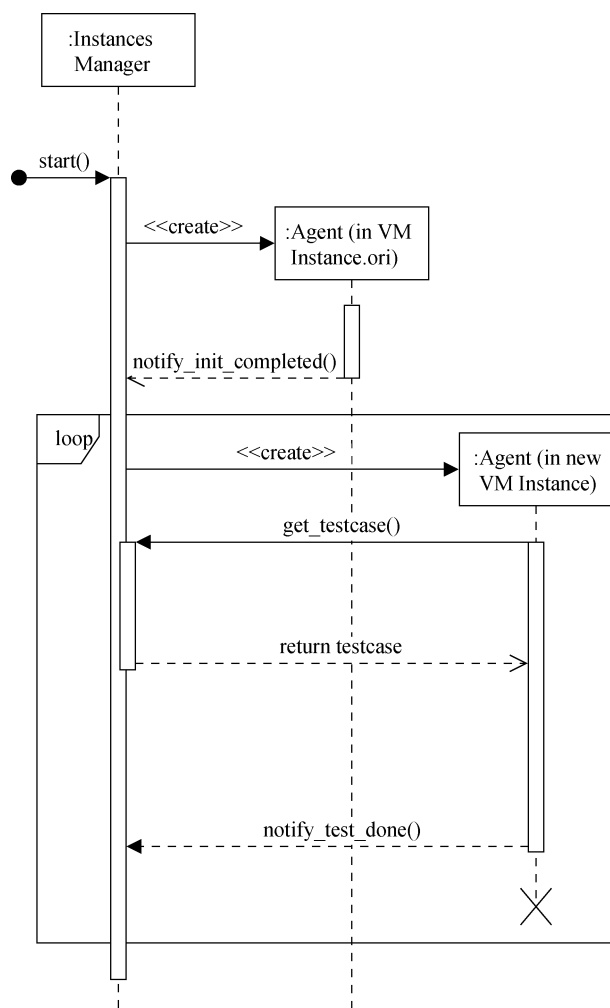


图 4 Instances Manager 与 Agent 交互流程图

Figure 4 Diagram of interaction between Instances Manager and Agent

3.3 Agent 的设计实现

Agent 负责控制单个测试用例执行的流程, 包括用例获取、用例执行、流程结束反馈。

Agent 是 REE 用户态程序, 在 REE 系统内核启动后立即启动 Agent。Agent 的执行流程如图 5 所示, Agent 启动后先进行必要的初始化, 包括检测是否首次执行 Agent、Syscall Executor 的加载和 session 的建立, 完成后将初始化结果通知 Instances Manager 并请求测试用例。Instances Manager 返回测试用例后, Agent 通过之前与 Syscall Executor 建立的 session 将

测试内容交给 Syscall Executor 执行, 执行完成后通知 Instances Manager 当前测试用例执行完成。

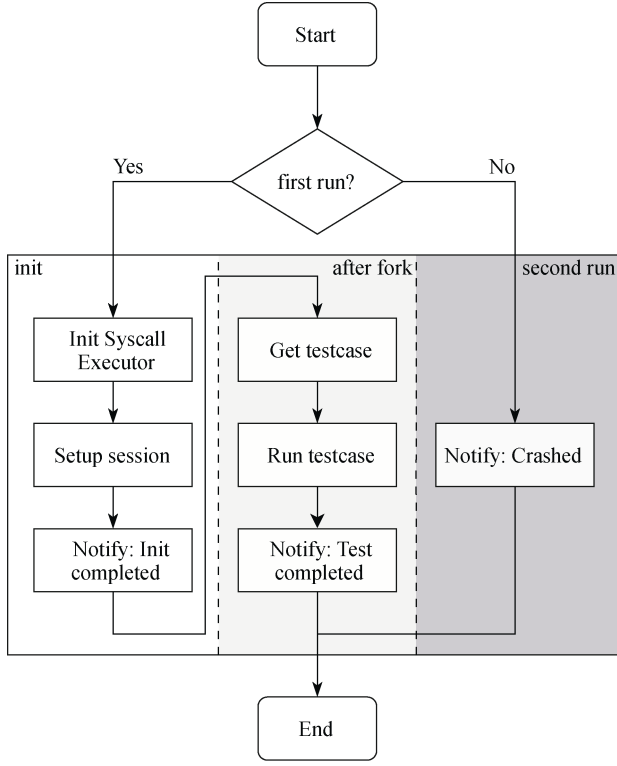


图 5 Agent 执行流程图

Figure 5 Execution flow chart of Agent

Agent 与 Instances Manager 的通信, 实际上通过在 VM Instance 的硬件模拟平台对 smc 指令的劫持实现。smc 是一条 ARMv8 平台下的指令, 用于从 REE 切换到 TEE, 指令格式为 `smc #<imm>`, 其中 *imm* 为该指令的操作数, 长度为 16 位通常为 0。当硬件模拟平台翻译到 smc 指令时, 检查 smc 的 *imm*, 针对不同的 *imm*, 实现 Agent 与宿主机的不同交互。

如果测试用例能正常执行, 则通知 Instances Manager 测试结束。如果执行过程发生崩溃导致 Agent 异常退出, REE 会重启 Agent, 并在启动时检测到该次执行不是首次测试, 依次判断上次执行发生崩溃, 并将崩溃结果通知 Instances Manager, 此次测试结束。该机制可以用于区分出导致宕机的测试用例。

3.4 Syscall Executor 设计实现

Syscall Executor 是运行在 TEE 侧的 Trusted App, 负责解析和执行从 Agent 收到的测试用例即一个 OP-TEE 系统调用序列, 其主要功能是向 REE 侧的 Agent 暴露 OP-TEE 的系统调用。由于 REE 侧的程序无法直接执行 OP-TEE 的系统调用, 因此需要设计该组件在 TEE 侧接收 Agent 的测试用例并执行。

Syscall Executor 需要解析来自 Agent 的测试用例, 每个测试用例是一个系统调用的集合, 其中每一条系统调用的格式包括了一个 8bit 长度的系统调用号, 八个 64bit 长度的字段用于存储系统调用的参数以及一个 32bit 长度的字段标识用于标识各个字段的类型。参数长度采用 64bit 长是能够兼容 32bit 的 OP-TEE 实现, 采用八个字段存储参数这是因为 OP-TEE 系统调用参数上限为 8。

Syscall Executor 解析完系统调用后对于每条系统调用逐条执行。执行过程中 VM Instance 将执行路径记录到与模糊测试工具兼容的数据结构中。在本文实现中 Fuzzer 模块基于 AFL, 执行路径由 VM Instance 硬件平台记录到和 AFL 兼容的 bitmap 中, 该 bitmap 在 Syscall Executor 执行测试用例的时候记录, 在执行完成后由 VM Instance 传递给 Instances Manager, 再由 Instances Manager 传递给 Fuzzer 模块。

3.5 预翻译优化

VM Instance 在模拟运行时会将待执行的基本块 (Basic Block, BB) 翻译成适合宿主机运行的翻译块 (Translated Block, TB)。基本块是一个只有一个入口和一个出口的代码片段。基本块中的指令只能顺序执行, 其入口为该代码片段的第一条指令, 出口为改代码片段的最后一条指令。翻译块是基本块经过 VM Instance 翻译后能够直接在宿主机处理器上执行的代码片段。现有成熟的模拟平台会将翻译块缓存起来, 同时缓存基本块和对应翻译块的映射关系, 待下次运行到该基本块时, 不用重新翻译基本块, 直接使用缓存的翻译块, 极大提高模拟运行的效率。由于从初始 VM Instance 复制得到的新 VM Instance 只运行了与 OP-TEE 启动相关的代码, 模拟平台只缓存了 OP-TEE 启动相关的翻译块, 而大部分待测的 OP-TEE 内核代码未被翻译并缓存, 因此每次执行测试用例时, 都会重新翻译并缓存待测的 OP-TEE 代码, 造成了大量不必要的性能损耗。

单个测试耗时, 如公式(1)所示:

$$T_{single_i} = T_{fork} + T_{exec_i} + \sum_{j=1}^{n_{BB_i}} T_{trans_{ij}} I_{BBs_{trans}}(BB_j) \quad (1)$$

$$I_{BBs_{trans}}(BB_j) = \begin{cases} 0 & BB_j \in BBs_{trans} \\ 1 & BB_j \notin BBs_{trans} \end{cases} \quad (2)$$

式子中 T_{single_i} 表示执行第 i 个测试用例耗时, T_{fork} 表示从初始 VM Instance 复制出新实例耗时, T_{exec_i} 表示第 i 个测试用例执行路径上所有翻译块耗时。

$\sum_{j=1}^{n_{BB_i}} T_{trans_{ij}} I_{BB_{s_{transj}}}(BB_j)$ 表示执行到第 i 个用例时模拟平台翻译基本块耗时, 其中 n_{BB_i} 表示当前测试用例需要执行的不同的基本块数量, $T_{trans_{ij}}$ 表示翻译第 i 个测试用例的第 j 个基本块所花时间, $I_{BB_{s_{transj}}}(BB_j)$ 为示性函数, 如式(2)所示, 当基本块 BB_j 在已翻译基本块集合 $BB_{s_{trans}}$ 时该函数值为 1 否则为 0。记 $p(1 > p > 0)$ 为初始 VM Instance 中已翻译 OP-TEE 内核代码占比, 平均单个测试用例耗时 \bar{T}_{single} 可表示为:

$$\bar{T}_{single} = T_{fork} + \bar{T}_{exec} + (1-p)\bar{T}_{trans} \quad (3)$$

其中 \bar{T}_{exec} 表示测试用例执行路径上所有翻译块的平均耗时, \bar{T}_{trans} 表示翻译测试用例执行路径上不同基本块的平均耗时。当执行了 n 个测试用例时, 总耗时 T_{total} 表示如公式(4)所示:

$$\begin{aligned} T_{total} &= T_{init} + \sum_{i=1}^n T_{single_i} \\ &= T_{init} + n\bar{T}_{single} \\ &= T_{init} + nT_{fork} + n\bar{T}_{exec} \\ &\quad + n(1-p)\bar{T}_{trans} \end{aligned} \quad (4)$$

其中 T_{init} 表示初始 VM Instance 模拟环境初始化用时, 随着 n 的增大, 当已翻译覆盖率 p 很小时, 翻译用时 $n(1-p)\bar{T}_{trans}$ 会显著增大, 其中包括了大量不必要的重复翻译。为了减少由重复翻译带来的性能损耗, 本文在 VM Instance 初始化的过程中加入预翻译优化机制。预翻译机制利用模拟平台的缓存机制, 在初始化 VM Instance 的过程中预先执行 OP-TEE 代码, 覆盖尽可能多待测 OP-TEE 代码, 提高翻译覆盖率 p , 缓存尽可能多的待测 OP-TEE 代码翻译块, 减少后续实际测试过程中由于重复翻译带来的性能损耗。

为了能在初始化阶段执行 OP-TEE 内核代码, 需要一个能在 TEE 侧执行代码的 TA。本文方案复用 Syscall Executor, 在初始 VM Instance 的初始化阶段, 由 Syscall Executor 执行常用应用场景的测试用例, 如随机数生成、密钥生成、加解密运算。为了尽可能覆盖内核代码, 在 VM Instance 的初始化阶段, 根据现有的 OP-TEE 的功能测试套件 OP-TEE Test^[24] 生成测试用例。OP-TEE Test 是 OP-TEE 的功能测试套件, 主要用于确保所有架构层之间的通信以及大部分 GlobalPlatform TEE Internal Core API^[25] 能正常工作, 它包含的测试用例能覆盖大部分 TEE 侧代码。

由 Syscall Executor 运行这些测试, 缓存 OP-TEE 内核大部分代码的翻译块。采用预翻译优化后执行 n 个测试用例总耗时 T'_{total} 如公式(5)所示:

$$\begin{aligned} T'_{total} &= T_{init} + T_{pre_trans} + n\bar{T}'_{single} \\ &= T_{init} + T_{pre_trans} + nT_{fork} + n\bar{T}_{exec} \\ &\quad + n(1-p')\bar{T}_{trans} \end{aligned} \quad (5)$$

p' 表示实现预翻译机制后初始 VM Instance 中已翻译内核代码占比, 且 $1 > p' \gg p > 0$ 。 T_{pre_trans} 表示预翻译耗时, 该值与 T_{init} 一样在不会随 n 的变化而变化。结合公式(4)(5)可以得出随着测试过程中 n 的不断增长, 预翻译优化机制降低测试总耗时的效果会越来越明显。

3.6 宕机检测

OP-TEE 存在直接与硬件平台的交互, 因此对其进行测试的过程中可能会触发足以导致整个系统宕机的崩溃。现有可行的针对 OP-TEE 的模糊测试方案 OP-TEE Fuzzer 在测试过程中需要 AFL 运行于 REE 侧, 由于宕机发生后 REE 侧的软件也无法正常进行, 因此对于这种类型的崩溃, 无法有效地检测出来, 而且后续测试将会被迫终止。

在本文方案中, 由于测试用例是在 VM Instance 中执行, 而单个 VM Instance 的宕机对 AFL 来说是一个测试进程的崩溃, 并不会影响 AFL 的正常运行, 因此由测试用例引起宕机并不会终止整个测试过程。测试过程发生宕机会使得 Agent 无法正常执行后续退出的流程, 后续 Agent 的结束信号就无法传达到 Instances Manager, 此时该测试用例的结果是测试超时。基于宕机的这个现象, 在 Agent 由于崩溃异常退出后, 由 Guest OS 重启 Agent, 此次启动不再是初次启动 Agent, Agent 会传递给 Instances Manager 一个结束信号, 可以用于区分出非宕机类型崩溃。而宕机类型的崩溃会作为超时测试用例处理, 根据测试用例的长度区分出其他未导致宕机但执行超时的测试用例。

4 实验评估

4.1 实验环境搭建

本文基于 QEMU 实现全系统模拟, 即 VM Instance 的硬件模拟部分, 基于 AFL 实现模糊测试工具即 Fuzzer 模块, VM Instance 中的可信操作系统即为待测对象 OP-TEE 内核, Guest OS 采用 Linux 内核, 启动固件和 Secure Monitor 采用 ARM Trusted

Firmware^[26]。AFL 原始版本 2.57b, QEMU 原始版本 5.2.0, OP-TEE OS 版本均为 3.10.0。OP-TEE OS 在本文的所有实验中所使用的版本均相同, 且除了验证是否具有漏洞发现能力的实验外, 其他实验均使用同一 OP-TEE 内核镜像。宿主机 CPU Intel(R) Core(TM) i5-7400 CPU @ 3.00GHz, 内存 8GB DDR4, VM Instance 采用的硬件配置为 Cortex A57 以及 600MB 内存。

为了测试本文方案的实际性能和代码覆盖率, 本文将现有的 OP-TEE Fuzzer 作为基线在相同的测试环境下, 对相同初始版本的 OP-TEE 内核进行模糊测试, 对比两者在模糊测试上的性能和代码覆盖率。

为了使得结果具有可比性, 搭建 OP-TEE Fuzzer 同样的环境, 在同为 QEMU 环境下, 对比 OP-TEE Fuzzer 与本文方法的优劣。运行 OP-TEE Fuzzer 的 QEMU 采用的硬件配置与 VM Instance 的相同。

4.2 预翻译优化效果

本文方案通过实现预翻译优化减少测试过程中的损耗。预翻译优化过程如下, 在初始 VM Instance 初始化的过程中, 预先翻译一部分待测代码的基本块, 并缓存翻译后得到的翻译块, 使得接下来的测试过程中从初始 VM Instance 复制得到的 VM Instance 同样具有该部分翻译块缓存, 减少了测试用例执行路径所需翻译的基本块数量, 减少了单个测试用例的执行时间, 最终达到提高整体测试效率的目的。

为了尽可能多的翻译待测代码的基本块, 提高翻译块的占比, 需要选择能够覆盖大部分常用场景的用例。在本文方案中采用 OP-TEE 功能测试套件 OP-TEE Test 中的所有用例作为预翻译阶段执行的对象。

预翻译效果根据 OP-TEE 内核中已翻译并缓存的基本块数量相对于 OP-TEE 内核基本块总数所占的比例 p_{trans} 来评测, 表示为公式(6)所示:

$$p_{trans} = \frac{num_{trans_BBs}}{num_{total_BBs}} \quad (6)$$

其中 num_{trans_BBs} 表示 OP-TEE 内核中已翻译并缓存的基本块数量, num_{total_BBs} OP-TEE 内核基本块总数。当 p_{trans} 的值越大时, 表明被翻译缓存的 OP-TEE 内核基本块数量越多, 预翻译优化的效果越好。

本文方案通过记录未使用预翻译优化以及使用了预翻译优化两者的 num_{trans_BBs} , 结合 num_{total_BBs} 计算出两者的 p_{trans} , 结果如表 2 所示:

表 2 预翻译优化实验效果

Table 2 Experimental effect of Pre-Translation

	num_{total_BBs}	num_{trans_BBs}	p_{trans}
未使用预翻译优化	13,976	1,911	13.67%
使用预翻译优化	13,976	4,781	34.21%

结果表明, 未使用预翻译优化的 num_{trans_BBs} 为 1,911, p_{trans} 为 13.67%, 而使用了预翻译优化的 num_{trans_BBs} 为 4,781, p_{trans} 为 34.21%, 使用预翻译优化与未使用预翻译优化相比, OP-TEE 内核已翻译块占比提高了 150.23%。

同时本文通过对比使用预翻译优化前后单个测试用例的执行耗时, 评估本文方案预翻译优化在实际模糊测试过程中的效果。实验选取了一组随机生成的测试用例, 对其中的每个测试用例分别在未使用预翻译优化以及使用预翻译优化两种情况下执行多次, 统计每次执行耗时并分别计算出单次执行的平均耗时, 最后计算分析使用预翻译优化相较于未使用预翻译优化耗时减少的比率, 该比率作为评估预翻译优化实际效果的指标。实验数据显示, 使用预翻译优化后执行单个测试用例耗时比未使用预翻译优化的平均减少了 1.00%~59.34%, 全部测试用例的平均执行耗时减少了 19.05%。结果表明, 预翻译优化能在一定程度上提高模糊测试的效率。

4.3 方案性能测试

本文通过实验对比了 OP-TEE Fuzzer 与本文方案的测试性能, 分别从单个测试用例平均用时与吞吐量两个方面对比两者的测试性能。

在单个测试用例平均用时方面, 选用两者的种子作为测试用例, 测试对比 OP-TEE Fuzzer 和本文方案对同样的测试用例测试用时。根据测试用例的长度和测试用例的系统调用的类型不同, 每个测试用例花费的时间不同。如表 3 所示, 本次实验中, OP-TEE Fuzzer 单个测试用例平均用时为 168ms, 本文方案单个测试用例平均用时为 129ms。本文方案单个测试用例平均用时略低于 OP-TEE Fuzzer。

表 3 OP-TEE Fuzzer 与本文方案测试效率对比

Table 3 Comparison of efficiency between OP-TEE Fuzzer and our solution

	单个用例平均用时	吞吐量
OP-TEE Fuzzer	168ms	5.38exec/s
本文方案	129ms	11.00exec/s

吞吐量方面, 测试了两者在 0~24h 的时间内同种子情况下的吞吐量。图 6 展示了在本次实验中, 本文方案与 OP-TEE Fuzzer 测试时间与测试用例总数的关系, 本文方案明显比 OP-TEE Fuzzer 具有更高的吞吐量以及更稳定的测试速率。吞吐量测试结果如表 3 所示, 在 0~24h 的时间内, OP-TEE Fuzzer 平均每秒执行 5.38 个测试用例, 本文方案平均每秒执行相同原始版本的 OP-TEE 进行模糊测试, 在种子选取行 11.00 个测试用例。实验结果表明, 在相同环境对相同的情况下, 本文方法的吞吐量与 OP-TEE Fuzzer 相比提高了 104%。

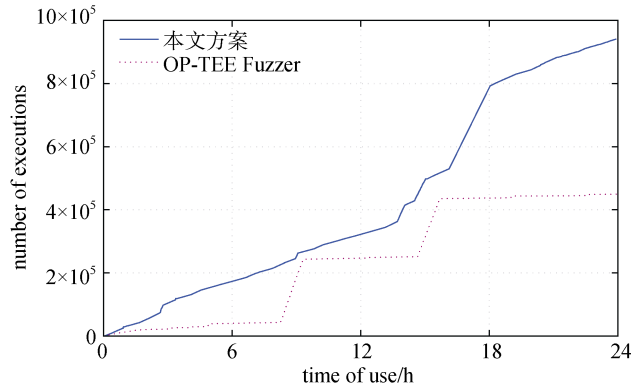


图 6 本文方案与 OP-TEE Fuzzer 执行用例数量随时间变化的对比

Figure 6 Comparison of the number of use cases executed over time between OP-TEE Fuzzer and our solution

4.4 路径发现速率与代码覆盖率分析

本文通过实验得到了本文方案与 OP-TEE Fuzzer 在 0~24h 的时间内路径发现的速率, 图 7 展示了本文方案与 OP-TEE Fuzzer 测试时间与路径发现数量的关系。

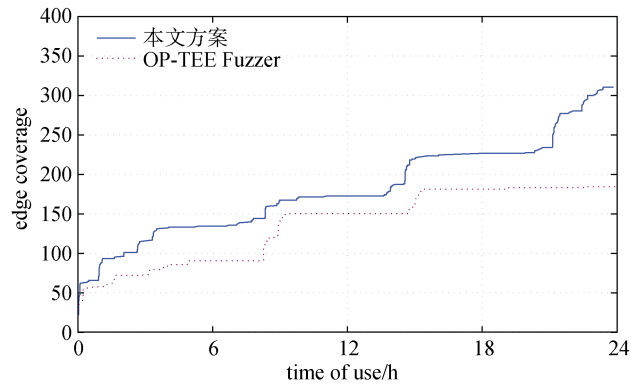


图 7 本文方案与 OP-TEE Fuzzer 路径发现数量随时间变化的对比

Figure 7 Comparison of the number of new paths over time between OP-TEE Fuzzer and our solution

从图 7 可见, 在从时刻 0h 开始的相同时间间隔内, 本文方案的路径发现数量始终高于 OP-TEE Fuzzer。经计算, 两者路径发现的平均速率如表 4 所示, OP-TEE Fuzzer 平均每小时发现 7.58 条新路径, 本文方案平均每小时发现 12.92 条新路径。在本次实验中, 本文方案路径发现平均速率高于 OP-TEE Fuzzer。

代码覆盖率通过分析本文方案和 OP-TEE Fuzzer 测试 24h 后的结果得到。代码覆盖率 cov 的计算采用基本块覆盖率的计算方式:

$$cov = \frac{num_{executed_BBs}}{num_{total_BBs}} \times 100\% \quad (7)$$

其中 $num_{executed_BBs}$ 为测试过程中 OP-TEE 内核被执行的基本块数量, num_{total_BBs} 为 OP-TEE 内核基本块总数。代码覆盖率分析结果如表 4 所示, 本次实验中, OP-TEE Fuzzer 的覆盖率为 13.04%, 本文方案的覆盖率为 17.03%。

表 4 OP-TEE Fuzzer 与本文方案测试 24h 代码覆盖率和路径发现速率对比

Table 4 Comparison of code coverage and new path discovery rate after fuzzing for 24-hours between OP-TEE Fuzzer and our solution

	路径发现平均速率	代码覆盖率
OP-TEE Fuzzer	7.58 条/h	13.04%
本文方案	12.92 条/h	17.03%

实验结果表明, 在相同环境对相同原始版本的 OP-TEE 进行模糊测试, 以及在种子选取相同的情况下, 本文方法的新路径发现速率 OP-TEE Fuzzer 的相比更高, 代码覆盖率本文方案比 OP-TEE Fuzzer 高 30.60%。由于本文方案以及 OP-TEE Fuzzer 均采用 AFL 的变异策略, 且发现的新路径不一定会导致代码覆盖率的变化, 本文方案代码覆盖率高于 OP-TEE Fuzzer 的结果符合预期。

4.5 漏洞发现能力评估

为了评估本文方案的漏洞发现能力, 本文从检验是否具有检出崩溃的能力以及检验是否具有发现能够导致崩溃或宕机现象发生的漏洞的能力两个角度开展实验。

为了检验本文方案是否具有检出崩溃的能力, 本文设计实现了两种能够模拟崩溃场景的系统调用, 并将它们预埋在与其他实验所使用的 OP-TEE 内核镜像不同的另一个镜像中, 然后通过本文方案对修改后的 OP-TEE 内核进行模糊测试, 最后通过分析实

验结果检验本文方案是否具有检出崩溃的能力。

预埋的两种系统调用分别模拟了触发 TA panic 时发生的崩溃场景, 称该场景为 TA panic 场景, 以及模拟触发了 OP-TEE 内核 panic 时发生的崩溃场景, 该类型崩溃会导致宕机, 称该场景为宕机场景。

对于上述两种模拟崩溃场景的模拟实现较为简单。首先在 OP-TEE 源码中选用多个未使用的系统调用号作为这两种新系统调用的系统调用号。选用多个系统调用号的目的是为了提高实验过程中对这两种系统调用的命中率。接着实现崩溃场景的系统调用过程。对于 TA panic 场景, 实现的系统调用过程内调用 syscall_panic 即可触发。对于宕机场景, 在实现的系统调用内执行无限循环或者调用 OP-TEE 内核的 panic 函数即可触发。最后将两种系统调用过程与系统调用号绑定。

本实验的结果为包含了第一种系统调用的测试用例被 AFL 当作崩溃存储在 crashes 文件夹中, 包含第二种系统调用的测试用例由于宕机超时, AFL 将其作为超时的测试用例存储在 hangs 文件夹中。实验结果表明, 本文方案具有检出崩溃的能力, 且能区分不同的崩溃场景。

为了检验是否具有发现漏洞的能力, 本文通过实验对 OP-TEE 进行数天的测试, 得到的崩溃和执行超时数量如表 5 所示。

表 5 实验过程发现的 crash 和 hang 数量
Table 5 The number of crashes and hangs found during the experiment

#crashes	#hangs	#unique crashes	#unique hangs
585	19,320	9	71

其中在测试过程中被 AFL 标记为 crash 的测试用例数量为 585 个, 被标记为 hang 的测试用例数量为 19,320, unique crash 数量为 9 个, unique hangs 数量为 71 个。

通过对标记为 unique crash 以及 unique hang 的测试用例分析, 可以发现以 OP-TEE 作为可信操作系统的 TEE 环境中, TEE 侧用户态与内核态的隔离并不像 REE 那样严格, TEE 侧用户态应用与内核态耦合程度高, 内核对于来自 TA 的输入校验较少。例如系统调用 syscall_cryp_random_number_generate, 其形参为 void *类型的 buf 和 size_t 类型的 blen, 该系统调用的功能是生成长度为 blen 的随机数并将结果保存在 buf 中。但该系统调用未对 blen 的合法性进行校验, 且由于缺乏 buf 指针所指向的类型, 无法确切知道 buf 指针指向的内存空间是否大于 blen。若

blen 在系统调用执行前遭到恶意攻击者修改使其大于 buf 指针指向内存空间的实际大小, 则 buf 所指向的内存空间的临近区域会被溢出的随机数内容改写从而遭到破坏。由于 buf 指向的可以由 TEE 侧 malloc 生成的内存空间, 因此该系统调用导致的溢出可以破坏整个 TEE 侧的堆, 影响系统的正常运行。目前该问题已提交开源社区处理。

通过实验结果结合结果分析可以得出, 本文方案能够有效地发现 OP-TEE 内核中潜在的安全问题, 本文方案具备 OP-TEE 内核漏洞发现的能力。

5 结论

本文提出了一种基于全系统模拟的 OP-TEE 模糊测试方法, 通过利用全系统模拟技术, 将 OP-TEE 托管于模拟环境并追踪 OP-TEE 执行过程, 并将模糊测试工具运行于模拟环境外以观测 OP-TEE 代码的执行过程。本文方案设计实现的基于全系统模拟的 OP-TEE 内核模糊测试方案, 解决了现有方案 OP-TEE Fuzzer 依赖源码、崩溃容忍度低的问题。

本文方案通过设计实现模拟环境内外通信组件, 将模拟环境内 OP-TEE 的系统调用暴露给模拟环境外的模糊测试工具, 使得模拟环境外的模糊测试工具能够对处于模拟环境内 TEE 侧的 OP-TEE 进行模糊测试, 同时通过设计实现预翻译优化机制降低了模糊测试过程中执行测试用例时由于重复翻译基本块所带来的性能损耗。

本文方案通过实验验证了本文方案的可行性和漏洞发现能力, 并与 OP-TEE Fuzzer 方案对比评估了实际性能。实验结果表明: (1) 本文方案具备漏洞发现能力; (2) 预翻译优化能够极大地提高 OP-TEE 内核已翻译基本块地占比, 减少重复翻译带来的性能损耗; (3) 与 OP-TEE Fuzzer 相比, 本文方案在执行导致宕机的测试用例时不会终止测试过程, 而是将其作为超时来处理; (4) 与 OP-TEE Fuzzer 相比, 模糊测试在性能上有较大的提升, 其中吞吐量提高了 104%。

本文工作仍然存在一定的局限性, 主要在于模糊测试的效率。虽然模糊测试的性能与 OP-TEE Fuzzer 相比要较大的提升, 但仍有较大提升空间。其中的一个原因是目前测试用例结构简单, 单个测试用例占用空间较大, 解析时间较长导致的。后续研究将逐步优化完善测试用例的结构, 减小测试用例的体积, 减少解析耗时, 进一步提升测试性能。此外该方案能检测的漏洞范围受限于能导致明显可观测的崩溃或宕机现象, 而对于更为隐蔽的漏洞则难以发

现, 后续研究将进一步研究其他类型漏洞的检测。

致 谢 本文工作受到国家自然科学基金项目(No. U1836112); 国家重点研发计划(No.2020YFB1805400); 国家自然科学基金项目(No.61876134)资助

参考文献

- [1] Sabt M, Achemlal M, Bouabdallah A. Trusted Execution Environment: What it Is, and what it is not[C]. *2015 IEEE Trustcom/ Big-DataSE/ISPA*, 2015: 57-64.
- [2] Ngabonziza B, Martin D, Bailey A, et al. TrustZone Explained: Architectural Features and Use Cases[C]. *2016 IEEE 2nd International Conference on Collaboration and Internet Computing*, 2017: 445-451.
- [3] Liang Cai. Guard Your Data With the Qualcomm Snapdragon Mobile Platform. <https://www.qualcomm.com/media/documents/files/guard-your-data-with-the-qualcomm-snapdragon-mobile-platform.pdf>. December, 2021.
- [4] Samsung. Samsung TEEgris. <https://developer.samsung.com/tee-gris/overview.html>. December, 2021.
- [5] Samsung. Samsung and Trustonic Launch Trustonic for KNOX, Delivering a Whole New Level of Trust Enhanced Experiences on Samsung Mobile Devices. <https://news.samsung.com/global/samsung-and-trustonic-launch-trustonic-for-knox-delivering-a-whole-new-level-of-trust-enhanced-experiences-on-samsung-mobile-devices>. December, 2021.
- [6] Linaro Limited. Open Portable Trusted Execution Environment. <https://www.op-tee.org/>. December, 2021.
- [7] Gal Beniamini. TrustZone Kernel Privilege Escalation (CVE-2016-2431). <http://bits-please.blogspot.com/2016/06/trustzone-kernel-privilege-escalation.html>. December, 2021.
- [8] NIST. CVE-2020-13831 Detail. <https://nvd.nist.gov/vuln/detail/CVE-2020-13831>. December, 2021.
- [9] NIST. CVE-2019-1010298 Detail. <https://nvd.nist.gov/vuln/detail/CVE-2019-1010298>. December, 2021.
- [10] Harrison L, Vijayakumar H, Padhye R, et al. PARTEMU: Enabling Dynamic Analysis of Real-World Trustzone Software Using Emulation[C]. *The 29th USENIX Conference on Security Symposium*, 2020: 789-806.
- [11] Ma J. Research on vulnerability mining and utilization method of ARM TrustZone[D]. Wuhan: Wuhan University. (马骏. ARM TrustZone 漏洞挖掘与利用方法研究[D]. 武汉: 武汉大学.)
- [12] Riscure. OP-TEE Fuzzer. https://github.com/Riscure/optee_fuzzer. December, 2021.
- [13] ARM Limited. Arm Architecture Reference Manual Armv8, for A-profile architecture. <https://developer.arm.com/documentation/ddi0487/gb/>. December, 2021.
- [14] Michal Zalewski. American Fuzzy Lop. <http://lcamtuf.coredump.cx/afl/>. December, 2021.
- [15] Bellard F. QEMU, a Fast and Portable Dynamic Translator[C]. *The annual conference on USENIX Annual Technical Conference*, 2005: 41.
- [16] ARM Limited. Learn the architecture: TrustZone for AArch64. <https://developer.arm.com/documentation/102418/latest>. December, 2021.
- [17] Binkert N, Beckmann B, Black G, et al. The Gem5 Simulator[J]. *ACM SIGARCH Computer Architecture News*, 2011, 39(2): 1-7.
- [18] NCC Group. TriforceAFL. <https://github.com/nccgroup/TriforceAFL>. December, 2021.
- [19] Schumilo S, Aschermann C, Gawlik R, et al. kAFL: Hardware-assisted feedback fuzzing for {OS} kernels[C]. *26th USENIX Security Symposium*. 2017: 167-182.
- [20] D. Vyukov. Syzkaller. <https://github.com/google/syzkaller>. December, 2021.
- [21] Zheng Y W, Davanian A, Yin H, et al. FIRM-AFL: High-Throughput Greybox Fuzzing of IoT Firmware via Augmented Process Emulation[C]. *The 28th USENIX Conference on Security Symposium*, 2019: 1099-1114.
- [22] D. D. Chen, M. Egele, M. Woo and D. Brumley, Towards automated dynamic analysis for Linux-based embedded firmware[J]. *Proc. Netw. Distrib. Syst. Security Symp*, 2016: 1-16.
- [23] Fábrega F J T, Javier F, Guttman J D. Copy on write[J]. 1995.
- [24] Linaro Limited. OP-TEE sanity test suite. https://github.com/OP-TEE/optee_test. December, 2021.
- [25] GlobalPlatform. TEE Internal Core API Specification. <https://globalplatform.org/specs-library/tee-internal-core-api-specification/>. December, 2021.
- [26] Linaro Limited. ARM Trusted Firmware. <https://www.linaro.org/engineering/projects/arm-trusted-firmware/>. December, 2021.



王丽娜 于 2001 年在东北大学获得博士学位。现任武汉大学国家网络安全学院教授, 博士生导师。研究领域为系统安全、多媒体信息隐藏、隐写分析理论和技术、网络安全、云安全及人工智能安全。Email: lnwang@whu.edu.cn



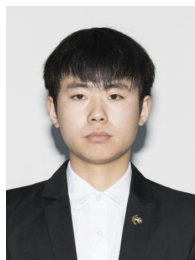
谢辉华 于 2019 年在武汉大学信息安全专业获得学士学位。现在武汉大学网络空间安全专业攻读硕士学位。研究领域为系统安全。研究兴趣包括: 人工智能。Email: x.h_h@qq.com



余荣威 于 2019 年在武汉大学信息安全专业获得博士学位。现任武汉大学国家网络安全学院副教授。研究领域为网络安全、密码学、系统安全。Email: roewe.yu@whu.edu.cn



张桐 于 2017 年在武汉科技大学信息安全专业获得学士学位。现在武汉大学网络空间安全专业攻读博士学位。研究领域为软件安全。研究兴趣包括软件安全、网络安全。Email: zhangtong2017@whu.edu.cn



赵敬昌 于 2016 年在哈尔滨工程大学信息安全专业获得学士学位, 现在武汉大学网络空间安全专业攻读硕士学位。研究领域为: 可信计算、漏洞挖掘。研究兴趣包括: 系统安全、软件安全。Email: kotwzjc@whu.edu.cn