# SiCsFuzzer: 基于稀疏插桩的闭源软件模糊测试方法

刘丽艳[1,5], 李 丰[2,3,4], 邹燕燕[2,3,4,5], 周建华[2,3,4,5], 朴爱花[2,3,4],
刘 峰[1,5], 霍 玮[2,3,4,5]

[1] 中国科学院信息工程研究所信息安全国家重点实验室, 北京 中国 100093
[2] 中国科学院信息工程研究所, 北京 中国 100093
[3] 中国科学院网络测评技术重点实验室, 北京 中国 100195
[4] 网络安全防护技术北京市重点实验室, 北京 中国 100195
[5] 中国科学院大学网络空间安全学院, 北京 中国 100049

**摘要** 传统的基于覆盖率反馈的模糊测试工具通过跟踪代码覆盖率来指导测试用例的变异, 从而发现目标程序中潜在的漏洞。但在闭源软件的模糊测试过程中, 跟踪覆盖率不仅带来额外的开销, 而且在模糊测试开销中占据主导。本文通过对 Windows 平台闭源软件模糊测试开销的剖析, 锁定其中两个主要来源, 插桩开销和“预热”开销。基于上述分析, 提出了一种基于稀疏插桩跟踪的模糊测试方法, 在不影响覆盖率计算精度的前提下, 采用基于稀疏插桩的跟踪策略, 仅对目标程序中覆盖率不可推导的基本块或分支进行插桩跟踪, 并根据跟踪结果推导其余基本块或分支的被覆盖情况; 同时结合“预热”优化, 避免因动态插桩平台反复启动以及对目标程序代码的重复翻译所引入的时间开销。基于上述方法实现的原型工具 SiCsFuzzer, 在 Windows 平台 9 个规模在 286KB~19.3MB, 类型涉及图片处理、视频处理、文件压缩、加密和文档处理等类型应用所组成的测试集上, 跟踪覆盖率引入的额外开销为程序正常执行时间的 1.1 倍, 比传统的基于覆盖率反馈的模糊测试工具快 3 倍, 并发现 PDFtk 和 XnView 程序最新版本中的未知漏洞各 1 个。

**关键词** 基于覆盖率反馈的模糊测试; 基于稀疏插桩的跟踪方法; “预热”优化
**中图法分类号** TP311 **DOI 号** 10.19363/J.cnki.cn10-1380/tn.2022.07.05

# SiCsFuzzer: A Sparse-instrumentation-based Fuzzing Platform for Closed Source Software

LIU Liyan[1,5], LI Feng[2,3,4], ZOU Yanyan[2,3,4,5], ZHOU Jianhua[2,3,4,5], PIAO Aihua[2,3,4],
LIU Feng[1,5], HUO Wei[2,3,4,5]

[1] State Key Laboratory of Information Security, Institute of Information Engineering, Chinese Academy of Science, Beijing 100093, China.
[2] Institute of Information Engineering, Chinese Academy of Sciences, Beijing 100093, China
[3] Key Laboratory of Network Assessment Technology, Chinese Academy of Sciences Beijing 100195, China
[4] Beijing Key Laboratory of Network security and Protection Technology, Beijing 100195, China
[5] School of Cyber Security, University of Chinese Academy of Sciences, Beijing 100049, China

**Abstract** Traditional coverage-guided fuzzing tools use code coverage tracing to guide test case mutation so that they could explore previously unseen code regions and trigger potential vulnerabilities in them more efficiently. However, during the fuzzing process of a close source software, code coverage tracing is time consuming and it is a dominant source of overhead. In this paper, we made a detailed analysis of the overhead of the coverage-guided fuzzing and our analysis shows that the overhead mainly comes from two parts: (1) the time spent on program instrumentation and (2) the expense incurred by “warm-up”. Based on the observation, we propose a sparse-instrumentation-based fuzzing approach which leverages a sparse-instrumentation-based tracing strategy without sacrificing the accuracy of coverage computing during fuzzing. The key idea of our approach is instrumenting only blocks or edges whose coverage cannot be implied by others and using their coverage to imply whether those un-instrumented blocks are executed or not. We also implement a warm-up optimal to discard the time cost of re-initializing the dynamic binary instrumentation framework and that of re-generating the same code snippet of the target program during fuzzing. We implement a prototype tool SiCsFuzzer based on the above approach. Evaluation shows that for nine real-world closed source binaries on Windows varying in size from 286KB to 19.3MB and types involving image processing, audio processing,

**通讯作者:** 李丰, 博士, 副研究员, lifeng@iie.ac.cn。
本课题得到国家自然基金(No. U1836209, No. 61602470)和重点研发计划(No. 2016QY071405)资助。
收稿日期: 2019-12-24; 修改日期: 2020-03-10; 定稿日期: 2022-05-11

data archiving, cryptography and document processing, SiCsFuzzer incurs an average overhead of 1.1 times compared to native execution, which is 3 times faster than traditional coverage-guided fuzzing tools and found a vulnerability in the latest versions of Windows platform close source software PDFtk and XnView, respectively.

**Key words** coverage-guided fuzzing; sparse-instrumentation-based tracing; warm-up optimization.

## 1 引言

模糊测试是目前发现软件漏洞的最有效方法之一。它通过向目标程序提供大量的经过特殊构造的测试用例, 并在程序运行过程中监控程序的执行行为, 以程序是否发生异常行为为标志, 来发现目标程序可能存在的安全漏洞。为了快速发现软件潜在的漏洞, 以 AFL[1]为代表的模糊测试工具[2-4]通过跟踪覆盖率来指导测试用例的变异, 力求变异生成的测试用例能够覆盖到目标程序中更多的未测代码, 从而更有效的发现目标程序中潜在的漏洞。上述模糊测试方法被称为基于覆盖率反馈的模糊测试方法。

给定目标程序 P 以及一系列种子输入 I, 目前主流的基于覆盖率反馈的模糊测试工具, 其工作流程可以分为以下三个步骤: (1)变异: 通过位翻转、删除、替换等一系列的策略对种子输入进行变异, 生成大量新的输入; (2)跟踪与反馈: 将新生成的输入交由目标程序执行, 通过程序插桩等技术跟踪目标程序在该输入下的代码覆盖率信息(如: 节点覆盖率、边覆盖率等), 并根据所反馈的覆盖率变化情况, 对于覆盖了新路径的输入, 在后续的变异过程中赋给其更高的优先级; (3)监控: 监控程序执行过程中的异常行为(如: 程序崩溃等)以及触发异常的输入, 作为漏洞分析及定位的依据。由此可见, 跟踪测试用例的代码覆盖率是决定模糊测试效能的关键环节之一。

然而, 跟踪测试用例的代码覆盖率也会为模糊测试引入额外的性能开销, 甚至成为模糊测试性能开销的主要来源。对于开源程序, 可通过编译时插桩插入用于在运行时获取覆盖信息的代码。但对于闭源程序, 尤其是 Windows 平台的二进制程序, 需要通过动态插桩[5]、二进制重写[6]或硬件辅助[7-8]的方法来追踪覆盖率信息。其中, 基于硬件辅助的方法几乎不引入额外开销, 但依赖于平台硬件支持, 不易扩展。最近的工作[9]表明, 对于 Linux 平台上的闭源目标程序, AFL 跟踪代码覆盖率的开销高达 13 倍。本文的实验分析(详见第 2 章和第 5 章)也表明, 对于 Windows 平台上的二进制程序, 跟踪覆盖率的开销是程序正常运行的 3~18.9 倍, 平均占执行单个测试用例时间的 82%。

相关工作也就模糊测试过程中跟踪覆盖率开销的缓解提出了改进方案。代表性方案 UnTracer[9]借助于轻量级检测和采用基于静态二进制重写的覆盖率跟踪技术, 在所选数据集上仅相对程序正常执行时间引入了 0.3%的额外开销。其采用的轻量级检测避免了对没有价值的种子进行覆盖率跟踪, 但是对于有价值的种子, UnTracer 仍然面对跟踪覆盖率的开销, 而其使用的基于静态二进制重写的覆盖率跟踪技术在 Windows 平台上缺乏鲁棒性, 无法扩展到 Windows 平台闭源软件的模糊测试中。新进提出的二进制重写技术 RetroWrite[10], 其速度是 QEMU 的 4.5 倍, 但仅适用于 64 位 Linux 平台上的地址无关代码。采用二进制动态跟踪代码覆盖率虽然开销较高, 但确是目前最具普适性的方法。

本文针对闭源软件(尤其是 Windows 平台上闭源软件)模糊测试过程中因跟踪代码覆盖率引入的额外时间开销问题, 提出了一种基于稀疏插桩跟踪的模糊测试方法, 在不影响覆盖率计算精度的前提下, 采用基于稀疏插桩的跟踪策略, 仅对目标程序中覆盖率不可推导的基本块或分支进行插桩跟踪, 并根据跟踪结果推导其余基本块或分支的被覆盖情况, 同时结合“预热”优化, 避免因动态插桩平台反复启动以及对目标程序代码的重复翻译所引入的时间开销。通过剔除闭源软件覆盖率跟踪过程中的冗余开销, 提高闭源程序模糊测试的效率, 进而提高针对闭源程序的漏洞发现效率。

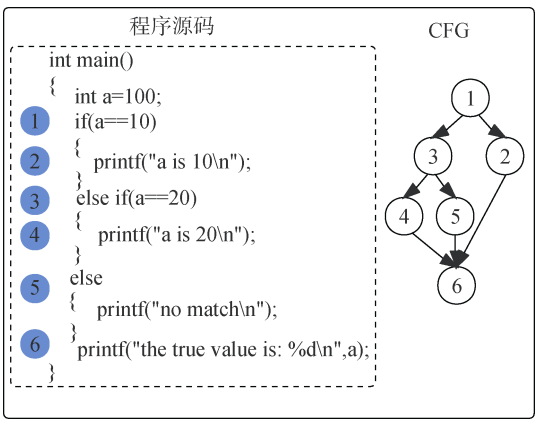



**图 1 示例程序和程序控制流图**

**Figure 1 A sample program and its CFG**

其中, 基于稀疏插桩的覆盖率跟踪策略以传统程序分析技术的支配关系作为插桩位置选择以及覆

盖率推导的基础。目前主流的基于覆盖率反馈的模糊测试工具默认对目标程序中的所有基本块或分支进行插桩。但并非所有插桩位置都是必要的。以图 1 所示的程序片段及其对应的控制流图(CFG)为例。根据后支配关系, 如果测试用例覆盖了基本块 6, 则必然也覆盖了基本块 1。换而言之, 基本块 1 是否被覆盖可以通过对基本块 6 的插桩跟踪结果推导出来。因此, 在对基本块 6 进行插桩的同时, 再对基本块 1 插桩是冗余的。文献[11]所述的程序优化技术也表明, 可以找到目标程序所有基本块/分支的一个子集, 该子集中的基本块/分支可以推导出目标程序控制流图上的所有基本块/分支。文献[11]的实验也表明, 对于选择的 8 个 C 程序, 只需要覆盖目标程序的特定的 32%的分支, 即可确保 100%的分支覆盖。本文第 5 章的实验也显示, 平均只需要对目标程序的 40.61%的分支进行插桩, 就可以推导出目标程序中其他分支是否被覆盖。可见, 通过支配关系指导的稀疏插桩, 可以有效的减少不必要的插桩与追踪。本文在文献[11]工作的基础上, 扩展了对二进制代码分析以及动态运行时插桩的支持, 以适应闭源程序的模糊测试需求。

如前所述, 对闭源程序的覆盖率跟踪可以采取动态二进制插桩或基于硬件辅助的技术。其中, 动态二进制插桩相对基于硬件辅助的技术可以适应更多的平台, 因此具有更强的鲁棒性。代表性的模糊测试工具 AFL 和 Vuzzer[12]分别使用 QEMU 和 Pin[13]对闭源程序进行插桩。为使所提出的方法及对应工具能够适应更多的平台, 尤其是支持对 Windows 平台上闭源程序的高效模糊测试, 本文也采用动态二进制插桩技术跟踪代码覆盖率。但文献[9]的统计显示, QEMU 引入的额外开销约为目标程序正常执行时间的 10 倍。本文的实验结果也表明, Pin 空载所引入的额外开销是程序正常运行时间的 2 倍以上, 其中 52%的开销来源于动态代码翻译。由于动态插桩平台每执行一个测试用例, 都需要对目标程序重新进行动态代码翻译, 由此带来冗余的开销。“预热”优化的引入可以使先前生成的目标程序代码被所有测试用例共享, 从而避免上述冗余开销。

基于上述方法实现的基于稀疏插桩的闭源软件模糊测试原型 SiCsFuzzer。对于 Windows 平台上的 9 个黑盒二进制目标程序, 在不影响覆盖率计算精度的前提下, SiCsFuzzer 只需对目标程序中 40.61%的分支进行插桩, 最终 SiCsFuzzer 的执行效率比传统的基于覆盖率反馈的模糊测试工具快 3 倍。

本文的主要贡献如下:

1) 提出了一种面向闭源程序的高效模糊测试方法。该方法借助基于稀疏插桩的覆盖率跟踪以及针对流行的动态二进制插桩工具(例如 Pin)的“预热”优化, 降低基于覆盖率反馈的模糊测试过程中跟踪覆盖率的开销, 提高模糊测试效率。

2) 基于该解决方案实现的原型工具 SiCsFuzzer, 在 Windows 平台 9 个规模在 286KB~19.3MB, 类型涉及图片处理、视频处理、文件压缩、加密和文档处理等类型应用所组成的测试集上, 跟踪覆盖率引入的额外开销为程序正常执行时间的 1.1 倍, 比传统的基于覆盖率反馈的模糊测试工具快 3 倍, 并发现 PDFtk 和 XnView 程序最新版本中的未知漏洞各 1。

本文的组织结构如下: 第 2 章介绍基于覆盖率反馈的模糊测试方法的工作流程, 剖析性能开销分布, 并提出本文的改进思路; 第 3 章阐述基于稀疏插桩的闭源软件模糊测试方法的系统设计及其中涉及的两个关键技术的细节; 第 4 章介绍基于稀疏插桩的闭源软件模糊测试原型 SiCsFuzzer 的实现细节; 第 5 章分析实验数据; 第 6 章介绍相关工作; 第 7 章总结全文。

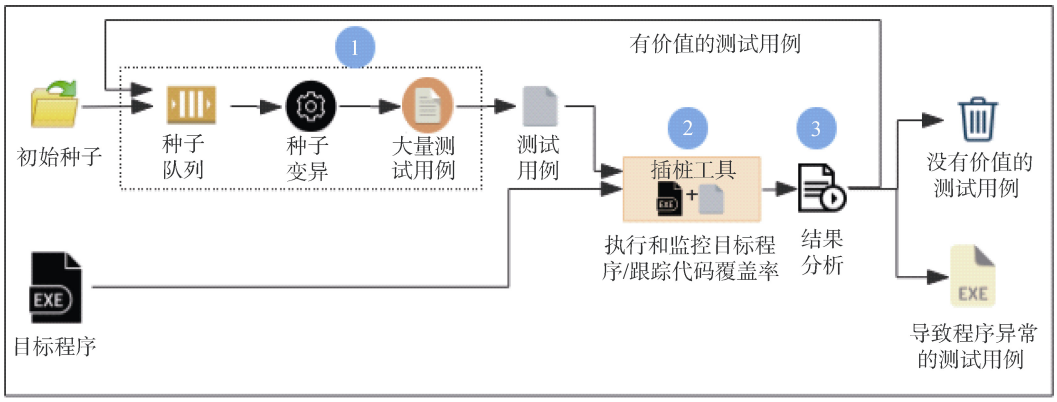



**图 2 传统基于覆盖率反馈的模糊测试工作流程**

**Figure 2 The workflow of traditional coverage-guided fuzzing**

## 2 基于覆盖率反馈的模糊测试性能开销

本章将依次介绍基于覆盖率反馈的模糊测试方法的工作流程, 剖析性能开销分布, 并提出改进思路。

### 2.1 基于覆盖率反馈的模糊测试方法

图 2 展示了传统的基于覆盖率反馈的模糊测试(以下简称模糊测试)的工作流程, 主要包括三个重要的部分: (1)测试用例的生成; (2)执行和监控目标程序/跟踪覆盖率; (3)覆盖率反馈和崩溃结果分析。在模糊测试运行前, 对于一个目标程序, 给定一个目标程序和一系列初始种子, 当模糊测试运行时, 首先, 模糊测试工具将初始种子加入到种子队列中, 按照特定的顺序从种子队列中选取一个种子并按照一系列的变异规则对种子进行变异, 生成大量新的测试用例, 并依次提供给目标程序。接着, 目标程序在执行每一个测试用例的过程中, 模糊测试工具通过静态或动态插桩的方式对目标程序进行插桩, 来跟踪代码覆盖率。如果一个测试用例能够探索到目标程序新的代码区域, 比如新的基本块、新的分支, 则该测试用例被认为是有价值的, 并被保存到初始种子队列中, 以备将来之用。如果该测试用例是无价值的, 则被丢弃。总之, 基于覆盖率反馈的模糊测试工具将目标程序执行测试用例时得到的覆盖率作为反馈信息, 来指导种子的选择和变异。以这种方法得到的测试用例更有可能探索到未被发现的目标程序代码区域, 并触发其中的漏洞。因此, 准确的代码覆盖率信息是模糊测试工具发现有价值的种子并快速发现软件漏洞的重要依据[14]。

当前主流的模糊测试工具多使用分支覆盖来计算测试用例对目标程序的代码覆盖率。以 AFL 为例。AFL 使用一个位图结构记录分支覆盖率, 并为每个分支计算一个哈希值, 作为其在位图中的键值。该哈希值由分支的起始基本块地址与结束基本块地址通过移位及异或的方式计算得到。在不考虑哈希碰撞的前提下, 每个分支对应的哈希值是唯一的。由于本文重点关注闭源程序, 尤其是以往工作普遍忽视的 Windows 平台上闭源程序模糊测试的效率优化, 本章的后续小节将针对 Windows 程序基于分支覆盖反馈的模糊测试时间开销进行剖析, 并根据剖析结果阐述本文的效率改进方案。

### 2.2 性能剖析

本节选取 Windows 平台上规模在 286KB~19.3MB, 类型涉及图片处理、视频处理和文件压缩的 9 个目标程序, 剖析模糊测试的性能构成。实验选择动态插桩工具 Pin 作为追踪和反馈分析覆盖的基础平台。

图 3 所示为模糊测试的一次迭代[15]过程中, 跟踪覆盖率的执行时间的占比。通过对 8 个目标程序的统计可以看出, 跟踪覆盖率(对应图 2 的第 2 部分)的平均执行时间占模糊测试的一次迭代总执行时间的 82%, 是模糊测试开销的主要来源。图 4 进一步剖析了跟踪覆盖率的时间开销构成。从中可以看出, 跟踪覆盖率所引入的额外开销为程序正常执行时间的 5.25 倍。上述额外的开销主要来自两个方面: (1)二进制插桩平台 Pin 的开销, 包括启动和初始化 Pin、加载和启动目标程序、在 JIT 模式下对目标程序代码进行动态翻译; (2)插桩开销, 包括用于确定在目标程序的哪些位置插入跟踪代码的开销和执行跟踪代码的开销。如图 4 所示, 上述两部分的开销分别占跟踪测试用例覆盖率开销的 60%和 24%。

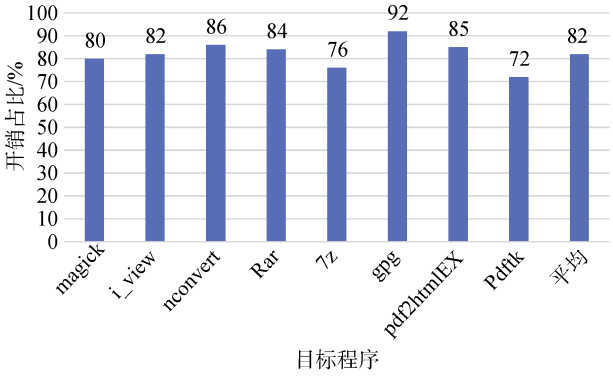



**图 3 跟踪覆盖率在单次模糊测试中占比**

**Figure 3 The percentage of tracing coverage in fuzzing**

图 5 进一步剖析了二进制插桩平台 Pin 的开销构成。从中可以看出, Pin 的空载执行时间(即使用 Pin 控制目标程序运行, 但不对目标程序进行插桩的执行时间)主要由三部分构成: (1)平台初始化和目标程序加载的开销; (2)动态翻译开销; (3)执行翻译后的代码的开销。本文将平台初始化和加载目标程序, 以及对目标程序代码的动态翻译这两部分统称为“预热”阶段。从图 5 可以看到, “预热”阶段的开销是 Pin 执行时间的主要组成部分, 其中初始化和加载过程占 Pin 执行时间的 33%, 动态代码生成占 52%。

上述实验表明, 跟踪覆盖率的时间开销是模糊测试开销的主要来源(82%), 而跟踪覆盖率相对于程序正常执行所引入的额外开销主要源于 Pin 的“预热”阶段的开销以及插桩开销。如何避免和缓解上述开销是本文工作的重点。

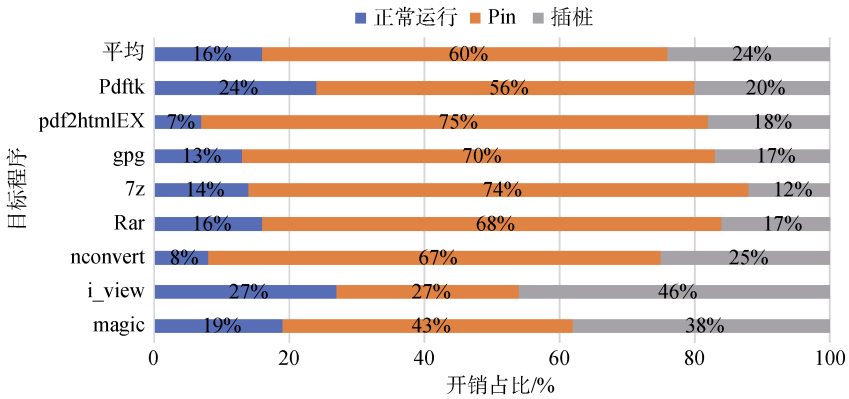



**图 4　跟踪覆盖率开销组成**

**Figure 4　Cost breakdown of Tracing**

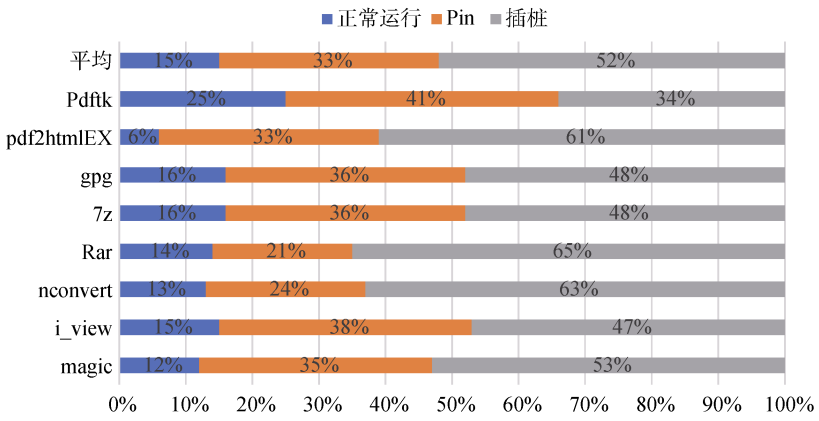



**图 5　Pin 空载执行时间**

**Figure 5　The overhead occurred by Pin**

## 2.3　改进方案

基于以上实验分析，我们发现基于动态插桩的跟踪覆盖率是基于覆盖率反馈的模糊测试的主要开销来源。为了降低跟踪覆盖率的开销，应该考虑两个关键因素: (1)插桩的开销; (2)Pin 的“预热”阶段的开销。接下来，本文将分别介绍本文降低跟踪覆盖率开销的关键思路。

**降低插桩的开销**。目前主流的基于覆盖率反馈的模糊测试工具默认对目标程序中的所有基本块或分支进行插桩。但并非所有插桩位置都是必要的。以图 1 所示的程序片段及其对应的控制流图(CFG)为例，对于该程序，为了跟踪覆盖率，只需要对 BB2(基本块 2)、BB4 和 BB5 进行插桩，就可以区分每一条执行路径，并且能够推导出准确的覆盖率信息。文献[11]所述的程序优化技术也表明，可以找到目标程序所有基本块/分支的一个子集，该子集中的基本块/分支可以推导出目标程序控制流图上的所有基本块/分支。文献[11]的实验也表明，对于选择的 8 个 C 程序，只需要覆盖目标程序中特定的 32%的分支，即可确保 100%的分支覆盖。受到该工作的启发，本文将插桩对象限定在目标程序中的一部分基本块或分支，这些基本块或分支运行与否无法通过程序中其它基本块或分支的运行信息进行推断。这样，可以在不牺牲代码覆盖率准确性的情况下减少插桩的开销。基于该思想，我们扩展文献[11]中的工作，以支持闭源软件代码分析和动态插桩。在本文接下来的部分称其为基于稀疏插桩的覆盖率跟踪技术。

**降低 Pin 的“预热”阶段的开销**。由于动态插桩平台每执行一个测试用例，都需要重新加载目标程序并对目标程序重新进行动态代码翻译，由此带来冗余的开销。本文引入“预热”优化，通过避免对同一程序的重复加载提高对 CodeCache 的复用率，从而避免重复加载和重复翻译的开销。

# 3　系统设计与技术细节

## 3.1　系统设计

基于 2.3 介绍的改进方案，本文设计实现了

SiCsFuzzer(A Sparse-instrumentation-based Fuzzing Platform for Closed Source Software)。SiCsFuzzer 系统设计如图 6 所示。与图 2 所示的传统的基于覆盖率反馈的模糊测试相比, SiCsFuzzer 对图 2 中的第 2 阶段, 即执行和监控目标程序/跟踪覆盖率的阶段进行了以下两方面改进。

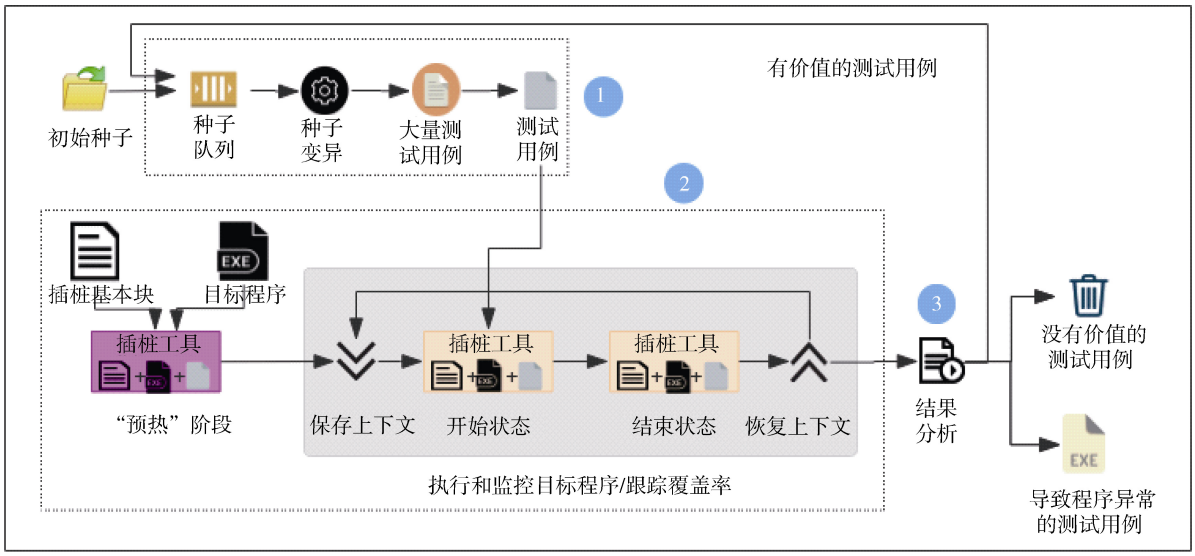



**图 6 SiCsFuzzer 系统设计**

**Figure 6 Design of SiCsFuzzer**

1) 提出基于稀疏插桩的覆盖率跟踪技术, 根据支配关系选择必要的插桩位置, 仅对覆盖率无法推导的分支或基本块进行稀疏插桩, 从而在不影响覆盖率准确性的情况下降低插桩的开销。基本思路是在对闭源目标程序进行模糊测试之前, 通过二进制代码逆向分析和控制流分析, 构建全局控制流图以及全局超级块支配图, 计算基本块之间的前/后支配关系, 从而在全局控制流图上标记出能够区分不同路径并且能够推导出其它基本块/边的覆盖情况的最小基本块/边的集合, 用于指导动态二进制插桩过程中位置的选择以及模糊测试一次迭代过程中覆盖率的推导和重建, 由此达到降低插桩代价, 提高模糊测试效率的目的。

2) 提出“预热”优化, 避免二进制插桩平台重复启动以及对目标程序热代码的重复翻译的开销, 从而进一步提高模糊测试的效率。与图 2 所示的传统模糊测试流程相比, SiCsFuzzer 将插桩平台的“预热阶段”从模糊测试的循环迭代中分离出来, 仅在模糊测试开始时进行插桩平台的初始化及被测目标程序的加载。在剔除重复启动、加载开销的同时, 使测试用例之间可以共享已被动态翻译的代码, 以缓解冗余动态翻译的开销。基本思路是在目标程序加载后, 执行到输入读取位置之前的某一程序点 p 时, 保存内存快照, 当目标程序读取输入并执行到指定的结束位置时, 使用保存的快照将程序执行状态恢复到程序点 p, 并读取新的输入。

## 3.2 技术细节

### 3.2.1 基于稀疏插桩的覆盖率跟踪技术

本节将介绍基于稀疏插桩的覆盖率跟踪技术的技术细节。

如前所述, 稀疏插桩的基础是标记程序控制流图上满足具备路径可区分性以及覆盖率的推导重建能力的基本块。路径可区分性是指通过被标记的基本块可以区分每一条不同的执行路径。也就是说, 对于任意两条不同的路径, 由路径上被标记的基本块构成序列是不同的。如图 1 所示, 如果被标记的基本块为 BB2、BB4 和 BB5, 则经过 BB1→BB3→BB4→BB6 的执行路径将被表示成[BB4], 经过 BB1→BB3→BB5→BB6 的执行路径被表示为[BB5], 经过 BB1→BB2→BB6 的执行路径被表示为[BB2]。换而言之, 只需要对这三个基本块进行插桩, 就可以区分出不同的执行路径。代码覆盖率的推导重建能力是指根据所跟踪到的被标记基本块, 能够推导出当前测试用例所执行的路径上其它未被标记的基本块, 即重建整条路径对应的代码覆盖率。仍以图 1 为例, 如果当前测试用例覆盖了被标记的基本块 BB4, 则可以推导出当前测试用例覆盖了图 1 代码片段中的路径 BB1→BB3→BB4→BB6。

本文参照并扩展了文献[11]和文献[16]所述方法, 将稀疏插桩位置的选择问题转化为全局超级块支配

图上的顶点(即基本块)标注问题。以图 7 所示的程序代码为例。采用文献[11]所述方法可以构建出各个函数的超级块支配图。构建流程如图 7~10 所示。即: 首先构建各个函数的控制流图(图 7), 并基于控制流图计算前支配树和后支配树(图 8), 然后合并前、后支配树中相同的节点, 生成如图 9 所示的基本块支配图, 再规约图中的强联通分量并去除冗余边后, 得到图 10 中所示的每个函数的超级块支配图。

超级块支配图具备如下性质: (1)如果一个测试用例覆盖了超级块支配图中的某个超级块, 则该超级块中包含的所有基本块也都被覆盖; (2)如果一个测试用例的执行路径覆盖了超级块 U 的一个子超级块 V, 则该测试用例也覆盖了 U。将上述性质应用在稀疏插桩的位置选择上, 即只需要标记超级块支配图上子节点数量少于两个的超级块中包含的任意一个基本块, 如图 10, 对于 main 函数, 只需要对基本块 2、4、5 插桩, 对于 display 函数, 只需要对基本块 8、10 插桩。通过动态插桩记录上述基本块集合是否被覆盖, 则可推导出同一函数中其它基本块的覆盖情况。

但由于单个函数的超级块支配图并未考虑函数调用对支配关系的影响, 基于目标程序中每个函数的超级块支配图所标记出的最小稀疏插桩位置集合有可能冗余, 甚至是不正确的。以图 7 所示的程序为例, 如果一个测试用例覆盖了 display 函数的基本块 8, 则可以推断出该测试用例同时覆盖了 main 函数中的基本块 1, 2, 6, 也就是说, 如果对基本块 8 插桩, 则不需要对基本块 2 插桩。另外, 当程序中出现 abort、exit 等函数时, 会影响支配关系的计算, 例如, 仍以图 7 为例, 若基本块 8 是一个 exit 函数, 当一个测试用例覆盖了基本块 8, 由函数调用关系可知, 该测试用例一定覆盖了基本块 2, 根据图 10 的 main 函数的超级支配图推导出, 如果测试用例覆盖了基本块 2, 则该测试用例一定覆盖了基本块 1, 6。显然, 该测试用例不会覆盖基本块 6。因此为了对程序的全局控制流进行分析, 本文实现了文献[17]提出的全局超级块支配图理论, 根据函数的调用关系, 并结合每个函数的超级块支配图, 得到全局超级块支配图, 如图 11 所示。最后, 本文将上述超级块支配图的性质应用到插桩位置选择, 根据全局超级块支配图(图 11), 只需要对基本块 4、5、8 和 10 进行插桩。并且基于全局超级支配图中, 本文实现了文献[17]中提出的方法解决了函数调用过程中不能正常返回的情况。

**算法 1.**基于稀疏插桩的跟踪算法.

输入: 目标程序和被标记的基本块

输出: 无

1: FUNCTION trace_instrument(trace)
2: FOR 基本块 $\varepsilon$ trace DO
3: IF 当前基本块是被标记的基本块 THEN
4: 在当前基本块前插入跟踪代码
5: END IF
6: END FOR
7: END FUNCTION

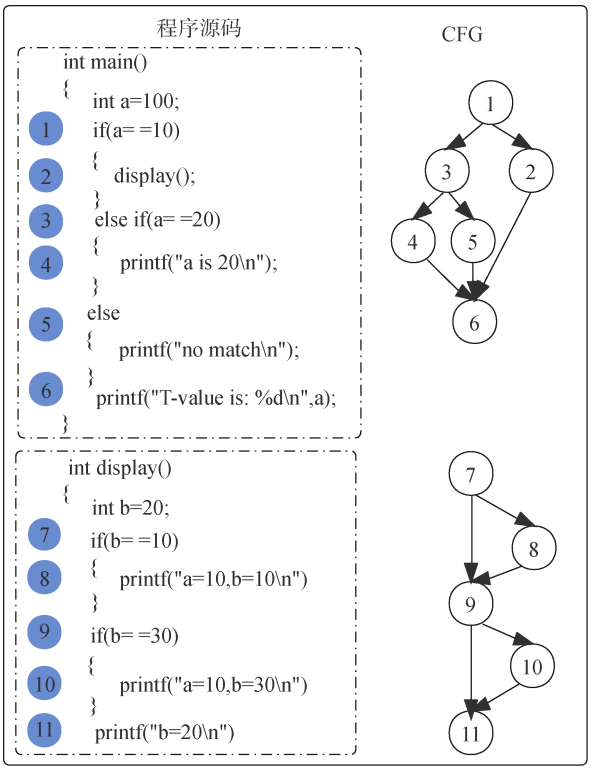



**图 7 示例程序**

**Figure 7 Sample programs**

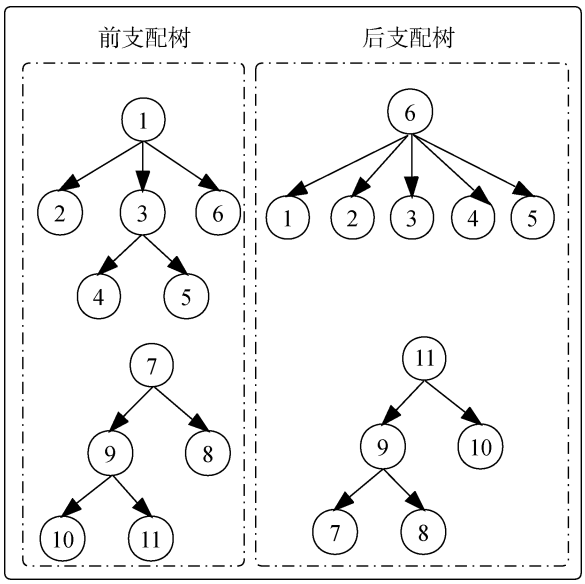



**图 8 前支配树和后支配树**

**Figure 8 Pre-dominator tree and post-dominator tree**

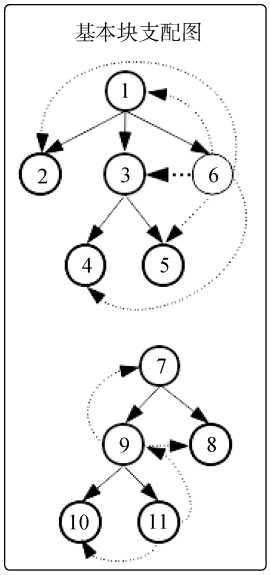



**图 9　基本块支配图**

**Figure 9　Basic block dominator graph**

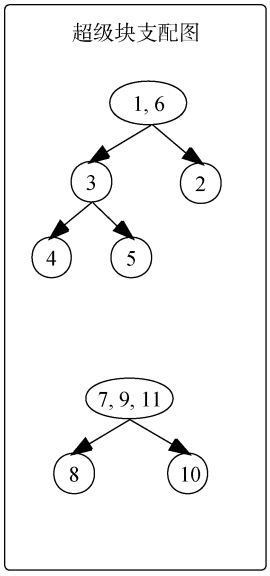



**图 10　超级块支配图**

**Figure 10　Super block dominator graph**

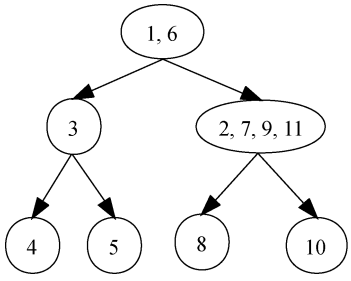



**图 11　全局超级块支配图**

**Figure 11　Global super dominator graph**

如算法 1, 当 SiCsFuzzer 在跟踪覆盖率时, 插桩工具取目标程序将要执行的一个顺序指令序列(记做 trace), 并判断 trace 中每个基本块是否被标记。如果被标记, 则在该基本块前插入跟踪代码。跟踪代码的功能是维护一个 bitmap 数据结构。为了重建代码覆盖率, 在跟踪代码中, 我们额外使用另外一个与 bitmap 大小相同的 bitmapAddr 数据结构记录被覆盖的基本块/分支的地址。比如如果覆盖了基本块 1,则会根据该基本块地址计算得到 bitmap 数组的下标, 相应地, 以该下标的 bitmap 的值代表对应的基本/分支块的被覆盖次数, 同时以该下标的 bitmapAddr 的值为该基本块/分支的地址。

如前所述, 基于稀疏插桩可以推导出其它未被插桩的基本块及边的覆盖率。算法 2 为覆盖率重建算法。虽然覆盖率重建对本文方法而言并不是必须的, 但考虑到与主流模糊测试工具之间的兼容性, 重建覆盖率可以更好的辅助主流模糊测试工具进行变异策略的选择。比如, AFL 会根据当前测试用例覆盖的基本块或边的数量选择变异次数, LibFuzzer 会给予覆盖了更多新的基本块的种子更高的优先级。在重建覆盖率的基础上能够更加直观的计算出这些信息, 从而提高本文方法的适用面。

**算法 2**. 代码覆盖率重建算法

输入: bitmapAddr 和 marked。

输出: 代码覆盖率

```
1:FUNCTION getCoverage (bitmapAddr,marked)
2: cur_coverage = 0
3: bbl_cover = {}
4: FOR 被覆盖的基本块 IN bitmapAddr DO
5: IF 被覆盖的基本块 NOT IN bbl_cover
6: bbl_cover[bbl_addr]=1
7: cur_coverage+=1
8: FOR 未插桩的基本块 IN marked[bbl_addr] DO
9: IF 未插桩的基本块 NOT IN bbl_cover
10: bbl_cover[unmark _addr]=1
11: cur_coverage+=1
12: END IF
13: END FOR
14: END IF
15: END FOR
16: RETURN cur_coverage
17: END FUNCTION
```

如算法 2 所示, 根据 bitmapAddr 和 marked 即可重建覆盖率。根据目标程序的全局超级块支配图, 我们得到被标记的基本块。对于每一个标记的基本块, 我们递归地记录它的所有父节点, 也就是与该标记的基本块同时被覆盖的未被标记的基本块, 记为 marked, 以图 11 为例, 其数据结构为: {4: 1, 3, 6; 5: 1,

3, 6; 8: 1, 2, 6, 7, 9, 11; 10: 1, 2, 6, 7, 9, 11}。另外, bitmapAddr 为算法 1 中提到的记录动态跟踪时被覆盖的被标记的基本块/分支的地址。在算法 2 中, 我们使用一个哈希表 bbl_cover 存储当前测试用例覆盖到的基本块/分支的地址, 首先遍历 bitmapAddr 中的每一个值, 如果该值是当前测试用例覆盖的基本块/分支地址, 则覆盖率增加 1, 并存储到哈希表中(第 5~7 行), 然后遍历该基本块/分支能推导出的其他基本块/分支, 同样如果该基本块/分支的地址不在 bbl_cover 哈希表中, 则将其存储到哈希表中, 并将覆盖率增加 1(第 8~13)。最终, 我们通过稀疏插桩的覆盖率跟踪方法, 实现了代码覆盖率重建。

#### 3.2.2 “预热”优化技术

本节介绍“预热”优化技术的细节。以 Pin 为代表的二进制动态插桩平台在执行到目标程序的一个新的基本块时, 会将该基本块转换成 Pin 的可执行指令序列存储到 CodeCache 当中; 如果 CodeCache 当中已经保存了该基本块对应的指令序列, 则无需再次翻译。一旦插桩平台重启或程序重新载入, 则 CodeCache 将被清空。“预热”优化的基本思路是通过避免对同一程序的重复加载提高对 CodeCache 的复用率, 从而避免重复加载和重复翻译的开销。该优化得以实施的基础是能够在目标程序加载后, 执行到输入读取位置之前的某一程序点 p 时, 保存内存快照, 当目标程序读取输入并执行到指定的结束位置时, 使用保存的快照将程序执行状态恢复到程序点 p, 并读取新的输入。

SiCsFuzzer 默认选择 main 函数的开始位置作为快照的生成和恢复位置。快照保存的内容包括寄存器状态、内存状态。如果测试用例是通过 main 函数的参数读入的, 则额外监控测试用例的存放位置 L, 并在每次恢复快照的同时将从种子队列中读取的新输入复制到 L。SiCsFuzzer 默认选择的结束位置为 main 函数的结束位置; 也可以选择在调用 exit 函数的位置; 对于有明显输出信息的程序, 可以通过动态调试并观察输出状态选择出结束位置。

这样, 我们可以只关注目标程序的代码部分, 而不是花费大量的时间用于初始化插桩工具和加载目标程序。同时, “预热”优化可以充分发挥插桩工具的“Trace Linking”优化机制, 避免重复将相同的 trace 生成新的指令代码的开销, 使得之前生成的指令代码存储在 CodeCache 中被执行所有测试用例时共享。并且随着模糊测试的进行, 被覆盖到的新的 trace 越来越少, 其中大多数已经存储在 CodeCache 中, 极大地降低了不必要的开销。

## 4 系统设计实现

本文基于上述设计, 在现有模糊测试平台 Puzzer[18]上实现了基于稀疏插桩的闭源软件模糊测试原型 SiCsFuzzer。本章将介绍原型中两个关键技术的实现细节。

### 4.1 基于稀疏插桩的覆盖率跟踪实现

基于稀疏插桩的覆盖率跟踪实现分为基本块标记、插桩追踪和覆盖率重建三部分。

其中, 基本块标记实现为 IDA Pro 插件的形式, 利用 IDA Pro 提供的接口提取目标程序的函数调用关系和每个函数的程序控制流图; 然后使用 Boost 图形库函数建立程序的前支配树和后支配树。如果模糊测试使用基本块覆盖, 则计算基本块的前支配树和后支配树, 如果模糊测试使用分支覆盖, 则计算分支的前支配树和后支配树; 合并前、后支配树中相同的节点, 构成图的形式, 规约图中的强联通分量并去除冗余边后, 得到每个函数对应的超级块支配图; 最后根据函数调用关系合并每个函数的超级块支配图, 得到全局超级块支配图。根据超级块支配图的性质得到需要插桩的基本块/分支和对应的未插桩的基本块/分支, 记录在文件中。该文件将在模糊测试开始时由模糊测试进程加载到共享内存。

如前所述, 基本块标记的结果将用于指导模糊测试过程中的动态二进制插桩位置的选择, 但插桩平台 Pin 与 IDA Pro 生成的控制流图对基本块的划分存在不一致的情况。图 12 所示为 IDA Pro 生成的用于计算全局超级支配图的控制流图片段。根据支配关系, SiCsFuzzer 会将标记图 12 中的基本块 2 和基本块 3 为需要动态插桩的基本块。但在覆盖率跟踪过程中, 如果执行的路径为基本块 1 的假分支, Pin 会将基本块 3 划入基本块 2, 进而造成基本块 3 的覆盖被重复计算。因此本文从 IDA Pro 提取目标程序的程序控制流图时, 对以 mov 指令结束的基本块进行标记, 在分析覆盖率时, 遇到带有标记的基本块时, 则将其对应覆盖率减一, 以调整覆盖率计算。

### 4.2 “预热”优化实现

对于本文工作而言, 理想的方式是使用静态重写, 其开销较低, 但是 Windows 平台上的二进制重写工具如 Dyninst、McSema[19]等工具鲁棒性较差。采用二进制动态跟踪代码覆盖率虽然开销较高, 但确是目前最具普适性的方法。因此本文实现了“预热”优化来降低二进制动态跟踪代码覆盖率的开销。

为实现“预热”优化, 我们设计并实现了 fuzzer 进程和 instrumentor 进程之间的通信。如图 13, 一共

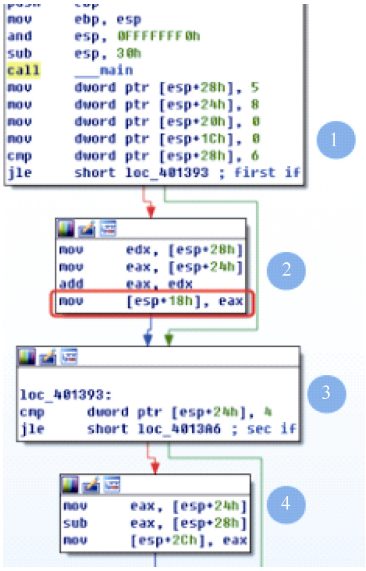



**图 12　部分程序控制流图**

**Figure 12　Part of the program control flow diagram**

有以下四个主要对象:

1) fuzzer: fuzzer 是基于覆盖率反馈的模糊测试的主进程, 主要负责初始化种子队列、生成测试用例、分析代码覆盖率以及启动 instrumentor 进程。

2) instrumentor: instrumentor 是由 fuzzer 开启的子进程, 该进程是插桩进程, 主要负责控制目标程序的执行, 并分析在目标程序的哪些位置插入跟踪代码。

3) 目标程序: 指的是目标二进制程序, 由 instrumentor 控制执行。

4) 共享内存: SiCsFuzzer 使用共享内存存储代码覆盖信息。在模糊测试运行过程中, instrumentor 首先初始化该共享内存, 目标程序在插桩工具的控制下执行并将覆盖率信息记录到共享内存中, 当目标程序运行到结束位置后, fuzzer 进程开始读取共享内存的数据并分析代码覆盖信息。

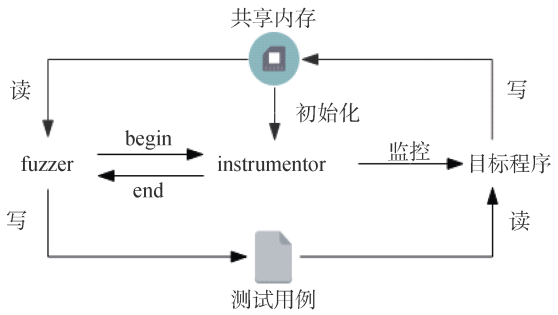



**图 13　进程间通信设计**

**Figure 13　The design of process communication for warm-up optimization**

fuzzer 进程与 instrumentor 进程的交互如下:

1) fuzzer 选择一个测试用例并开始新一轮的模糊测试。首先 fuzzer 向 instrumentor 进程发送一个 *begin* 信号。表示 instrumentor 可以开始运行。

2) instrumentor 收到 *begin* 信号后, 首先对共享内存做相关初始化工作, 然后控制目标二进制程序开始运行。同时保存目标程序当前的上下文信息。

3) 目标二进制程序首先读取测试用例, 并运行, 并在 instrumentor 的控制下, 将覆盖率信息记录到共享内存中。

4) instrumentor 监控目标二进制程序运行到指定结束指令位置后, instrumentor 发送 *end* 信号给 fuzzer 进程, 表示目标程序运行结束, 同时 instrumentor 恢复 2)中保存的上下文, 并返回到程序开始指令位置。

5) fuzzer 接收到 *end* 信号后, 开始读取共享内存的数据并分析程序运行时的覆盖率等信息。

6) 返回到 1)并重复。

本文实现的“预热”优化基于插桩工具 Pin。为了实现在指定的开始指令位置获取目标程序的上下文信息, 并在结束指令位置恢复保存的上下文信息, 传统的方法可使用 Pin 进行指令插桩, 记录写内存的

**表 1　二进制目标程序信息**

**Table 1　Binary target program information**

| 程序 | magick | i_view | nconvert | flashplayer | Rar | 7z | gpg | pdf2htmlEX | Pdftk |
|---|---|---|---|---|---|---|---|---|---|
| 程序包 | ImageMagick | IrfanView | Xnview | \ | WinRAR | 7Zip | GnuPG | \ | \ |
| 版本 | 7.0.8 | 4.53.0 | 7.25 | 23.0.0 | 5.11.1 | 19.0.0 | 2.2.17 | 0.14.6 | 2.02 |
| 类别 | 图片处理 | 图片处理 | 图片处理 | 视频处理 | 文档压缩 | 文档压缩 | 加密 | 文档处理 | 文档处理 |
| 大小 | 15.2MB | 1.77MB | 6.8MB | 14.1MB | 400KB | 286KB | 1.08MB | 19.3MB | 8.47MB |
| 基本块数 | 387531 | 53227 | 71622 | 448961 | 22358 | 16356 | 39422 | 478850 | 150433 |
| 分支数 | 545085 | 77011 | 102984 | 589655 | 31795 | 19906 | 59220 | 689909 | 184583 |

指令操作和内存的状态, 当程序运行到结束指令的位置时, 恢复内存状态。但是该方法开销很大, 而本文采用的方法是在程序运行到开始位置时, 使用 PyDbg 获取目标程序的上下文信息, 包括寄存器

状态, 内存状态, 在程序运行到结束指令的位置时, 恢复保存的寄存器、内存状态。

AFL 实现了 fork-server 机制[20]来避免“预热”阶段的开销, 但是 fork-server 的实现思想主要依赖 Linux 的系统函数 *fork*, 在每一轮模糊测试时, 使用 *fork* 函数复制一个新的目标程序进程并执行, 避免了载入目标程序的重复性工作。*fork* 函数使用写时复制(*copy on write*)的机制, 开销非常低, 但是 Windows 平台没有类似的函数, 因此在 Windows 平台实现 fork-server 机制具有很大的挑战。

## 5 实验与分析

本文在现有模糊测试平台 Puzzer[18]上实现了基于稀疏插桩的闭源软件模糊测试原型 SiCsFuzzer。本章将 SiCsFuzzer 与 Puzzer 和 WinAFL 进行对比, 本文的实验主要回答了以下 5 个问题:

Q1: 采用稀疏插桩的覆盖率跟踪技术使需要插桩的基本块或分支的数量降低了多少?

Q2: 基于稀疏插桩的覆盖率跟踪技术对模糊测试的效率改进情况如何?

Q3: “预热”优化对模糊测试效率的改进情况如何?

Q4: 在执行时间相同的前提下, SiCsFuzzer 是否能够比 Puzzer 达到更高的覆盖率?

Q5: SiCsFuzzer 与 WinAFL 的效能对比结果如何?

### 5.1 实验配置

本章以表 1 所示的 9 个 Windows 平台闭源二进制软件作为实验分析的目标程序。程序涉及图片处理、视频处理、文件压缩、加解密和文档处理 5 类, 软件规模从 266KB~19.3MB 不等。实验均运行在 Windows 7-32、2 核 Intel i7-87700 处理器的虚拟机上。

为了避免随机因素的影响, 确保对比实验的公平性, 我们先使用 Puzzer 工具分别对每一个目标程序测试了 24h, 将变异生成的测试用例保存下来, 作为对比实验的测试用例集。再对工具 Puzzer 和 SiCsFuzzer 在不执行变异的前提下, 分别执行上述测试用例集, 并记录每个测试用例跟踪覆盖率的时间开销。在统计和对比过程中, 使用“切尾均值降噪”的方法[21], 删除了执行时间中最高和最低的 30%, 以减少其他因素对程序执行时间的影响, 更好的展示中位数的趋势。最后, 以目标程序正常执行所需的时间为“基准”, 将所有执行时间换算为相对于基准时间的“相对执行时间”。

### 5.2 实验结果分析

**Q1**: 图 14 所示为 SiCsFuzzer 对表 1 所示的 9 个程序采用稀疏插桩后, 每个目标程序中需要插桩的分支数量占传统方法(对所有基本块/分支插桩)插桩数量的比例。与传统模糊测试默认对所有基本块插桩的方法相比, 目标程序需要插桩的分支数量平均减少了 59.39%, 最高了减少了 67%。

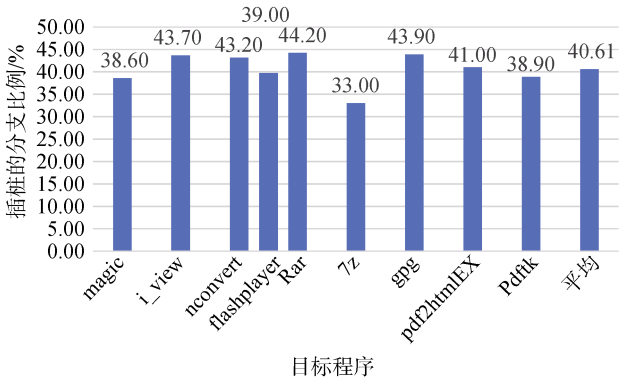



**图 14 插桩的分支数量的比例**

**Figure 14 The percentage of marked edges of SiCsFuzzer**

**Q2**: 接下来, 本文分析了采用基于稀疏插桩跟踪技术后 SiCsFuzzer 的性能。本文记录了对于每一个目标程序, 采用基于稀疏插桩的覆盖率跟踪技术的 SiCsFuzzer 和改进前的模糊测试工具 Puzzer 的相对执行时间。如图 15 所示, 采用基于稀疏插桩跟踪技术后, SiCsFuzzer 的性能相比 Puzzer 提升了 16.8%, 图中的折线代表将目标程序运行在二进制插桩平台 Pin 上, 但不对目标程序进行插桩的时间开销, 相对于程序正常执行时间开销进行标准化后的比值, 简称“Pin 空载”的相对执行时间。该折线也是采用稀疏插桩跟踪技术后, SiCsFuzzer 能达到的性能上限。实验结果表明, 对于所有被测目标程序, Pin 空载的平均相对执行时间为 6.1, Puzzer 的平均相对执行时间为 8.3, 而 SiCsFuzzer 的平均相对执行时间为 6.9。换而言之, SiCsFuzzer 相对于对所有基本块进行插桩的 Puzzer, 消除了将近 64%的插桩开销(计算方法: (8.3–6.9)/(8.3–6.1))。

对于 *flashplayer* 这类需要交互的目标程序, 程序解析完文件后, 无法自动结束, 因此对于程序正常运行时间、Pin 空载时间和模糊测试时间的获取方法如下: 本文将其开始显示视频内容的时间其作为结束时间, 然后根据统计将它的正常执行时间设为 2 s, Pin 空载时间设为 3 s。在执行模糊测试之前, 通过逆向确定其解析文件结束的位置, 在该位置插入结束运行的代码, 以此方法获取交互类软件的执行

模糊测试的时间。

**Q3**: 图 16 所示为使用“预热”优化后的 SiCsFuzzer 与 Puzzer 的性能差异。实验结果表明, 加入“预热”优化后, SiCsFuzzer 的平均相对执行时间是 2.1, 也就是说 SiCsFuzzer 的平均执行时间是程序正常运行时间的 2.1 倍。而改进前工具 Puzzer 的平均相对执行时间是 8.3。换而言之, SiCsFuzzer 的模糊测试效率平均比 Puzzer 快 3 倍(1.24~8.25 倍)。

对于选定的 9 个目标软件, *gpg* 和 *flashplayer* 的模糊测试效率甚至分别优于这两个程序的正常执行效率。这是由于引入“预热”优化后, 和目标程序正常运行相比, SiCsFuzzer 避免了加载和启动目标程序的开销, 并且只执行目标程序中选定的部分, 即从读取输入到程序退出之间的部分。尤其是对于 *flashplayer* 这类图形界面软件, 加载和启动目标程序的开销本身在程序正常执行时间中占比较高, 更能发挥“预热”优化的优势。

**Q4**: 图 17 所示为相同时间(24 h)内, SiCsFuzzer 与 Puzzer 的覆盖率增长情况。根据实验结果统计, SiCsFuzzer 平均只用了 6.3 h 就达到了 Puzzer 执行 24 h 才能达到的覆盖率。

对于部分目标软件(*magick,Rar,pdf2html,iview*), SiCsFuzzer 在 24 h 内达到的覆盖率明显高于 Puzzer。对于 *7zip* 和 *gpg* 两个目标软件, 在模糊测试执行一段时间后, 覆盖率不再变化。由此可见, 在达到相同覆盖率时, SiCsFuzzer 所需时间明显少于 Puzzer, 不过为了更好的提升模糊测试的效率, 仍需要结合通过生成有价值的种子来提升覆盖率的方法。

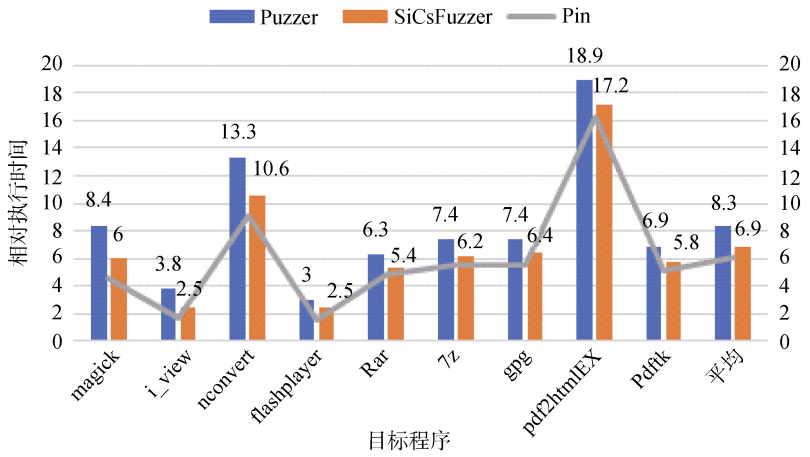



**图 15　基于稀疏插桩跟踪技术的性能对比**

**Figure 15　Per-program relative overheads of SiCsFuzzer versus Puzzer**

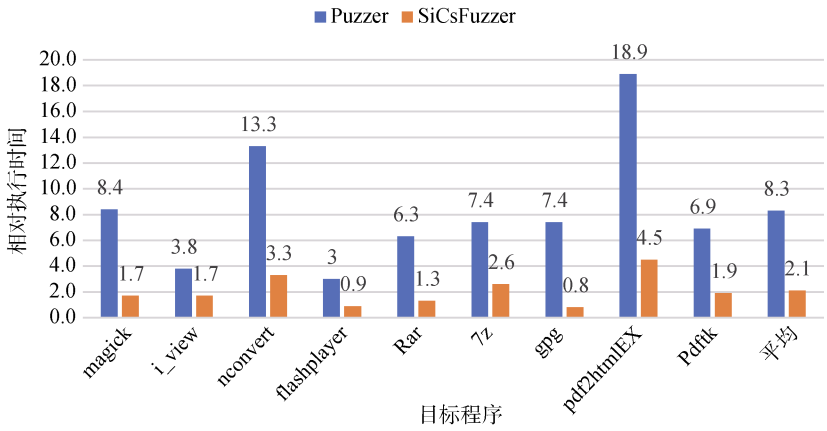



**图 16　结合“预热”优化的 SiCsFuzzer 与 Puzzer 的性能对比**

**Figure 16　Per-program relative overheads of SiCsFuzzer with warm-up optimization versus Puzzer**

**Q5:** WinAFL 是 AFL 的扩展版本, 其使用二进制插桩工具 DynamoRIO[22]跟踪覆盖率, 以支持 Windows 平台上目标软件的模糊测试使用。此外, WinAFL 默认开启了与 SiCsFuzzer 类似的“预热”优化机制, 但只保存并恢复了指定程序点的寄存器状态, 故在实验过程, 对本章选定的 9 个目标程序, 只

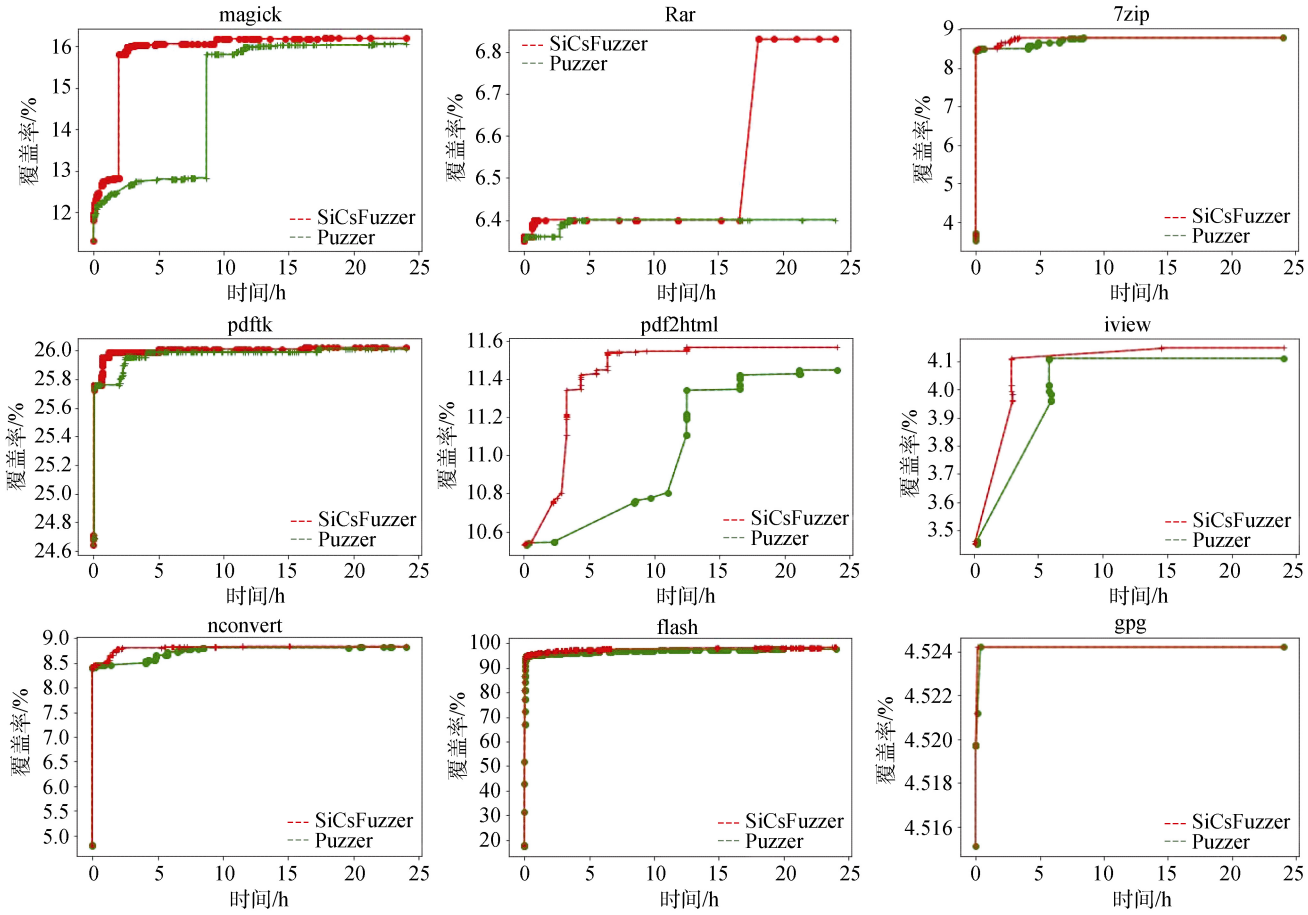



**图 17　覆盖率**

**Figure 17　Code coverage rate**

能成功测试其中的 4 个目标程序。图 18 对比了 SiCsFuzzer 和 WinAFL 相对执行时间。实验结果表明, 对于 WinAFL 能成功测试的 4 个目标软件, 其模糊测试效率平均比 SiCsFuzzer 快 18%。但上述性能差异主要来自二进制插桩平台 Pin 和 DynamoRIO 之间的性能差异[23], 且 Pin 更加稳定, 能够测试更多的目标程序[13]。

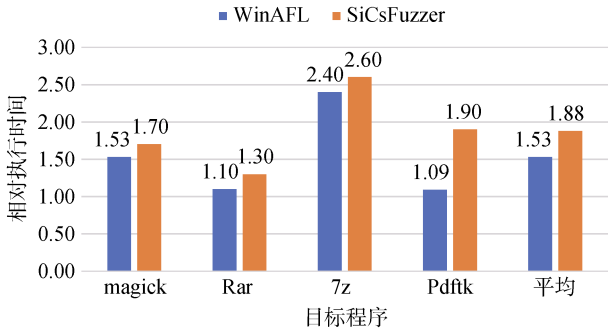



**图 18　SiCsFuzzer 与 WinAFL 性能对比**

**Figure 18　Per-program relative overheads of SiCsFuzzer versus WinAFL**

综上, SiCsFuzzer 在 Windows 平台 9 个规模在 286KB~19.3MB, 类型涉及图片处理、视频处理、文件压缩、加密和文档处理的应用所组成的测试集上, 平均只需对 40.61%的分支进行插桩, 模糊测试引入的额外时间开销为程序正常运行时间的 1.1 倍, 执行效率比传统的基于覆盖率反馈的模糊测试方法快 3 倍。

此外, 借助 SiCsFuzzer, 新发现了 PDFtk 最新版本(v2.02)和 XnView 最新版本系列软件 Nconvert (v7.32)中的未知漏洞各1个; 前者是在文件解析过程中由于栈溢出导致的程序崩溃; 后者是由于内存破坏导致软件崩溃, 已获厂商确认。上述未知漏洞的发现也得益于模糊测试效率的提升。以 XnView 为例, SiCsFuzzer 运行6.5 h 后触发该漏洞, 漏洞触发时的覆盖率为9.97%; 而 Puzzer 在运行到相同时间时的覆盖率为8.2%, 再运行18 h 后, 才触发了漏洞, 触发漏洞时的覆盖率为9.99%。对于 PDFtk, SiCsFuzzer 运行 4 h 后触发该漏洞, 漏洞触发时覆盖率为26.15%; 而 Puzzer 在运行到相同时间时的覆盖率为26.03%, 再运行9 h 后, 才触发了漏洞, 触发漏洞时的覆盖率为 26.18%。

## 6 相关工作

当前针对提升模糊测试性能提升的相关工作可以分为两大类[15]: (1)通过生成有价值的种子降低模糊测试整体的开销。因为有价值的种子更有可能触发软件漏洞, 也就是通过生成有价值的种子来缩短发现软件漏洞的时间, 从而降低模糊测试整体的开销; (2)通过降低执行单个测试用例的时间来降低模糊测试的开销。也就是说, 在给定时间内, 模糊测试能够测试更多个测试用例, 达到更高的代码覆盖率。目前现有的工作中, 大多属于第一种。而本文通过降低执行单个测试用例时跟踪测试用例代码覆盖率的开销, 属于第二种方法。两个方向的相关工作如下:

**生成有价值的种子**。生成有价值的种子主要与以下因素有关: (1)种子选择策略; (2)种子变异策略。一些研究[3-4]表明种子选择策略对模糊测试非常重要, 像 libFuzzer[24]对提升代码覆盖率的种子给予更大的优先权, 因为这样的种子更有可能探索到目标程序未被执行到的代码区域, VUzzer[12]对执行路径更深的种子给予更大的优先权, 因为 VUzzer 旨在发现更深的路径。总之, 良好的种子选择策略能够加速模糊测试工具探索到目标程序不同的代码区域和发现软件漏洞。

AFL 和 libFuzzer 等模糊测试工具采用一组确定性和随机的变异策略对种子进行变异, 生成大量的测试用例, 这种方法虽然简单, 但是在模糊测试进行一段时间后, 很难提升代码覆盖率。一些更加先进的模糊测试工具试图使用一种更聪明的方法对种子进行变异。Driller[25]将模糊测试与符号执行结合在一起, 当覆盖率不再提升时, 通过符号执行计算有效输入。但是, 由于符号执行不适用于实际程序, 因此应用符号执行来辅助测试大型软件是不切实际的。VUzzer 使用控制流和数据流特征来识别要变异的字节。 Skyfire[26]利用大量现有样本中的知识来生成分布良好的种子输入。

尽管这些模糊工具可以有效地生成更多有价值的种子, 但它们仍然面临着执行所有测试用例的开销。因此, 降低执行单个测试用例的开销, 也会带来极大的好处。

**降低执行单个测试用例的开销**。文献[9]就降低执行单个测试用例的开销提出了改进方案 UnTracer, 其在跟踪代码覆盖率之前, 先通过一轮轻量级的检测, 判断变异生成的测试用例是否是有价值的测试用例, 即是否能够触发新的基本块或分支。对于没有价值的测试用例, 直接丢弃; 而针对有价值的测试用例, UnTracer 仍然面对继续跟踪其覆盖率的开销; 近期的相关工作[10]提出了二进制重写技术 RetroWrite, 其速度是 QEMU 的 4.5 倍, 但该工作针对的目标程序为 64 位的地址无关代码的 Linux 闭源程序。并且, 文献[9]的统计显示, QEMU 引入的额外开销约为目标程序正常执行时间的 10 倍, 因此得出, RetroWrite 的执行速度约为程序正常运行的 2.4 倍, 而本文的 SiCsFuzzer 的执行速度为程序正常运行的 2.1 倍。

文献[15]提出为模糊测试工具设计三个操作原语, 降低了在多核计算机进行模糊测试的性能开销。还有一些其他工作, 提出使用硬件辅助的方法, 如 kAFL[7]、PTfuzz[8]和 honggFuzz[2]使用 Intel Processor Tracking(IPT)[27]进行跟踪测试用例的代码覆盖。但是这些工作对硬件有一定的要求, 不能较好地应用于大部分主流的应用平台。另外, 文献[28]提出了一种插桩方法, 使用程序控制流图的前支配树信息来减少插桩位置的数量, 但是由于只使用了前支配树信息, 缺乏后支配树信息, 所以这种插桩策略仍存在冗余。文献[29]提出结合前支配树和后支配树信息来减少插桩位置的数量, 但是该工作是针对具有源代码的目标程序。针对具有源代码的目标程序, 这些工作根据程序的控制流图信息, 通过前支配树信息或结合后支配树信息得到每个函数需要插桩的基本块, 然后通过静态插桩的方法即可对目标程序进行测试。而本文是针对闭源程序(尤其是 Windows 平台), 闭源程序的插桩多使用动态插桩。因此本文详细地剖析了动态跟踪覆盖率的开销, 基于分析结果, 本文提出稀疏插桩的覆盖率跟踪技术和“预热”优化。本文提出的基于稀疏插桩的覆盖率跟踪技术对全局控制流进行分析, 处理了程序中含有 exit、abort 等函数的情况, 根据动态插桩的特点, 设计并实现了覆盖率重建算法。结合 “预热”优化, 更有效的降低了模糊测试的开销。

## 7 总结

本文提出了一种面向闭源程序的高效模糊测试方法。该方法借助基于稀疏插桩的覆盖率跟踪以及针对流行的动态二进制插桩工具(例如 Pin)的“预热”优化机制, 降低基于覆盖率反馈的模糊测试过程中跟踪覆盖率的开销, 提高模糊测试效率。基于该解决方案实现的原型工具 SiCsFuzzer, 在 Windows 平台 9 个规模在 286KB~19.3MB、类型涉及图片处理、视频处理、文件压缩、加密和文档处理等类型应用所组成的测试集上, 引入的额外时间开销为程序正常

运行时间的 1.1 倍, 执行效率比传统的基于覆盖率反馈的模糊测试工具快 3 倍。此外, 使用 SiCsFuzzer, 还发现了 PDFtk 和 XnView 软件最新版本中的未知漏洞各 1 个。

本文的下一步工作包括以下两个方面。首先, 本文提出的基于稀疏插桩的覆盖率跟踪技术需要借助静态分析技术得到目标程序中各基本块间的支配关系。静态分析的精度, 尤其是对间接跳转目标的识别精度, 将直接影响超级支配图的精度, 甚至造成有价值的种子被丢弃。在后续工作中, 将尝试通过模糊测试过程中记录的动态执行轨迹, 反过来指导并补充静态分析未识别出的间接跳转目标, 完善控制流图。此外, 目前有很多工作[30-32]致力于提升静态分析技术的准确性, 随着该技术的完善, 本文提出的算法也将得到更好的效果。其次, 如 4.1 节所述, 本文方法在实现过程中遇到动态插桩工具 Pin 与静态分析工具 IDA Pro 生成的控制流图对基本块的划分不一致的情况, 虽然就目前分析的实例而言, 识别以 mov 指令结尾的基本块能在一定程度缓解该问题, 但仍然不够精准。在后续的工作中, 将通过对更多实例的分析总结, 归纳并识别出所有导致基本块划分不一致的情况, 提高覆盖率计算的精度。

## 参考文献


[1] “american fuzzy lop,” http://lcamtuf.coredump.cx/afl/. 2019.

[2] “honggfuzz.” https://github.com/google/honggfuzz. 2019.

[3] Böhme M, Pham V T, Nguyen M D, et al. Directed Greybox Fuzzing[C]. *The 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017: 2329-2344.

[4] Böhme M, Pham V T, Roychoudhury A. Coverage-Based Greybox Fuzzing as Markov Chain[J]. *IEEE Transactions on Software Engineering*, 2019, 45(5): 489-506.

[5] Nethercote, Nicholas. Dynamic binary analysis and instrumentation. No. UCAM-CL-TR-606. University of Cambridge, Computer Laboratory, 2004.

[6] “AFL-Dyninst.” https://github.com/talos-vulndev/afl-dyninst.2019.

[7] Schumilo, Sergej, et al. kAFL: Hardware-Assisted Feedback Fuzzing for OS Kernels. 26th USENIX Security Symposium (USENIX Security 17). 2017:167-182.

[8] Zhang G, Zhou X, Luo Y Q, et al. PTfuzz: Guided Fuzzing with Processor Trace Feedback[J]. *IEEE Access*, 2018, 6: 37302-37313.

[9] Nagy S, Hicks M. Full-Speed Fuzzing: Reducing Fuzzing Overhead through Coverage-Guided Tracing[C]. *2019 IEEE Symposium on Security and Privacy*, 2019: 787-802.

[10] Dinesh S, Burow N, Xu D Y, et al. RetroWrite: Statically Instrumenting COTS Binaries for Fuzzing and Sanitization[C]. *2020 IEEE Symposium on Security and Privacy*, 2020: 1497-1511.

[11] Agrawal H. Dominators, Super Blocks, and Program Coverage[C]. *The 21st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 1994: 25-34.

[12] Rawat S, Jain V, Kumar A, et al. VUzzer: Application-Aware Evolutionary Fuzzing[C]. *Proceedings 2017 Network and Distributed System Security Symposium*, 2017: 1-14.

[13] Luk, Chi-Keung, et al. Pin: building customized program analysis tools with dynamic instrumentation. Acm sigplan notices, 2005, 40(6), 190-200.

[14] Gan S T, Zhang C, Qin X J, et al. CollAFL: Path Sensitive Fuzzing[C]. *2018 IEEE Symposium on Security and Privacy*, 2018: 679-696.

[15] Xu W, Kashyap S, Min C, et al. Designing New Operating Primitives to Improve Fuzzing Performance[C]. *The 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017: 2313-2328.

[16] Hsu C C, Wu C Y, Hsiao H C, et al. INSTRIM: Lightweight Instrumentation for Coverage-Guided Fuzzing[C]. *Proceedings 2018 Workshop on Binary Analysis Research*, 2018: 74-78.

[17] Agrawal H. Efficient Coverage Testing Using Global Dominator Graphs[J]. *ACM SIGSOFT Software Engineering Notes*, 1999, 24(5): 11-20.

[18] Yang M F, Huo W, Zou Y Y, et al. Programmable Fuzzing Technology[J]. *Journal of Software*, 2018, 29(5): 1258-1274.
(杨梅芳, 霍玮, 邹燕燕, 等. 可编程模糊测试技术[J]. *软件学报*, 2018, 29(5): 1258-1274.)

[19] “McSema,” https://github.com/lifting-bits/mcsema/blob/master/. 2019.

[20] “Fuzzing random programs without execve(),” https://lcamtuf.blogspot.com/2014/10/fuzzing-binaries-without-execve.html. 2019.

[21] Arnautov, Sergei, et al. SCONE: Secure Linux Containers with Intel SGX. 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16). 2016: 689-703.

[22] “DynamoRIO,” http://dynamorio.org. 2019.

[23] Bernat A R, Miller B P. Anywhere, Any-Time Binary Instrumentation[C]. *The 10th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools*, 2011: 9-16.

[24] Serebryany K. Continuous Fuzzing with libFuzzer and AddressSanitizer[J]. *2016 IEEE Cybersecurity Development* (*SecDev*), 2016: 157.

[25] Stephens N, Grosen J, Salls C, et al. Driller: Augmenting Fuzzing through Selective Symbolic Execution[C]. *Proceedings 2016 Network and Distributed System Security Symposium*, 2016: 21-24.

[26] Wang J J, Chen B H, Wei L, et al. Skyfire: Data-Driven Seed Generation for Fuzzing[C]. *2017 IEEE Symposium on Security and Privacy*, 2017: 579-594.

[27] “Intel Processor Trace Tools.” https://software.intel.com/en-us/node/721535. 2019.

[28] Tikir M M, Hollingsworth J K. Efficient Instrumentation for Code Coverage Testing[J]. *ACM SIGSOFT Software Engineering Notes*, 2002, 27(4): 86-96.

[29] XU Xiao-feng, et al. Design and implementation of software testing tool based on super block dominator graph. Application Research of Computers. 2010, 27(3): 923-927.

[30] Meng X Z, Miller B P. Binary Code is not Easy[C]. *The 25th In-*

*ternational Symposium on Software Testing and Analysis*, 2016: 24-35.

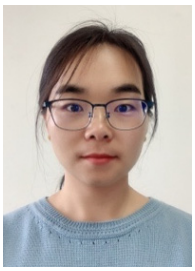
[31] Bao, Tiffany, et al. BYTEWEIGHT: Learning to Recognize Functions in Binary Code. 23rd USENIX Security Sym-posium (USENIX Security 14), 2014: 845-860.

[32] Harris L C, Miller B P. Practical Analysis of Stripped Binary Code[J]. *ACM SIGARCH Computer Architecture News*, 2005, 33(5): 63-68.

**刘丽艳** 于 2017 年在电子科技大学计算机科学与技术专业获得学士学位。现在中国科学院大学计算机软件与理论专业攻读硕士学位。研究领域为模糊测试。Email: liuliyan@iie.ac.cn

**李丰** 于 2013 年在中国科学院大学获博士学位, 现为中国科学院信息工程研究所副研究员, 研究方向为程序分析与软件漏洞挖掘。Email: lifeng@iie.ac.cn

**邹燕燕** 于 2014 年在中国科学技术大学计算机系统结构专业获得硕士学位。现在中国科学院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分析。研究兴趣包括: 漏洞挖掘、模糊测试、程序分析。Email: zouyanyan@iie.ac.cn



**周建华** 于 2013 年在北京科技大学通信工程专业获得硕士学位, 现任中国科学院信息工程研究所工程师, 研究领域为软件安全分析, 研究兴趣包括: 代码审计、漏洞挖掘。Email: zhoujianhua@iie.ac.cn

**朴爱花** 于 2005 年在北京科技大学系统工程专业获得了硕士学位。现任中国科学院信息工程研究所高级工程师。研究领域为软件漏洞分析与风险评估。Email: piaoaihua@iie.ac.cn

**刘峰** 于 2009 年在中国科学院大学获博士学位。现为中国科学院信息工程研究所研究员, 研究领域为信息对抗理论与技术, 信息安全战略研究。 Email: liufeng@iie.ac.cn

**霍玮** 于 2010 年在中国科学院计算技术研究所获博士学位。现任中国科学院信息工程研究所博士生导师, 中国科学院青年创新促进会成员。主要研究领域包括软件漏洞挖掘、利用和安全评测、基于大数据及知识图谱的软件安全分析、信息系统安全分析等。Email: huowei@iie.ac.cn