计算机系统应用 ISSN 1003-3254, CODEN CSAOBN
[doi: 10.15888/j.cnki.csa.009703] [CSTR: 32024.14.csa.009703]

E-mail: csa@iscas.ac.cn
http://www.c-s-a.org.cn
Tel: +86-10-62661041

基于程序分析的二进制软件模糊测试①

王文婷[1,3], 孙嘉珺[2], 万逸峰[2], 王文杰[2], 田东海[2]

[1](浙江大学 控制科学与工程学院, 杭州 310027)
[2](北京理工大学 软件安全工程技术北京市重点实验室, 北京 100081)
[3](国网山东省电力公司电力科学研究院, 济南 250003)
通信作者: 田东海, E-mail: dhai@bit.edu.cn

摘　要: 针对现有二进制模糊测试难以深入程序内部发现漏洞这一问题, 提出一种融合硬件程序追踪、静态分析和混合执行 3 种技术的多角度优化方案. 首先, 利用静态分析和硬件追踪评估程序路径复杂度及执行概率; 之后, 根据路径复杂度与执行概率进行种子选择和变异能量分配; 同时, 利用混合执行辅助种子生成并记录关键字节用于针对性变异. 实验结果表明, 相比现有模糊测试方案, 该方案在多数情况下能发现更多的程序路径和 crash.

关键词: 模糊测试; 硬件程序追踪; 静态分析; 混合执行; 二进制软件

引用格式: 王文婷,孙嘉珺,万逸峰,王文杰,田东海.基于程序分析的二进制软件模糊测试.计算机系统应用. http://www.c-s-a.org.cn/1003-3254/9703.html

Fuzzing for Binary Programs Based on Program Analysis

WANG Wen-Ting[1,3], SUN Jia-Jun[2], WAN Yi-Feng[2], WANG Wen-Jie[2], TIAN Dong-Hai[2]

[1](College of Control Science and Engineering, Zhejiang University, Hangzhou 310027, China)
[2](Beijing Key Laboratory of Software Security Engineering Technique, Beijing Institute of Technology, Beijing 100081, China)
[3](State Grid Shandong Electric Power Institute, Jinan 250003, China)

Abstract: Existing methods for binary fuzzing are difficult to dive into programs to find vulnerabilities. To address this problem, this study proposes a multi-angle optimization method integrating hardware-assisted program tracing, static analysis, and concolic execution. Firstly, static analysis and hardware-assisted tracing are used to calculate program path complexity and execution probability. Then, seed selection and mutation energy allocation are performed according to the path complexity and execution probability. Meanwhile, concolic execution is leveraged to assist seed generation and record key bytes for targeted variations. Experimental results show that this method finds more program paths as well as crashes in most cases, compared to other fuzzing methods.

Key words: fuzzing; hardware processor tracing; static analysis; concolic execution; binary program

1　引言

模糊测试 (fuzzing) 具有自动化程度高、资源消耗小和适用范围广等特点, 被广泛应用于计算机软件的安全测试中[1,2]. 例如, 当程序中存在缓冲区溢出漏洞时, 模糊测试器会尝试生成一些较长的畸形输入用例, 在测试过程中触发被测程序执行崩溃. 研究人员通过分析触发崩溃的用例文件和崩溃上下文, 定位漏洞存在位置. 较为典型的溢出类漏洞如“心脏滴血” (CVE-2014-0106) 漏洞, 曾大范围影响了使用 OpenSSL 库的设备. 然而, 传统模糊测试方法存在随机性和盲目性强的特点, 导致测试过程的针对性较差, 表现为测试深度和广度不够[3], 尤其对于测试二进制软件, 传统模糊测试的效率较低. 以目前主流的模糊测试工具 AFL[4]为例, AFL 首先将所有初始种子文件加入到种子队列. 随

① 基金项目: 国家电网有限公司科技项目 (5700-202316312A-1-1-ZN)
收稿时间: 2024-05-19; 修改时间: 2024-06-12; 采用时间: 2024-07-04; csa 在线出版时间: 2024-11-15

后, AFL 从种子队列中选择一个种子文件进行变异, 从而生成若干新测试用例. 通过执行被测程序, AFL 可以统计被测程序的分支覆盖信息, 从而评估测试用例是否触发新的程序路径. AFL 保存触发新路径的测试用例作为新的种子文件, 在后续测试中基于该种子文件生成其他测试用例. AFL 反复执行种子选择、种子变异以及执行被测程序 3 个过程, 直到测试完成.

在测试二进制软件时, AFL 使用 QEMU 模式[5]统计分支覆盖信息. 然而 QEMU 对模糊测试速度有较大影响, 其时间开销是插桩方式的 2–5 倍[5]. 为了提高测试速度, PTrix[6]使用性能开销较低的 Intel PT[7]和改进的 PT 解码器获取二进制软件运行的控制流信息, 然后基于这些信息更新 AFL 的 bitmap, 在一定程度上提高了二进制模糊测试的效率. 然而, PTrix 直接利用 PT 包更新 bitmap 无法获得完整程序路径, 不利于进行详细分析.

此外, AFL 种子选择策略的盲目性问题可能导致触发程序 crash 或新路径的重要种子文件需要花费较长时间才被选中. AFLFast[8]发现 AFL 的多数测试用例都执行覆盖了被测程序中相同的高频路径, 导致花费大量时间而没有发现新路径. 如果优先执行低频路径对应的种子文件, 可以更快提高 AFL 的路径覆盖率. 此外, 模糊测试的目标是遍历程序状态空间以发现程序漏洞, 因此种子选择需要关注程序路径存在漏洞的可能性并优先选择漏洞可能性大的程序路径对应的种子文件. 另一方面, AFL 的种子变异策略容易产生大量低效测试用例, 消耗测试时间和资源. 例如, AFL 没有考虑种子文件对应程序路径的执行频率, 可能造成部分程序路径多次重复执行, 而其他路径的执行次数较少. AFLFast 为执行低频路径的种子文件分配更多的变异能量, 提高了 AFL 的路径覆盖率. 此外, AFL 的变异策略难以生成满足复杂约束条件的测试用例. 尤其对于含有大量校验字段和 Magic Bytes 的格式化文件, AFL 很难通过变异生成满足格式化文件中条件检查的测试用例.

为了缓解上述问题, 本文提出了一种硬件追踪与程序分析相结合的技术辅助模糊测试. 本文利用硬件程序追踪机制快速获取程序执行路径, 辅助模糊测试统计被测程序的分支覆盖情况并为后续优化提供程序路径信息. 基于该信息, 模糊测试器动态计算路径执行概率, 同时利用静态分析得到的基本块复杂度计算程序路径的复杂度. 之后, 模糊测试综合利用路径复杂度和执行概率进行种子选择和变异能量分配. 对于选中的种子文件, 本文利用混合执行提取到的关键字节信息进行关键字节变异, 同时为模糊测试提供满足复杂路径约束的测试用例. 通过以上方法, 本文可以提高二进制软件模糊测试的效率和漏洞发现能力. 本文的主要工作及创新点如下.

(1) 提出了一种基于程序分析与硬件追踪配合的种子选择和变异能量分配方法. 通过程序分析技术结合硬件追踪收集到的程序路径信息动态评估程序路径复杂度及执行概率两个指标, 提升了模糊测试的针对性.

(2) 提出了一种基于混合执行的种子文件生成和关键字节变异方法. 利用混合执行针对性生成覆盖复杂路径的种子文件, 在此过程中记录种子文件中的关键字节信息供模糊测试器进行针对性变异, 提高了种子覆盖率及变异有效性.

(3) 实现了集成上述策略的原型系统 BPAFuzz. 在真实软件场景下与 AFLFast 等测试器进行的对比实验中, BPAFuzz 可以在多数情况下发现更多程序路径或 crash, 表明本文方案对闭源二进制软件具有良好的程序路径探索和漏洞发现能力.

2 方法的设计与实现

2.1 方法概述

本文融合硬件追踪技术、静态分析技术以及混合执行技术实现二进制软件模糊测试. 通过硬件追踪技术获取分支覆盖信息, 支撑后续模糊测试过程上的优化; 基于对程序中复杂路径较容易产生漏洞的观察, 本文采用静态分析技术计算程序路径复杂度, 将其作为指标参与种子选择与能量分配; 针对测试中可能出现的复杂约束难以求解的问题, 本文结合混合执行技术求解分支约束, 同时收集关键字节信息, 对用例进行针对性变异, 生成更高质量的种子, 以提升程序覆盖率或触发程序中的漏洞.

图 1 展示了本文提出的二进制软件模糊测试的整体架构. 该架构包含 3 个组件, 分别为静态分析组件、混合执行组件以及基于硬件追踪的模糊测试组件. 其中, 静态分析组件计算程序路径的复杂度信息并提供给模糊测试器; 混合执行组件采用并行的方式与模糊测试器同时运行, 避免为模糊测试器带来额外开销, 同

时针对测试过程中产生的文件进行约束求解, 并记录关键字节信息, 最终生成关键字节信息文件以及提升覆盖率的种子文件; 核心组件模糊测试器优化原本的基于 QEMU 的插桩方式, 使用硬件跟踪的方式 (Intel PT) 获取覆盖率信息, 降低执行开销. 在种子变异阶段, 模糊测试器会同步混合执行生成的种子, 并基于关键字节信息进行针对性变异. 在种子调度阶段, 模糊测试器基于静态分析器计算出的复杂度信息, 优先选择能够覆盖复杂路径的种子进行探索, 以提高触发漏洞概率.

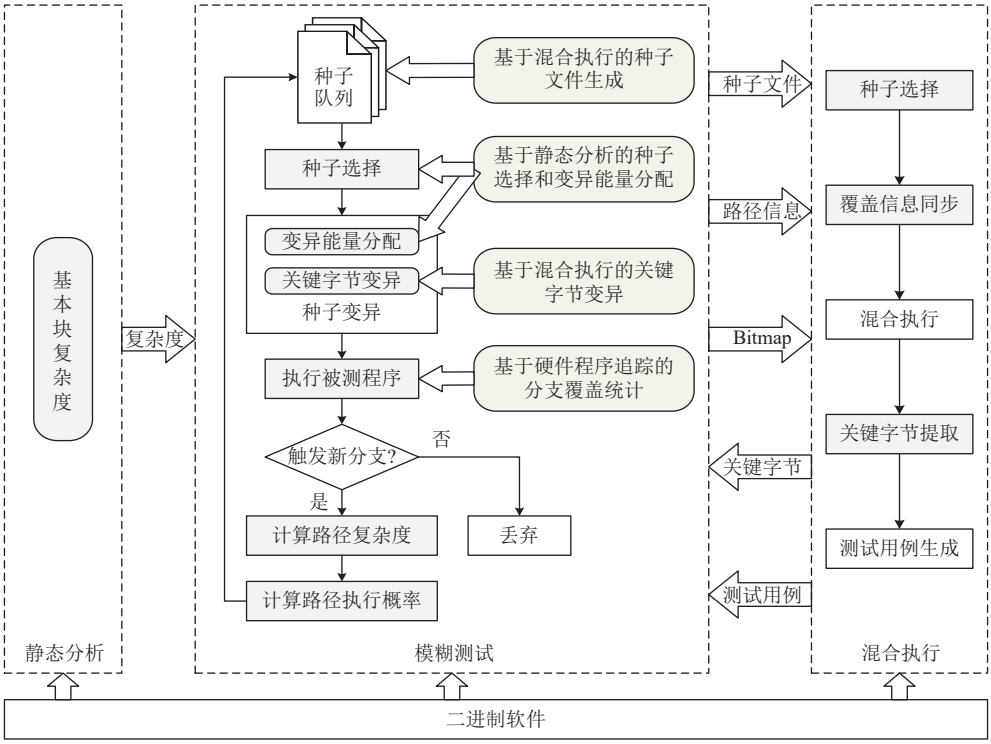

图 1 二进制软件模糊测试的整体架构

本文的创新性主要体现在以下方面.

(1) 在获取覆盖率信息方面, 本文优化原本的基于 QEMU 插桩的覆盖率跟踪方式, 使用基于硬件的跟踪方式, 减小插桩带来的额外开销.

(2) 在种子变异方面, 本文利用混合执行辅助复杂约束的求解, 同时优化混合执行器, 使其能够在执行的过程中记录关键字节信息. 模糊测试器利用关键字节信息进行针对性变异, 降低原生模糊测试器变异方式的盲目性, 提高变异生成的测试用例的质量.

(3) 在种子调度方面, 本文基于对越复杂的路径存在漏洞的概率越大的观察, 设计复合的复杂度计算指标, 在静态分析阶段基于多种方式计算程序路径综合复杂度, 基于复杂度进行种子选择以及能量分配.

2.2 静态分析

在通常情况下, 程序中结构或逻辑越复杂的执行路径越容易出现错误或隐藏漏洞, 优先测试这些复杂路径可以有效提高漏洞挖掘效率. 由于程序路径由若干基本块构成, 本文首先利用静态分析评估被测程序中每个基本块的复杂度, 然后在模糊测试过程中基于基本块复杂度计算路径复杂度, 最后优先测试较复杂的程序路径. 此外, 模糊测试和混合执行擅长处理不同的程序路径, 模糊测试擅长处理限制较宽泛的条件分支 (如 x>1000), 而混合执行擅长处理较具体的条件分支 (如 x==1000). 为了使路径复杂度信息能够分别辅助模糊测试和混合执行进行针对性种子选择, 本文为模糊测试和混合执行设计了不同的复杂度计算方法, 分别为模糊测试复杂度 (fuzzing complexity, FC) 计算和混合执行复杂度 (concolic execution complexity, CEC) 计算.

为了驱动模糊测试对结构和功能更复杂的路径进行优先测试, 在计算模糊测试复杂度 FC 时, 本文考虑了被测程序的结构复杂度和操作复杂度这两个指标.

此外, 考虑到调用有安全风险的库函数可能造成程序安全隐患, 例如不正确地调用 strcpy, memcpy 等函数可能造成缓存区溢出, 本文将调用风险库函数的数量作为评估模糊测试复杂度的第 3 个指标, 记作 VLF (vulnerable function). 具体而言, FC 的计算过程可以分为以下 3 个步骤.

(1) 计算被测程序中每个函数在 3 个评估指标下的复杂度. 本文使用 CC (McCabe's cyclomatic complexity)[9] 和 H.B (Halstead complexity measure) [9]方法分别计算函数的结构复杂度和操作复杂度. 其中, CC 由函数内基本块的数量和基本块间边的数量计算获得, 用于评估函数的结构复杂度; H.B 由函数中操作数和操作符的数量计算获得, 用于评估函数的操作复杂度. 对于 VLF, 本文将微软安全开发[10]过程中禁用的函数作为风险库函数, 统计它们在被测程序的每个函数中被调用的次数.

(2) 将 3 个指标所得的结果进行归一化处理. 由于上述 3 个评估指标具有不同的数量级, 为了消除不同数量级造成的影响, 本文对每个指标进行归一化处理, 使它们的值映射到区间 [0, 1].

(3) 综合归一化处理后的结果. 在得到被测程序中每个函数的 CC、H.B 以及 VLF 的归一化表示后, 本文为它们分配相同的权重, 即利用它们的和作为函数复杂度的估计值. 在程序运行过程中, 函数内部只有部分基本块被实际执行.

$$bb_{\mathrm{fc}}=\frac{norm(f_{\mathrm{CC}})+norm(f_{\mathrm{H.B}})+norm(f_{\mathrm{VLF}})}{bb_{\mathrm{count}}} \tag{1}$$

为了提高模糊测试统计路径复杂度时的精度, 本文通过式 (1) 将每个函数的复杂度除以函数内基本块的数量, 然后使用它们的商作为函数内每个基本块复杂度的估计值. 其中, bb_{fc} 表示基本块的模糊测试复杂度, bb_{count} 表示函数中基本块的数量.

混合执行复杂度 CEC 主要关注被测程序中与路径约束相关的赋值语句、条件跳转语句的数量. 此外, 由于混合执行不擅长处理库函数调用和系统调用, 过多的系统调用会导致混合执行丢失重要约束信息, 因此在计算 CEC 时需要减少库函数调用和系统调用的数量, 使得 CEC 高的程序路径优先使用混合执行求解. 综合上述标准, 本文首先使用以下方法评估被测程序每个基本块的复杂度:

$$bb_{\mathrm{cec}}=\frac{\sqrt{A^2+B^2-C^2}}{bb_{\mathrm{count}}} \tag{2}$$

其中, A 表示函数中赋值语句的数量, B 表示函数中条件跳转语句的数量, C 表示函数中使用的库函数调用和系统调用的数量. bb_{cec} 表示基本块的混合执行复杂度.

2.3 模糊测试

为了对二进制软件进行有效模糊测试, 本文在 AFL 的基础上应用了以下 3 种策略: (1) 通过硬件程序追踪机制 Intel PT 和高速 PT 解码器 libxdc 快速获取程序执行路径, 统计被测程序的分支覆盖信息并为后续优化提供程序路径信息; (2) 利用静态分析计算路径复杂度和路径执行概率, 优先选择复杂度高且执行概率小的程序路径对应的种子文件, 并为这些文件分配更多的变异能量, 以提高模糊测试的路径探索和漏洞发现能力; (3) 采用混合执行生成种子文件并提取种子文件中与程序行为相关的关键字节信息, 在种子变异时优先修改关键字节, 以降低模糊测试变异过程的盲目性.

算法 1 描述了优化后的模糊测试的基本工作流程, 优化方面主要体现: (1) 程序路径提取采用 Intel PT 硬件跟踪技术替换传统的基于 QEMU 插桩的方式, 降低了性能开销; (2) 在进行种子调度时, 基于程序复杂度优先选择能够覆盖复杂路径的种子, 增加了漏洞暴露的概率.

算法 1. 模糊测试的基本工作流程

输入: 目标程序 *P*、初始种子集合 *S*、基本块复杂度 *C*.

```
1. begin
2. Q←Ø; B←Ø; //初始化种子队列 Q 和 bitmap
3. for s in S //执行所有初始种子文件
4.    path = PathExtraction(P, s); //Intel PT 获取执行路径
5.    UpdateBitmap(path, B); //更新 bitmap
6.    pfc, pcec = GetComplexity(path, C); //计算路径复杂度
7.    poss = GetPossibility(path); //计算路径执行概率
8.    PushQueue(Q, s, pfc, pcec, poss); //加入种子队列
9. end for
10. while in the fuzzing loop //模糊测试流程
11.    seed = SeedPrioritization(Q); //种子选择
12.    energy = PowerScheduling(seed); //变异能量分配
13.    for i from 1 to energy
14.       case = Mutation(seed);
15.       path = PathExtraction(P, case);
16.       if HasNewPaths(path, B) then
17.          UpdateBitmap(path, B);
18.          pfc, pcec = GetComplexity(path, C);
```

```
19.          poss = GetPossibility(path);
20.          PushQueue(Q, case, pfc, pcec, poss);
21.       end if
22.    end for
23.    SyncFuzzers(); //同步其他模糊器生成的测试用例
24. end while
25. end
```

具体而言, 对于每个初始的种子文件, 模糊测试以该文件作为输入执行被测程序时, 同时利用 Intel PT 获取程序执行路径对应的基本块序列. 基于该序列, 模糊测试器更新 bitmap, 然后根据基本块复杂度计算相应执行路径模糊测试复杂度 (PFC) 和执行路径混合执行复杂度 (PCEC), 同时计算当前路径被执行的概率. 最后, 模糊测试器将初始种子文件加入到种子队列中. 所有初始种子文件执行完毕后, 模糊测试器基于 PFC 和执行概率从种子队列中选取合适的种子并为其分配变异能量. 在变异过程中, 模糊测试器首先根据混合执行提取到的关键字节信息进行针对性变异, 然后依据 AFL 提供的确定性变异和非确定性变异策略修改种子文件, 生成新用例. 对于新用例, 模糊测试器同样进行执行、筛选和保存流程. 在变异阶段完成后, 模糊测试同步其他模糊器和混合执行生成的测试用, 其中触发新路径的用例将被作为新种子文件保存. 模糊测试不断重复上述步骤, 直到测试完成.

在统计分支覆盖信息方面, 本文利用 Intel PT 记录程序执行的控制流, 然后通过解码控制流对应的 PT 包并结合程序汇编代码, 获得程序的完整执行路径. 相比使用 QEMU 和二进制插桩等方法, Intel PT 能够以较低性能开销记录程序执行的控制流, 对模糊测试器性能的影响较小. 具体而言, 模糊测试器在每次执行被测程序时, Intel PT 将程序执行的控制流信息以 PT 包的形式保存在指定缓存区中. 被测程序执行完毕后, 模糊测试器利用 PT 解码器对缓存区中的 PT 包进行解码, 在解码过程中结合被测程序汇编代码进行分析, 从而得到程序执行路径上每个基本块的地址. 模糊测试器将基本块地址作为更新 bitmap 时的基本块 ID, 然后更新 bitmap 中相应的值, 进而统计分支覆盖情况. 由于 PT 解码过程需要结合程序汇编代码进行分析, 该过程产生较大的时间开销. 为了提高解码速度, 本文使用了 libxdc 开源解码器. 该解码器主要通过哈希缓存和编译优化技术, 降低了解码相同 PT 序列的时间开销, 提高了 Intel PT 的解码速度, 使得模糊测试可以在单位时间内执行更多的测试用例. 此外, 为了支持本文的种子选择和变异能量分配策略, 本文在 libxdc 的基础上增加了程序路径的复杂度计算和执行概率统计功能.

其次, 在计算路径复杂度时, 本文考虑了程序路径模糊测试复杂度 (path fuzzing complexity, PFC) 和程序路径混合执行复杂度 (path concolic execution complexity, PCEC). 路径复杂度的计算基于静态分析得到的程序基本块复杂度和 Intel PT 获取到的程序执行路径. 由于路径复杂度的计算利用了程序的静态分析结果, 因此可以反映程序路径的静态特征. 算法 2 描述了程序路径复杂度的计算过程. 对于执行路径上的每个基本块, 首先记录基本块被执行的次数. 当基本块的执行次数为 2 的次幂时, 根据基本块复杂度累加计算路径复杂度. 当路径上的所有基本块遍历完成后, 返回计算得到的路径复杂度.

算法 2. 程序路径复杂度的计算

输入: 程序执行路径 *path*、基本块复杂度 *C*.
输出: 路径模糊测试复杂度 *pfc*、路径混合执行复杂度 *pcec*.

```
1. function GetComplexity(path, C)
2. pfw = 0;
3. pcew = 0;
4. ETS←Ø; //记录每个基本块的执行次数
5. for each bb in path
6.    ETS[bb] += 1; //基本块的执行次数加 1
7.    if IsPowerOfTwo(ETS[bb]) then
8.       pfc += C[bb].fc; //累加 FC, 获得 PFC
9.       pcec += C[bb].cec; //累加 CEC.获得 PCEC
10.   end if
11. end for
12. return pfc, pcec
13. end
```

被测程序中的循环、递归等操作会导致部分基本块重复出现, 一定程度上干扰路径复杂度的精确性, 使得计算出的路径复杂度偏大. 为缓解这一负面影响, 本文采用了指数退避算法, 算法中只统计了执行次数为 2 的次幂的基本块. 具体而言, 该算法在遍历基本块的同时, 统计基本块被执行的次数, 只有执行次数为 2 的次幂时, 才将基本块复杂度加入路径复杂度. 由于路径复杂度计算需要遍历执行路径上的所有基本块, 造成了较大的时间开销. 为了减少时间花费, 模糊测试器仅在本次执行触发新的程序路径时计算相应的路径复杂度, 从而避免了频繁的复杂度计算过程, 提高了基于路径复杂度的模糊测试的整体运行速度. 具体而言, 在模

糊测试调用被测程序执行测试用例后, 解码器获取程序执行路径并获得本次执行对应的 bitmap. 通过对比 bitmap 判断是否有新路径被触发, 如果测试用例触发了新路径, 则再次以该用例执行程序并利用算法 2 计算本次执行对应的新路径复杂度.

由于模糊测试变异生成的大量用例都执行了相同的高频路径, 导致这些测试用例无法发现新路径而无效化, 因此有必要识别低频路径并对其进行更多的测试, 从而发现其中隐藏的新路径. 为了综合考虑程序在某条执行路径上所表现出的静态特征和动态特征, 本文在路径复杂度的基础上引入了路径的动态执行概率, 使得在种子选择和变异能量分配时能够综合考虑路径复杂度及执行概率. 路径执行概率即为当前程序路径在模糊测试执行被测程序的所有可能路径中被执行的概率, 这是一种测试过程中不断变化的动态特征. 程序执行路径是一条由若干基本块组成的序列. 如果将这条序列上两个相邻基本块构成的边称为分支 (branch), 则路径执行概率 (记作 P) 为该条路径上所有分支被执行概率的乘积. 例如, 由基本块 b1, b2, b3, b4 组成的程序路径 b1→b2→b3→b4 的执行概率为 $P = P(\text{b1}\to\text{b2})\times P(\text{b2}\to\text{b3})\times P(\text{b3}\to\text{b4})$. 假设分支$br'$是一条以基本块$b_s$为起点, 基本块$b_e$为终点的边, $T_{br}(br')$ 表示分支br'被执行的次数, $T_b(b_s)$表示基本块b_s被执行的次数, 则分支br'的执行概率为:

$$P(br') = \frac{T_{br}(br')}{T_b(b_s)},\ br' \in BR(b_s) \tag{3}$$

其中, $BR(b_s)$表示以基本块b_s为起点的所有分支的集合, T_b为基本块被执行的次数, T_{br}为分支被执行的次数. 基于分支执行概率, 程序路径执行概率的计算方法如下:

$$\begin{aligned} P(path) &= \lg\left(\prod_{i=1}^{N} P(br_i)\right) \\ &= \sum_{i=1}^{N} \lg P(br_i),\quad br_i \in path \end{aligned} \tag{4}$$

其中, N 为程序路径中分支的数量, br_i表示某个分支. 由于路径概率小于等于 1, 取对数后的结果小于等于 0, 因此计算结果的相反数越大, 则实际路径概率越小. 为了避免循环、递归等操作产生的重复基本块的影响, 本文使用了指数退避算法, 只有基本块在局部计数器 (即本次执行) 中的执行次数为 2 的次幂时, 将全局计数器的值加 1. 在计算分支执行概率时, 使用全局计数器中的值进行计算.

在种子选择时, 本文综合考虑了种子文件对应执行路径的路径复杂度和路径执行概率, 并结合二者进行种子选择, 综合体现程序路径的静态特征和动态特征. 此外, 由于模糊测试效率受种子文件大小和执行时间的影响, 因此在种子选择时也需要考虑这两点. 综上, 本文中模糊测试进行种子选择的指标如下.

(1) 路径模糊测试复杂度, 记作 *PFC*. 由于复杂的程序路径通常包含复杂的执行逻辑, 在开发过程中越容易引入程序漏洞, 因此优先选择 *PFC* 更高的种子文件, 有助于尽早触发程序 crash.

(2) 路径执行概率, 记作 *P*. 路径执行概率越小, 代表路径中存在复杂约束条件或者模糊测试多次选择了其他高频路径对应的种子文件. 因此, 模糊测试需要优先选择执行概率较小的程序路径, 通过对这些路径进行探索而发现其他新路径, 提高路径覆盖率.

(3) 种子文件大小, 记作 *size*. 种子文件越小代表其压缩性越好, 对小文件进行变异更容易命中影响程序运行的关键字节.

(4) 种子执行时间, 记作 *time*. 种子文件所需的执行时间越短, 则模糊测试单位时间内执行被测程序的次数越多, 模糊测试的速度越快.

综合上述 4 个指标, 本文使用如下公式计算不同执行路径的得分, 得分越高越容易被选中:

$$Score_{\text{fz}}(path) = \frac{\log_2(\varepsilon + PFC)\times\log_2(\varepsilon - P(path)) + \varepsilon}{time \times size} \tag{5}$$

其中, $\varepsilon = 1$, 用于保证计算结果大于 0. 本文以 2 为底对 $\varepsilon + PFC$ 和$\varepsilon - P(path)$取对数, 是为了缩小不同复杂度间和不同路径执行概率间的差距, 使它们的值不会过大或过小, 最终使得 *PFC* 和 *P* 在种子质量评估时所占的比重相近. 由于种子文件的 *time* 和 *size* 变化相对稳定, 因此本文未对它们取对数, 最终在文件大小和执行时间相近时, 根据程序路径的不同复杂度和执行概率选择种子文件.

在变异能量分配时, 本文在 AFL 原有能量分配策略的基础上, 考虑了种子文件对应程序执行路径的路径复杂度 *PFC* 和路径执行概率 *P*. 路径复杂度越大、执行概率越小, 则分配越多的变异能量. 此外, 本文也考虑了模糊测试过程中种子文件被选中的次数, 记作 *cnt*, 种子被选中得越频繁, 则分配越多的变异能量. 在实际

计算中, 由于种子选择过程已经得到了基于 PFC、P、种子文件大小、执行时间 4 个指标的得分$Score_{\text{fz}}(path)$, 模糊测试直接利用$Score_{\text{fz}}(path)$ 对变异能量的得分进行计算, 计算方法如下:

$$Score_{\text{energy}}(path)=\log_2\frac{cnt\times Score_{\text{fz}}(path)\times N}{\sum_{i=1}^{N}Score_{\text{fz}}(path_i)} \tag{6}$$

其中, $Score_{\text{energy}}(path)$表示种子文件对应路径在种子变异时的得分, cnt 为种子文件被选中的次数, N 为种子队列中的种子文件数量. 在计算变异得分时, 本文考虑了当前种子文件对应路径的得分$Score_{\text{fz}}(path)$与所有种子文件得分均值之间的比值, 然后使用 cnt 乘以该比值, 从而评估不同种子文件所需的变异能量大小. 在获得变异得分后, 本文基于该得分计算变异能量. 如果将 AFL 原有的能量分配策略计算得到的变异能量记作$energy'$, 则种子变异在$energy'$的基础上进行缩放, 相应的缩放因子如式 (7) 所示. 其中α表示缩放因子下界, 根据不同实际测试程序设置, β表示缩放因子上界, 依据种子队列被完全遍历次数设置, 设置原则为: 种子队列完全遍历次数越大, 表明测试趋于测试后期, 此时缩放因子增大, 以实现更多变异.

$$\mu=\begin{cases}\alpha, & \text{if } Score_{\text{energy}}(path)<\alpha\\ Score_{\text{energy}}(path), & \text{if } Score_{\text{energy}}(path)\in[\alpha,\beta]\\ \beta, & \text{if } Score_{\text{energy}}(path)>\beta\end{cases} \tag{7}$$

在种子变异阶段, 本文在 AFL 原有变异策略的基础上, 新增了关键字节变异阶段. 其中, 关键字节指种子文件中与程序行为相关的字节. 关键字节变异需要混合执行在求解路径约束时, 记录关键字节在种子文件中的位置以及可能的取值. 通过使用关键字节变异, 一方面可以提高变异过程的针对性, 避免花费大量时间变异种子文件中与程序行为无关的位置;另一方面, 可以在混合执行对复杂路径约束求解失败的情况下, 基于关键字节变异增加满足路径约束条件的可能性. 在种子文件的变异操作开始后, 首先执行关键字节变异, 具体操作分为 7 步, 前 3 步对单个关键字节进行变异, 后 3 步对多个关键字节同时变异, 最后 1 步对非关键字节进行变异, 步骤如下.

(1) 利用混合执行记录的可能取值替换关键字节, 用于测试记录的取值是否可以触发新路径或 crash.

(2) 使用字节最大值、字节最小值等特殊值替换关键字节, 用于测试约束越界的情况.

(3) 使用若干随机值替换关键字节, 用于进行随机测试.

(4) 同时变异相邻关键字节. 相邻关键字节可能共同组成程序输入中的一个值, 同时修改相邻关键字节为混合执行记录的取值或者随机值, 可以测试不同数据类型的影响.

(5) 随机变异部分关键字节. 随机选择若干个关键字节, 然后利用混合执行记录的取值替换这些字节并进行测试, 之后将其中的部分字节替换为随机值, 用于测试部分关键字节是否可以触发新路径或 crash.

(6) 同时变异所有关键字节. 利用混合执行记录的取值替换所有关键字节, 用于测试由所有关键字节构成的被测程序路径.

(7) 变异非关键字节. 首先利用混合执行记录的取值替换所有关键字节, 然后保持关键字节不变, 随机变异非关键字节, 用于测试非关键字节触发程序 crash 的情况.

关键字节变异阶段完成后, 后续变异过程使用 AFL 原有的变异策略. 由于每个种子文件的关键字节位置是确定的, 因此每个种子文件仅需执行一次关键字节变异. 对于每个种子文件, 本文标记其是否已经执行过关键字节变异, 对于执行过该阶段的种子文件, 直接进行后续变异操作.

2.4 混合执行

本文的混合执行引擎基于 QSYM[11], 使用 Intel Pin 工具[12]收集程序执行路径的约束表达式, 然后通过约束取反并求解, 生成覆盖相反分支路径的测试用例. 图 2 展示了本文中混合执行的基本工作流程. 在混合执行开始前, 首先从模糊测试的种子队列中选取种子文件作为混合执行时的输入. 随后将模糊测试的 bitmap 引入混合执行, 使它们的覆盖信息同步, 以减少生成执行重复路径的测试用例. 之后, 利用混合执行引擎 QSYM 执行被测程序并收集程序运行时的路径约束表达式, 生成覆盖相反路径的测试用例. 在混合执行约束求解的同时, 记录种子文件中与程序控制流相关的关键字节位置及可能的取值, 用于关键字节变异. 对于混合执行生成的测试用例, 模糊测试器将其中能触发新路径的测试用例加入种子队列.

为了提高模糊测试与混合执行的协同性, 本文基于 QSYM 进行优化, 优化方面体现在: (1) 覆盖率信息统计优化, 为避免覆盖率信息不同步造成的生成重复

种子的问题, 本文将模糊测试器覆盖率信息与混合执行器覆盖率信息进行同步; (2) 为提高模糊测试器变异效率, 在进行混合执行时, 本方案跟踪程序中的关键字节信息, 用于指导模糊测试器提供有效的变异.

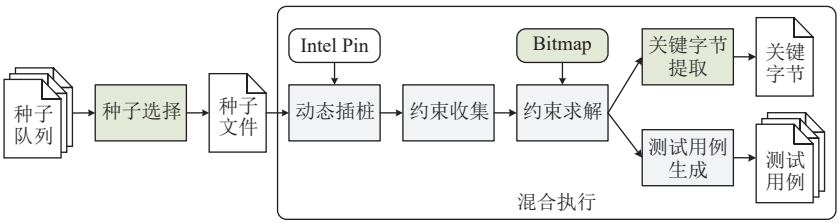

图 2 混合执行的基本流程

在种子选择时, QSYM 从未执行过的种子文件中优先选择能够触发新分支的、较小的、新生成的种子文件. 当种子队列中存在大量新生成的小文件时, 该方法可能导致部分较大的种子文件长时间无法被执行. 然而, 这些较大的种子文件可能包含更多的路径信息, 更适合利用混合执行覆盖相关路径. 例如, 对于存在复杂约束的执行路径, 模糊测试的变异操作难以满足约束条件, 可能导致测试用例重复执行相同路径, 这种情况下, 混合执行可以通过求解执行概率较大的程序路径对应的约束条件, 从而生成覆盖执行概率较小的相反分支路径的测试用例. 此外, 本文根据混合执行擅长处理条件分支而对系统调用支持较弱的特性, 评估获得了程序路径的混合执行复杂度, 该复杂度表示程序路径对于混合执行的适宜程度, 应该优先选择混合执行复杂度高的程序路径对应的种子文件. 综上所述, 本文在 QSYM 的基础上考虑了种子文件对应执行路径的混合执行复杂度 ($PCEC$) 和路径执行概率 (P), 最终种子选择的综合得分如下:

$$Score_{\text{ce}}(path)=\frac{\log_2 PCEC}{\log_2(-P(path)\times size+\varepsilon)+\varepsilon} \tag{8}$$

其中, ε=1 用于保证计算过程中$(-P(path)\times size+\varepsilon)\geqslant 1$且除数不为 0. 基于上述得分, 混合执行在进行种子选择时的具体步骤如下: (1) 找到所有尚未执行的种子文件; (2) 在未执行的种子文件中, 优先选择触发新分支的种子文件; (3) 对于所有触发新分支的种子文件, 优先选择$Score_{\text{ce}}$较大的; 如果没有种子文件触发新分支, 则直接比较种子文件的$Score_{\text{ce}}$; (4) 如果种子文件的$Score_{\text{ce}}$相同, 则优先选择新生成的种子文件. 基于上述步骤, 混合执行可以优先选择触发新分支的、$Score_{\text{ce}}$较大的、新生成的种子文件.

覆盖信息同步, 指同步模糊测试和混合执行中用于记录被测程序覆盖信息的 bitmap. 在测试过程中, 模糊测试和混合执行利用 bitmap 判断执行路径是否为新路径, 混合执行对新路径的约束条件取反并求解以生成覆盖相反路径的测试用例. 由于混合执行引擎 QSYM 独立维护 bitmap, 可能导致 QSYM 对已经执行过的路径进行约束求解而生成覆盖重复路径的测试用例. 因此, 本文在混合执行开始前, 将模糊测试的 bitmap 引入 QSYM, 与 QSYM 原有的 bitmap 同步, 使得 QSYM 同时拥有模糊测试和混合执行的分支覆盖信息. 在遇到条件语句时, QSYM 首先检查模糊测试的 bitmap, 如果条件语句中存在未被执行的分支, 则对当前路径约束取反并求解, 从而可以有效避免混合执行在模糊测试已经覆盖过的路径上求解, 避免生成大量低效测试用例, 提高了模糊测试和混合执行的协同性.

在混合执行过程中进行关键字节提取, 主要用于解决模糊测试的变异操作存在盲目性、生成大量低效测试用例的问题. 在通常情况下, 种子文件中只有特定位置的字节与程序行为相关, 而模糊测试的变异策略无法确定这些关键位置并进行针对性变异, 导致其在与程序行为无关的位置花费了较多的变异时间, 影响测试效率. 借助混合执行引擎 QSYM, 可以追踪种子文件的每个字节在程序执行过程中的传播行为, 从而收集路径约束并提取种子文件中与程序路径相关的关键字节, 然后将关键字节信息提供给模糊测试以进行针对性变异. 通过关键字节变异, 一方面可以提高模糊测试变异过程的针对性, 另一方面可以在混合执行求解复杂路径约束失败的情况下, 尝试通过关键字节变异生成满足路径约束的测试用例.

3 实验与分析

3.1 实验说明

由于模糊测试工具发现的 unique crash 数目通常比真实存在的 bug 数目多, 导致模糊测试工具性能被

高估, 故在评估过程中, 本文选择使用 LAVA-M[13]数据集, 该数据集通过向正常程序中插入已知 bug 构建, 因而能够较准确地评估出模糊测试工具性能; 同时, 为了验证本文提出的方法在真实程序上的适用性. 如表 1 所示, 本文测试了 Linux 下的 8 个真实二进制程序. 这些程序涵盖广泛的功能和输入格式, 包括汇编代码编译器, 二进制工具集 GNU Binutils 中的部分分析工具, 图像处理工具, 以及 PDF 处理工具, 可以用于评估 BPAFuzz 在真实程序中的效果. 由于真实程序区别于数据集, bug 数目难以统计, 故采用 crash 数目作为评估指标. 同时, 为了体现模糊测试器的工程应用, 本文实验主要体现在真实程序中.

表 1 实验中使用的真实程序

程序名	版本	输入格式	功能
nasm	2.15rc0	汇编代码	汇编编译器
nm	2.23	二进制代码	列出文件符号表
size	2.23	二进制代码	显示文件的节大小
objdump	2.23	二进制代码	显示文件信息
strings	2.23	二进制代码	列出可打印字符串
jhead	2.97	JPEG文件	JPEG文件编辑工具
tiff2ps	3.9.7	TIFF文件	TIFF转换为PS文件
pdftotext	4.02	PDF文件	PDF转换TEXT文本

为了评估不同二进制软件模糊测试方案的效果本文使用了如下两个评估指标: (1) 模糊测试在不同程序发现的分支数, 用于评估不同方案的路径覆盖情况; (2) 模糊测试在不同程序发现的 crash 数, 用于评估不同方案的漏洞发现能力. 基于上述评估指标, 本文为 BPAFuzz 逐个开启文中提出的解决方案, 从而比较不同方案的影响. 本文实验方案分为以下部分: (1) 验证静态分析策略的有效性. 本文开启基于静态分析的种子选择和变异能量分配策略, 然后与 AFLFast、PTfuzz[14]、仅开启硬件程序追踪的 BPAFuzz 进行对比, 从而评估种子选择和变异能量分配策略对于模糊测试效果的影响; (2) 验证基于混合执行的种子文件生成和关键字节变异策略的有效性. 本文比较了 3 种情况下混合执行对 BPAFuzz 的辅助效果, 分别为: 原始的 QSYM 辅助仅开启硬件程序追踪的 BPAFuzz、开启覆盖信息同步和关键字节提取的 QSYM 辅助开启硬件程序追踪和关键字节变异的 BPAFuzz、开启所有策略的 BPAFuzz, 从而评估本文提出的混合执行策略对于模糊测试的辅助效果.

由于模糊测试具有较强的随机性, 因此本文使用相同初始种子文件在真实程序与 LAVA-M 数据集上进行重复实验, 每轮实验测试 24 h, 最终实验解决为 5 次实验后的结果取算数平均值.

本文实验在如下环境进行: 主机 CPU 采用 Intel(R) Core(TM) i7-9700K CPU @ 3.60 GHz; 内存大小为 32 GB; 操作系统为 Ubuntu 16.04.

3.2 静态分析策略的有效性

为了评估基于静态分析的种子选择和变异能量分配策略的有效性, 本文在开启硬件程序追踪、静态分析策略的前提下, 将 BPAFuzz 与 AFLFast、PTfuzz 在真实程序上进行比较. 其中, 为了控制实验变量, 本文为 AFLFast 添加了硬件程序追踪支持, 记作 AFLFast (PT), 从而在相同的分支覆盖统计策略下, 比较不同种子选择和变异能量分配策略的效果. 为了方便描述, 本文将基于硬件程序追踪和静态分析的 BPAFuzz 记作 BPAFuzz (PTSD), 仅开启硬件程序追踪的 BPAFuzz 记作 BPAFuzz (PT).

本文在 8 个真实程序上统计 BPAFuzz (PT)、AFLFast (PT) 、BPAFuzz (PTSD) 和 PTfuzz 在每个程序中发现的分支数和 crash 数, 实验结果如表 2 所示. 实验发现, BPAFuzz (PTSD) 总计发现了 24 811 个分支和 62 个 crash. 与 BPAFuzz (PT) 相比, BPAFuzz (PTSD) 的分支数增加了 0.42%, crash 数增加了 6.90%, 尤其在程序 strings 上多发现了 60% 的 crash. 与 AFLFast (PT) 相比, BPAFuzz (PTSD) 发现的分支数增加了 3.11%, crash 数增加了 29.17%, 并且在程序 nm, size 和 strings 上分别多发现了 28.57%, 166.67% 和 60% 的 crash. 与 PTfuzz 相比, BPAFuzz (PT) 发现分支数增加 6.48%, 发现 crash 数目增加 1.75%, 而 BPAFuzz (PTSD) 发现分支数增加 6.93%, 发现 crash 数目增加 8.77%. 但是, BPAFuzz (PTSD) 在程序 nm、objdump 和 pdftotext 上发现的分支较少, 这是因为 BPAFuzz (PTSD) 为了优先执行可能触发漏洞的复杂程序路径, 需要统计路径复杂度和执行概率, 影响了执行速度. 对于程序 nasm 和 size, BPAFuzz (PTSD) 发现的 crash 较少, 可能是因为其在相同的测试时间内扩展了路径探索的广度, 发现了较多分支, 导致无法兼顾隐藏在程序路径深处的部分 crash. 因此, 综合考虑路径复杂度和执行概率进行种子选择和变异能量分配, 可以在多数程序中发现更多的分支, 而且更擅长发现程序 crash, 说明了路径复杂度和执行概率对于种子选择和变异能量分配的有效性.

表 2 不同种子选择和变异能量分配策略在真实程序中发现的分支数和 crash 数

程序名	BPAFuzz (PT)		AFLFast (PT)		BPAFuzz (PTSD)		PTFuzz	
	分支数	Crash数	分支数	Crash数	分支数	Crash数	分支数	Crash数
Nasm	2832	3	2892	4	2901	3	2780	3
nm	3011	17	2984	14	2880	18	2746	17
size	3067	9	2824	3	3148	8	3016	9
objdump	5612	12	5178	10	5577	12	5164	10
strings	2395	5	2259	5	2448	8	2135	5
jhead	379	0	380	1	380	1	378	0
tiff2ps	504	0	504	0	509	0	504	0
pdftotext	6906	12	7041	11	6968	12	6480	13
total	24706	58	24062	48	24811	62	23203	57

3.3 混合执行策略的有效性

为了评估基于混合执行的种子文件生成和关键字节变异策略的有效性, 本文比较了 3 种情况下混合执行的辅助效果, 分别为: (1) 利用混合执行引擎 QSYM 的原始策略, 辅助仅开启硬件程序追踪的模糊测试, 记作 BPAFuzz (PT-QSYM); (2) 利用开启覆盖信息同步和关键字节提取的混合执行引擎, 辅助开启硬件程序追踪和关键字节变异的模糊测试, 记作 BPAFuzz (PT-Bytes); (3) 集成本文所有解决方案, 包括使用硬件程序追踪、静态分析和混合执行辅助模糊测试, 记作 BPAFuzz (All).

表 3 展示了不同混合执行策略在真实程序中发现的分支数和 crash 数. 从表 3 中可以发现, BPAFuzz (PT-Bytes) 总计发现了个 25 891 分支和 90 个 crash, 比 BPAFuzz (PT-QSYM) 多发现了 0.99% 的分支和 3.45% 的 crash. 对于分支数, BPAFuzz (PT-Bytes) 在 6 个程序上发现的分支数多于 BPAFuzz (PT-QSYM), 特别在程序 jhead 上多发现了 8.77% 的分支; 对于 crash 数, BPAFuzz (PT-Bytes) 在 3 个程序上发现的 crash 数多于 BPAFuzz (PT-QSYM), 在 4 个程序上发现的 crash 数与 BPAFuzz (PT-QSYM) 相同, 尤其在程序 strings 和 jhead 上多发现了 25.00% 和 11.76% 的 crash. 但是, BPAFuzz (PT-Bytes) 在 strings 和 tiff2ps 上发现的分支较少, 可能的原因一方面是由于 BPAFuzz (PT-Bytes) 花费了较多时间进行关键字节变异, 导致其发现的分支数减少, 但是在程序 strings 上发现的 crash 更多; 另一方面是由于模糊测试的分支覆盖统计策略对于程序路径不敏感, 导致混合执行的覆盖信息同步策略在减少重复路径的同时, 减少了混合执行对新路径求解的可能性, 导致 BPAFuzz (PTBytes) 在程序 strings 及 tiff2ps 上发现的分支较少. 从整体效果而言, 本文中基于混合执行的种子文件生成和关键字节变异策略, 可以在多数程序中发现更多的分支和 crash, 说明其可以提高模糊测试发现程序路径和 crash 的能力.

表 3 不同混合执行策略在真实程序中发现的分支数和 crash 数

程序名	BPAFuzz (PT-QSYM)		BPAFuzz (PT-Bytes)		BPAFuzz (All)	
	分支数	Crash数	分支数	Crash数	分支数	Crash数
Nasm	2859	4	2889	4	2897	4
nm	3053	23	3111	23	3068	22
size	3220	13	3281	12	3309	13
objdump	5549	13	5581	14	5645	14
strings	2599	4	2597	5	2466	6
jhead	787	17	856	19	861	19
tiff2ps	572	0	555	0	555	0
pdftotext	6998	13	7021	13	7076	14
total	25637	87	25891	90	25877	92

在表 3 中, 集成本文所有策略的 BPAFuzz (All) 总计发现 25 877 个分支和 92 个 crash, 相比 BPAFuzz (PT-QSYM) 多发现了 0.94% 的分支和 5.75% 的 crash, 比 BPAFuzz (PT-Bytes) 多发现了 2.22% 的 crash. 虽然 BPAFuzz (All) 比 BPAFuzz (PT-Bytes) 发现的总分支数减少, 但是其在 5 个程序上发现的分支数多于 BPAFuzz (PT-Bytes), 仅在程序 nm 和 strings 上发现的分支数较少, 这是因为 BPAFuzz (All) 在 BPAFuzz (PT-Bytes) 的基础上增加了基于静态分析的种子选择和变异能量分配策略并优化了混合执行的种子选择过程, 这些策略更关注发现被测程序的潜在漏洞, 因此 BPAFuzz (All) 在这些程序上发现的 crash 数量增加而分支数量减少. 尤其对于程序 jhead, BPAFuzz (All) 发现的 crash 数量比 BPAFuzz (PT-QSYM) 多 50%, 比 BPAFuzz (PT-Bytes) 多 20%. 但是, BPAFuzz (All) 在 nm 上发现的 crash 较少, 这是因为每种方案对于不同程序的适用程度不同, 单个方案很难在所有程序中均获得更好的效果, 而本文方案对于 nm 的适用性可能较低. 综合实验

结果可以说明, 结合本文所有方案进行二进制软件模糊测试, 可以在多数情况下发现更多 crash, 具有更好的漏洞发现能力.

表 4 展示了不同策略在 LAVA-M 中发现的 bug 数量. 表中总 bug 数为 LAVA-M 在程序中植入的 bug 总数. 从表 4 中结果可知, 开启文中所有策略的 BPAFuzz (All) 在这些程序中发现的 bug 数量最多, 且在每个程序发现的 bug 数都等于或优于 BPAFuzz (PT-QSYM) 和 BPAFuzz (PT-Bytes). BPAFuzz (All) 在 4 个程序中发现的 bug 数分别占 LAVA-M 列出 bug 总数的 109.09%、15.79%、100.00%、18.02%, 尤其在 base64 和 uniq 中发现了 LAVA-M 列出的所有 bug, 而且在 base64 中额外发现了 4 个 LAVA-M 没有列出的 bug. 对于开启关键字节变异的 BPAFuzz (PT-Bytes), 同样在 base64 和 uniq 中发现了所有 bug, 并且比 QSYM 的原始策略在 md5sum 和 who 上发现得更多. 综上说明关键字节变异可以辅助模糊测试发现更多的潜在漏洞. 实验结果说明, 使用覆盖信息同步和关键字节变异策略后, 混合执行可以更有效地辅助模糊测试探索发现程序中的隐藏 bug, 而且综合使用本文所有策略进行二进制软件模糊测试相比使用单一策略在 LAVA-M 上具有更好的漏洞发现能力.

表 4 不同混合执行策略在 LAVA-M 中发现的 bug 数

程序名	总bug数	BPAFuzz (PT-QSYM)	BPAFuzz (PT-Bytes)	BPAFuzz (All)
base64	44	48	48	48
md5sum	57	2	7	9
uniq	28	28	28	28
who	2 136	325	328	385
total	2 265	403	411	470

4 讨论与展望

虽然本文对二进制软件模糊测试的针对性、路径探索能力、漏洞发现能力 3 个方面进行了优化, 但是本文方案也存在一些不足. 在评估基本块复杂度时, 本文使用函数复杂度除以基本块数量的均值作为基本块复杂度的估计值, 虽然这样可以保留结构信息且避免忽略简单基本块, 但是该方法存在一定误差, 后续研究可以对函数中相对复杂的基本块赋予更高的复杂度, 增强基本块复杂度计算的准确性. 此外, 在计算路径复杂度和执行概率时, 本文需要重新执行被测程序并获取程序执行路径, 影响了模糊测试整体吞吐量, 未来可以将基本块复杂度和执行次数加入 PT 解码器缓存中, 然后从缓存直接读取基本块信息并计算程序路径复杂度和执行概率, 从而减少重复执行和收集带来的时间开销.

5 相关工作

针对传统模糊测试方法存在盲目性、随机性, 生成的测试用例质量不高以及测试效率低等问题, 研究人员从基于硬件追踪的模糊测试、测试过程优化以及程序分析辅助模糊测试等方面进行相关改进.

(1) 基于硬件追踪的模糊测试

在二进制软件模糊测试时, AFL[4]借助 QEMU[5]模拟执行程序以收集分支覆盖信息. 然而, 利用 QEMU 进行二进制软件模糊测试, 存在速度慢、效率低的问题. 随着硬件技术的发展, 近年来出现了基于 Intel PT[7]的二进制软件模糊测试研究.

kAFL[15]结合虚拟化技术和 Intel PT 对操作系统内核进行模糊测试, 并发现了多个内核错误. PTfuzz[13]利用 Intel PT 获得二进制软件运行时经过的基本块地址, 从而辅助 AFL 统计分支覆盖信息. 通过将基本块地址作为分支覆盖索引, 相比之下, PTfuzz 缓解了 AFL 在统计分支覆盖时存在的哈希碰撞问题, 提高了模糊测试的精度. PTrix[6]通过并行执行 PT 包的记录和解码过程, 以及直接利用 PT 包生成分支覆盖信息而无需反汇编的方式, 解决了引入 Intel PT 的模糊测试工具在解码 PT 包时需要较大时间开销这一问题, 降低了 PT 包解码对模糊测试性能的影响.

μAFL[16]利用 ARM 处理器中的 ETM 硬件追踪机制对固件程序进行测试, 在实际的测试环境中取得了一定的效果.

(2) 测试过程优化

测试过程优化集中体现在种子选择、变异策略以及被测程序覆盖率等方面的优化. 通过对测试过程中相关策略等的优化, 能较好地解决模糊测试盲目性、随机性强的问题.

AFLFast[8]在 AFL 的基础上利用马尔可夫链 (Markov chain) 识别低频路径, 然后为低频路径对应的种子文件分配更多的变异能量, 从而提高了 AFL 的路径覆盖率. MOPT[17]使用粒子群算法计算变异操作的最优概率分布, 从而寻找并选择更合适的变异操作, 使得模糊测试更快速地生成有效种子文件. REDQUEEN[18]基于

多数情况下程序输入和运行状态直接相关这一发现, 通过提取程序控制流相关的指令信息以及变异、着色等策略, 获得被测程序的 Magic Bytes 及校验和对应的字节, 辅助模糊测试通过程序条件检查, 降低了变异的盲目性, 并在 LAVA-M[12]数据集上相较于其他模糊测试器表现突出, 仅遗漏开发者列出的 2 个 bug, 同时在真实程序测试中的表现也由于其他模糊测试器. Cluzz[19]结合种子执行路径覆盖的分布来分析种子在特征空间上的区别, 使用聚类分析对种子在程序空间中的执行分布情况进行划分, 根据不同种子簇群的路径覆盖模式与聚类分析结果对种子进行优先级评估, 探索稀有代码区域并优先调度评估得分较高的种子. Jigsaw[20]针对累积型缺陷模糊测试对应的状态特征值最优化问题提出一种对特征值依赖的输入数据的格式判别和差分变异方法, 提升累积型缺陷的复现和定向测试效率. HMFuzzer[21]通过在模糊测试的预处理、测试和结果分析阶段引入专家经验, 利用上一阶段获取的关键信息, 结合强化学习算法, 优化种子变异和模糊测试流程, 提升了模糊测试的覆盖率、效率以及漏洞挖掘能力.

(3) 基于程序分析的模糊测试

程序分析技术包括静态分析及符号执行、混合执行等技术. 程序分析技术提供的代码复杂度、程序覆盖率等信息用于指导模糊测试的种子选择及测试用例变异阶段, 为模糊测试生成高质量测试用例提供帮助.

Shudrak 等人[22]首先使用 IDA Pro 逆向分析工具对程序汇编代码进行复杂度度量, 获得程序中复杂度高的函数, 然后使用模糊测试优先选择和变异覆盖复杂度高的函数的测试用例. Cerebro[9]将静态分析得到的函数复杂度与模糊测试得到的程序执行信息相结合, 使用基于多目标的种子文件选择算法, 综合评估代码复杂度、覆盖率、执行时间和文件大小等指标, 从而指导模糊测试的种子选择和变异能量分配过程.

SAFL[23]使用符号执行工具 KLEE[24]生成 AFL 的初始种子文件, 然后通过模糊测试时优先选择低频路径和变异时保留约束条件的策略, 减少了重复变异的时间开销并提高了路径覆盖率. Driller[25]在 AFL 遇到瓶颈时, 采用符号执行同具体执行相结合的混合执行技术, 将 AFL 的种子文件作为混合执行输入, 然后生成满足复杂路径约束的测试用例, 从而辅助 AFL 探索更多程序路径. SymCC[26]提出了基于编译的混合执行方法, 通过将混合执行代码直接编译到程序中, 提高了混合执行的性能. QSYM[9]使用动态二进制插桩工具 Intel Pin 收集程序运行时的路径约束表达式, 然后通过对该表达式取反并求解, 从而生成可以触发新路径的测试用例. 通过使用 Intel Pin 进行指令级混合执行, 相比前 3 种方式, QSYM 避免了低效的中间表示和状态保存等过程, 提高了混合执行效率.

DigFuzz[27]则同时应用静态分析与混合执行技术辅助模糊测试用例的生成. 首先使用静态分析获得程序的控制流图, 然后在模糊测试过程中动态计算程序路径被执行的概率, 最后选择执行概率较低的路径交给混合执行处理, 从而生成能触发低频路径的测试用例.

SAVIOR[28]区别于传统的覆盖率导向混合执行, 使用漏洞导向指导测试过程. 具体而言, 在编译阶段, SAVIOR 利用 UBSan 对程序脆弱点进行标记, 之后使用静态分析信息确定优先混合执行的种子; 测试过程中按照能够访问更多脆弱性标记的原则对种子进行排序. 确保混合执行能够对路径上可能存在的漏洞进行全面验证. Intriguer[29]针对混合执行资源消耗原因进行研究, 提出一种字段级别的约束求解, 利用污点分析技术构建字段转移树, 依据树深度分配求解器资源, 确保混合执行资源分配的合理性.

Pangolin[30]针对混合执行中计算冗余问题进行优化, 提出多边形路径摘要, 利用历史路径分支求解信息, 缩小后续分支求解搜索空间, 加快求解速度.

基于上述多种改进方案, 本文同时结合硬件追踪、静态分析以及混合执行多种技术来辅助模糊测试的进行. 相较于上述技术, 本文能够获取更多优化模糊测试的信息, 如基本块复杂度、路径复杂度、程序覆盖率等信息, 从而进一步提升模糊测试生成测试用例的质量以及模糊测试效率.

6 总结

本文针对二进制软件模糊测试的效率较低、种子选择和种子变异过程具有随机性的问题, 提出了综合利用硬件程序追踪、静态分析和混合执行 3 种程序分析技术辅助进行二进制软件模糊测试的解决方案. 本方案以模糊测试为核心, 基于硬件程序追踪机制辅助模糊测试进行程序分支覆盖统计, 通过静态分析为模糊测试提供程序基本块复杂度信息, 利用混合执行为模糊测试提供满足程序路径约束条件的种子文件以及

种子文件中的关键字节信息. 本方案可以增强二进制软件模糊测试的针对性, 缓解模糊测试无法深入程序较深路径的问题, 提升模糊测试的路径覆盖率和漏洞发现能力. 此外, 本方案原型系统 BPAFuzz 能够针对二进制商用软件进行模糊测试, 具备发现商用软件中未知漏洞的能力.

参考文献

1 邹权臣, 张涛, 吴润浦, 等. 从自动化到智能化: 软件漏洞挖掘技术进展. 清华大学学报 (自然科学版), 2018, 58(12): 1079–1094 [doi: 10.16511/j.cnki.qhdxxb.2018.21.025]

2 刘剑, 苏璞睿, 杨珉, 等. 软件与网络安全研究综述. 软件学报, 2018, 29(1): 42–68 [doi: 10.13328/j.cnki.jos.005320]

3 Li J, Zhao BD, Zhang C. Fuzzing: A survey. Cybersecurity, 2018, 1(1): 6. [doi: 10.1186/s42400-018-0002-y]

4 Zalewski M. American fuzzy lop. http://lcamtuf.coredump.cx/afl/. [2024-06-20].

5 Bellard F. QEMU, a fast and portable dynamic translator. Proceedings of the 2005 Annual Conference on USENIX Annual Technical Conference. Anaheim: USENIX Association, 2005. 41.

6 Chen YH, Mu DL, Xu J, *et al*. PTrix: Efficient hardware-assisted fuzzing for cots binary. Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security. Auckland: ACM, 2019. 633–645.

7 Processor tracing. https://software.intel.com/en-us/blogs/2013/09/18/processor-tracing. (2013-09-18).

8 Böhme M, Pham VT, Roychoudhury A. Coverage-based greybox fuzzing as Markov chain. IEEE Transactions on Software Engineering, 2019, 45(5): 489–506. [doi: 10.1109/TSE.2017.2785841]

9 Li YK, Xue YX, Chen HX, *et al*. Cerebro: Context-aware adaptive fuzzing for effective vulnerability detection. Proceedings of the 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. Tallinn: ACM, 2019. 533–544.

10 Howard M. Security development lifecycle (SDL) banned function calls. https://msdn.microsoft.com/enus/library/bb288454.aspx. (2012-06-12).

11 Yun I, Lee S, Xu M, *et al*. QSYM: A practical concolic execution engine tailored for hybrid fuzzing. Proceedings of the 27th USENIX Security Symposium. Baltimore: USENIX Security Symposium, 2018. 745–761.

12 Luk CK, Cohn R, Muth R, *et al*. Pin: Building customized program analysis tools with dynamic instrumentation. Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation. Chicago: Association for Computing Machinery, 2005. 190–200.

13 Dolan-Gavitt B, Hulin P, Kirda E, *et al*. LAVA: Large-scale automated vulnerability addition. Proceedings of the 2016 IEEE Symposium on Security and Privacy (SP). San Jose: IEEE, 2016. 110–121.

14 Zhang G, Zhou X, Luo YQ, *et al*. PTfuzz: Guided fuzzing with processor trace feedback. IEEE Access, 2018, 6: 37302–37313. [doi: 10.1109/ACCESS.2018.2851237]

15 Schumilo S, Aschermann C, Gawlik R, *et al*. kAFL: Hardware-assisted feedback fuzzing for OS kernels. Proceedings of the 26th USENIX Conference on Security Symposium. Vancouver: USENIX Association, 2017. 167–182.

16 Li WQ, Shi JM, Li FJ, *et al*. µAFL: Non-intrusive feedback-driven fuzzing for microcontroller firmware. Proceedings of the 44th IEEE/ACM International Conference on Software Engineering (ICSE). Pittsburgh: IEEE, 2022. 1–12.

17 Lyu CY, Ji SL, Zhang C, *et al*. MOPT: Optimized mutation scheduling for fuzzers. Proceedings of the 28th USENIX Security Symposium. Santa Clara: USENIX Security Symposium, 2019. 1949–1966.

18 Aschermann C, Schumilo S, Blazytko T, *et al*. REDQUEEN: Fuzzing with input-to-state correspondence. Proceedings of the 26th Annual Network and Distributed System Security Symposium. San Diego: NDSS, 2019. 1–15.

19 张文, 陈锦富, 蔡赛华, 等. 一种聚类分析驱动种子调度的模糊测试方法. 软件学报, 2024, 35(7): 3141–3161. [doi: 10.13328/j.cnki.jos.007105]

20 杨克, 贺也平, 马恒太, 等. 面向递增累积型缺陷的灰盒模糊测试变异优化. 软件学报, 2023, 34(5): 2286–2299. [doi: 10.13328/j.cnki.jos.006491]

21 况博裕, 张兆博, 杨善权, 等. HMFuzzer: 一种基于人机协同的物联网设备固件漏洞挖掘方案. 计算机学报, 2024, 47(3): 703–716. [doi: 10.11897/SP.J.1016.2024.00703]

22 Shudrak MO, Zolotarev VV. Improving fuzzing using software complexity metrics. Proceedings of the 18th International Conference on Information Security and Cryptology. Seoul: Springer, 2015. 246–261.

23 Wang MZ, Liang J, Chen YL, *et al*. SAFL: Increasing and accelerating testing coverage with symbolic execution and guided fuzzing. Proceedings of the 40th IEEE/ACM International Conference on Software Engineering:

Companion. Gothenburg: IEEE, 2018. 61–64.

24 Cadar C, Dunbar D, Engler D. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation. San Diego: USENIX Association, 2008. 209–224.

25 Stephens N, Grosen J, Salls C, *et al*. Driller: Augmenting fuzzing through selective symbolic execution. Proceedings of the 23rd Annual Network and Distributed System Security Symposium. San Diego: NDSS, 2016. 1–16.

26 Poeplau S, Francillon A. Symbolic execution with SymCC: Don’t interpret, compile! Proceedings of the 29th USENIX Conference on Security Symposium. USENIX Association, 2020. 11.

27 Zhao L, Duan Y, Yin H, *et al*. Send hardest problems my way: Probabilistic path prioritization for hybrid fuzzing. Proceedings of the 26th Annual Network and Distributed System Security Symposium. San Diego: NDSS. 2019.

28 Chen YH, Li P, Xu J, *et al*. SAVIOR: Towards bug-driven hybrid testing. Proceedings of the 2020 IEEE Symposium on Security and Privacy (SP). San Francisco: IEEE, 2020. 1580–1596.

29 Cho M, Kim S, Kwon T. Intriguer: Field-level constraint solving for hybrid fuzzing. Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security. London: ACM, 2019. 515–530.

30 Huang HQ, Yao PS, Wu RX, *et al*. Pangolin: Incremental hybrid fuzzing with polyhedral path abstraction. Proceedings of the 2020 IEEE Symposium on Security and Privacy (SP). San Francisco: IEEE, 2020. 1613–1627.

(校对责编: 孙君艳)