软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2019,30(10):3071−3089 [doi: 10.13328/j.cnki.jos.005789]

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel: +86-10-62562563

基于符号执行与模糊测试的混合测试方法*

谢肖飞[1], 李晓红[1], 陈 翔[2], 孟国柱[3], 刘 杨[4]

[1](天津市先进网络重点实验室(天津大学),天津 300050)

[2](南通大学 计算机科学与技术学院,江苏 南通 226019)

[3](信息安全国家重点研究室(中国科学院 信息工程研究所),北京 100093)

[4](School of Computer Science and Engineering, Nanyang Technological University 639798, Singapore)

通讯作者: 李晓红, E-mail: xiaohongli@tju.edu.cn

摘 要: 软件测试是保障软件质量的常用方法,如何获得高覆盖率是测试中十分重要且具有挑战性的研究问题.模糊测试与符号执行作为两大主流测试技术已被广泛研究并应用到学术界与工业界中,这两种技术都具有一定的优缺点:模糊测试随机变异生成测试用例并动态执行程序,可以执行并覆盖到较深的分支,但其很难通过变异的方法生成覆盖到复杂条件分支的测试用例.而符号执行依赖约束求解器,可以生成覆盖复杂条件分支的测试用例,但在符号化执行过程中往往会出现状态爆炸问题,因此很难覆盖到较深的分支.有工作已经证明,将符号执行与模糊测试相结合可以获得比单独使用模糊测试或者符号执行更好的效果.分析符号执行与模糊测试的优缺点,提出了一种基于分支覆盖将两种方法结合的混合测试方法——Afleer,结合双方优点从而可以生成具有更高分支覆盖率的测试用例.具体来说,模糊测试(例如 AFL)为程序快速生成大量可以覆盖较深分支的测试用例,符号执行(例如 KLEE)基于模糊测试的覆盖信息进行搜索,仅为未覆盖到的分支生成测试用例.为了验证 Afleer 的有效性,选取标准程序集 LAVA-M 以及实际项目 oSIP 作为评测对象,以漏洞检测能力以及覆盖能力作为评测指标.实验结果表明:(1) 在漏洞检测能力上,Afleer 总共可以发现 755 个漏洞,而 AFL 仅发现 1 个;(2) 在覆盖能力上,Afleer 在标准程序集上以及实际项目中都有不同程度的提升.其中,在 oSIP 中,Afleer 比 AFL 在分支覆盖率上提高 2.4 倍,在路径覆盖率上提升 6.1 倍.除此之外,Afleer 在 oSIP 中还检测出一个新的漏洞.

关键词: 软件质量保障;模糊测试;符号执行;测试用例生成

中图法分类号: TP311

中文引用格式: 谢肖飞,李晓红,陈翔,孟国柱,刘杨.基于符号执行与模糊测试的混合测试方法.软件学报,2019,30(10):3071−3089. http://www.jos.org.cn/1000-9825/5789.htm

英文引用格式: Xie XF, Li XH, Chen X, Meng GZ, Liu Y. Hybrid testing based on symbolic execution and fuzzing. Ruan Jian Xue Bao/Journal of Software, 2019,30(10):3071−3089 (in Chinese). http://www.jos.org.cn/1000-9825/5789.htm

Hybrid Testing Based on Symbolic Execution and Fuzzing

XIE Xiao-Fei[1], LI Xiao-Hong[1], CHEN Xiang[2], MENG Guo-Zhu[3], LIU Yang[4]

[1](Tianjin Key Laboratory of Advanced Networking (Tianjin University), Tianjin 300050, China)

[2](School of Computer Science and Technology, Nantong University, Nantong 226019, China)

[3](State Key Laboratory of Information Security (Institute of Information Engineering, Chinese Academy of Sciences), Beijing 100093, China)

*基金项目: 国家自然科学基金(61572349, 61272106)

Foundation item: National Natural Science Foundation of China (61572349, 61272106)

本文由“面向 DevOps 的软件工程新技术”专题特约编辑荣国平、白晓颖、岳涛推荐.

收稿时间: 2018-08-29; 修改时间: 2018-10-31; 采用时间: 2018-12-14; jos 在线出版时间: 2019-04-29

CNKI 网络优先出版: 2019-04-30 09:19:14, http://kns.cnki.net/kcms/detail/11.2560.TP.20190430.0918.010.html

[4](School of Computer Science and Engineering, Nanyang Technological University 639798, Singapore)

Abstract: Software testing is a common way to guarantee software quality. How to achieve high coverage is a very important and challenging goal in testing. Fuzz testing and symbolic execution, as two mainstream testing techniques, have been widely studied and applied to academia and industry, both technologies have certain advantages and limitations. Fuzz testing can execute and cover deeper branches by randomly mutating test cases and dynamically executing programs. However, it is difficult to generate test cases that can cover complex conditional branches by random mutation. Symbolic execution can cover complex conditional branches with SMT solvers, but it is difficult to cover deeper branches due to state explosion during symbolic execution. Current works have shown that hybrid testing involving fuzzing and symbolic execution can archive better performance than fuzzing or symbolic execution. By analyzing the advantages and disadvantages in fuzzing and symbolic execution, this study proposes a branch coverage-based hybrid testing approach that combines the two methods with each other to achieve better test cases with high branch coverage. Specifically, fuzz testing (e.g., AFL) quickly generates a large number of test cases that can cover deeper branches, and symbolic execution (e.g., KLEE) performs a search based on the coverage of fuzz testing, and generating test cases for uncovered branches. To evaluate the effectiveness of Afleer, the study selects the standard benchmark LAVA-M and one real project oSIP as the evaluation object, and uses bug detection and coverage as the evaluation measures. The experimental results show that: 1) For bug discovery, Afleer found 755 bugs while AFL only found 1; 2) For coverage, Afleer achieved some improvement on benchmarks and real project. In the project oSIP, Afleer increases the branch coverage by 2.4 times and the path coverage by 6.1 times. In addition, Afleer found a new bug in oSIP.

Key words: software quality assurance; fuzz testing; symbolic execution; test case generation

1 引 言

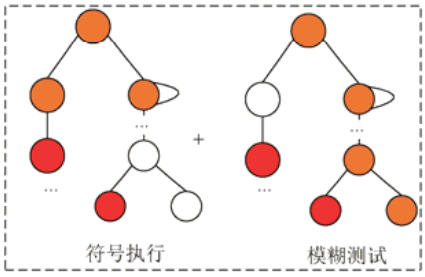
随着信息技术的发展,软件已经渗透到现代社会的方方面面,而由于开发不当引入的软件漏洞也日益增多.据统计,最近 5 年内软件漏洞数增加了 38%,而仅在 2016 年~2017 年间就增加了 14%[1].软件测试是检测软件漏洞的一种主要方法,当前工业界的主流方法还是通过手工设计测试用例来提高软件产品的质量,然而,手工生成测试用例通常效率较低、成本高昂并且容易出错[2].每年成千上亿的资金被投入到软件行业,其中软件测试一般需要占据 50%以上的成本预算[3].

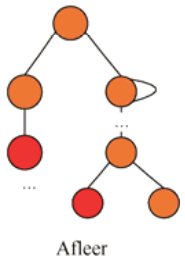
软件漏洞可以被看作是隐藏在某个条件下的错误语句,通过提升测试用例的代码覆盖率可以提高软件漏洞的检测概率.软件测试致力于为待测程序生成高代码覆盖率(例如语句覆盖、分支覆盖等)的测试用例以发现软件漏洞,当被测程序配套的测试用例覆盖率高且均执行通过时,则认为该程序在一定程度上具有高可靠性.

基于覆盖率引导的模糊测试(coverage-guided fuzz testing)[4−7](下文中所提到的模糊测试均指基于覆盖引导的模糊测试)与符号执行(symbolic execution)[8−12]是目前两种被广泛研究和使用的测试技术.给定初始测试用例,模糊测试(例如 AFL 工具)动态地执行目标程序,并基于覆盖率选择已有测试用例进行随机变异(mutation),从而生成新的测试用例.常见的变异操作包括字节翻转等.该过程将不断被重复,直到不能覆盖到更多的分支或语句为止.在动态执行测试用例时,如果检测到异常崩溃(crash),则认为发现了漏洞.由于模糊测试采取随机变异,因此难以生成可以覆盖到复杂条件的测试用例.例如,图 1(a)中,若通过随机变异的方法,则需要最多变换 2^{32} 次才能覆盖到条件 *input*=123456789.图 1(b)所示的模糊测试覆盖图表示模糊测试难以检测到漏洞 error1(即左侧红色节点);而对于处于较深位置的漏洞 error2(即右侧红色节点),模糊测试由于采用动态执行的方法,因此容易检测到漏洞 error2.

符号执行通过符号化执行程序来收集约束条件,并借助约束求解器[13−15]为每条路径生成测试用例.例如,符号执行可以很容易生成覆盖 *input*=123456789 分支并触发漏洞 error1 的测试用例.但其缺点是,由于需要符号化地执行程序,因此在遇到循环,尤其是循环次数依赖于输入的循环(例如图 1(a)中的 while 循环)时,循环的执行次数无法确定,此时,符号执行会陷入不停的循环展开中,从而影响到测试用例生成的效率.并且,当遍历到程序很深的位置时,约束条件也会变得更为复杂,使得约束求解器很难进行求解.例如,在图 1(b)中,由于循环或者其他函数的存在,造成符号执行很难覆盖到较深的位置,从而难以检测到漏洞 error2.

```
int input, i=0;
if(input==123456789){
    error1;
    ...
}else{
    while(i<input){
        ...
        i++;
    }
    func();
    if(input>1000)
        error2;
}
```





(a) 示例 (b) 符号执行与模糊测试覆盖 (c) Afleer 覆盖

Fig.1 Code coverage comparison

图 1 代码覆盖率比较

当前已有多个工作[2,17−21]提出了混合测试的方法,从而可以更高效地生成测试用例.例如,DART[16]、SAGE[17]、SYMFUZZ[18]以及 HCT[2]将随机测试与符号执行一起使用,但这些方法的随机测试中并未考虑程序的覆盖信息.而且,DART、SAGE 以及 SYMFUZZ 都属于单向结合.具体来说,DART 与 SAGE 利用随机测试提高符号执行,而 SYMFUZZ 利用符号执行辅助随机测试.Munch[19]将符号执行与模糊测试相结合,但其目的在于提高函数覆盖率,Driller[20]将混合执行与模糊测试相结合,其效果依赖于混合执行所选取的测试用例.不同于上述方法,本文通过分析模糊测试与符号执行的优缺点,以分支覆盖信息为引导,提出了一种符合执行与模糊测试多次交互与结合的混合测试方法.

模糊测试和符号执行各有优缺点,通过结合双方优点可以在一定程度上弥补不足,从而获得更高的覆盖率.直观上,程序的运行空间可以看作是一棵执行树,模糊测试通过随机变异的方法可以覆盖执行树上的多条路径,并且每条路径都可以执行到较深的位置;而由于一些复杂的条件分支,使得执行树中较浅的位置无法被突破.而符号执行可以突破执行树内较浅位置的复杂条件分支,但却难以深入到较深的位置.因此,本文的出发点是通过模糊测试探索程序执行树中的多个易覆盖的以及较深位置的分支,通过符号执行来系统地搜索执行树,以尽可能地覆盖模糊测试未覆盖到的分支.例如,通过结合模糊测试和符号执行,本文所提方法可以覆盖图 1(a)中的所有分支(如图 1(c)所示).

具体来说,本文提出了一种将模糊测试(基于 american fuzzy lop[4],简称 AFL)与符号执行(基于 KLEE[9])相结合的方法,并开发出相应的工具 Afleer.其中,AFL 进行标准的模糊测试,KLEE 依据 AFL 当前的覆盖信息调整搜索策略,当遇到未覆盖的分支时生成测试用例并发送给 AFL.AFL 通过读取新的测试用例进而探索该分支下更多的分支.在实证研究中,将 Afleer 工具应用到标准程序集 LAVA-M[21]以及 1 个实际项目 oSIP[22]上,以分析本方法的代码覆盖能力和漏洞检测能力.实证研究结果表明:与 AFL 相比,在标准程序集上,Afleer 可以检测到 755 个漏洞而 AFL 仅能发现 1 个.并且,在分支覆盖与路径覆盖上都有不同程度的提升.其中,LAVA-M 在最好情况下(项目 who 中),分支覆盖率提升了 35%,路径覆盖率提升了 492%.在实际项目 oSIP 中,分支覆盖率提升了 2.4 倍,路径覆盖率提升了 6.1 倍.除此之外,在 oSIP 中,Afleer 发现了一个新漏洞并已被提交.上述实证研究结果证明了 Afleer 的有效性.

本文的主要贡献可总结如下.

- 本文提出了一种将模糊测试与符号执行相结合的新方法,即通过模糊测试的覆盖率信息来引导符号执行的搜索,从而生成具有更高代码覆盖率的测试用例,并有助于找到更多的软件漏洞.
- 基于该方法,实现了工具 Afleer,并设计了相关实验,在标准程序集与实际项目上证明了本方法的有效性.
- 本文深入分析了影响代码覆盖率的因素,在此基础上讨论了符号执行和模糊测试的优缺点,为之后两种技术的研究以及结合提供了指导性建议.

本文第 2 节介绍研究背景.第 3 节介绍所提方法的框架和具体实现细节.第 4 节介绍本文的实证研究并对实验结果进行分析与讨论.第 5 节对相关工作进行总结,并与本文所提方法进行比较.最后进行总结并对未来工作加以展望.

2 研究背景和研究动机

2.1 模糊测试与AFL工具

基于覆盖率引导的模糊测试是一种高效的自动化测试技术,它通过随机变异的方式来生成大量测试用例,依靠覆盖信息来决定新生成的测试用例是否应该被保留.其中,American Fuzzy Lop(AFL)[4]工具是当前最先进和最流行的模糊测试工具之一,并已在大量实际项目中检测到了新的漏洞.

AFL 在编译过程中对程序进行插桩来记录程序的覆盖信息,并使用遗传算法来变异已有测试用例以生成最有可能探索到新的程序状态的测试用例.具体来说,AFL 在编译过程中为待测程序的每个基本块(basic block)插入一个编号,当一个条件分支(*prev*,*cur*)被覆盖一次时,其执行次数加 1(即 *trace_bits*[*h*(*prev*,*cur*)]++).其中,*prev* 和 *cur* 分别表示先前基本块与当前基本块的编号,*trace_bits* 是记录当前测试用例覆盖信息的数组,索引值表示分支的哈希值.这样,在执行一个测试用例后,*trace_bits* 记录了程序中每个分支的覆盖次数.同时,在执行多个测试用例后,AFL 也会记录全局覆盖信息数组 *virgin_bits*,它表示已执行的所有测试用例的覆盖信息.通过比较 *trace_bits* 与 *virgin_bits*,可以判断出当前测试用例是否可以覆盖到新状态,从而决定是否保留该测试用例.

AFL 通过以下变异操作来对一个已有测试用例进行变异,对于变异后的新测试用例,如果其可以覆盖到新状态,则保留该测试用例.

- 变异操作 1:针对位或者字节进行翻转.
- 变异操作 2:以单字节、双字节或者四字节为单位进行加减操作.
- 变异操作 3:删除、复制、重写或者插入新的字节块.
- 变异操作 4:两个测试用例随机选取位置并进行拼接.

2.2 符号执行与KLEE工具

动态执行时,一个具体测试用例对应于程序控制流图(control flow graph,简称 CFG)中的一条路径.因此,模糊测试通常只能执行到程序的部分行为(即 under-approximation).

与此相反,符号执行的输入不是一个具体测试用例,而是一个符号值,这样可以模拟执行程序的多条路径.符号执行在执行过程中为每条路径记录一个路径条件(path condition)ϕ,当遇到一个分支条件时,该分支条件ρ分叉(fork)为两条路径,其路径条件被更新为$\phi\wedge\rho$以及$\phi\wedge\neg\rho$.通过约束求解器[14,16]可以检查路径条件是否能够进行求解,如果无法求解,则该条路径是不可行的(infeasible);否则,沿该分支将继续往下执行.当遇到循环时,其多次迭代可以看作是多个条件分支,在循环的每次迭代中都将进行分叉.KLEE 工具[9]是基于 LLVM[23]实现的符号执行工具.被测程序被编译为 LLVM 位码,KLEE 直接解释指令集并将其转为相应的约束条件.

2.3 优缺点分析及研究动机

正如引言所述,模糊测试的优点在于动态执行,因此执行一个测试用例,其可以覆盖到较深的位置;而缺点是当遇到复杂的分支条件时,则很难通过随机变异的方法来生成满足该条件的测试用例.符号执行的优点是其可以突破复杂分支条件,由于维护了路径条件,因此通过约束求解器可以生成满足该条件的测试用例.而缺点是由于所有值都需要进行符号化,因此,当遇到依赖符号输入的循环或者递归时,其无法确定迭代次数,从而陷入到循环展开中.此外,随着路径深度的增加,路径条件也会越来越复杂,其对于约束求解器也会构成很大的挑战.

综上所述,如果将程序看作执行树,一个分支能否被覆盖,主要受到两个因素的影响:分支所在路径的深度以及分支所在路径的路径条件复杂度.该因素主要受到两种方法的本质特征所决定.从概率上来讲,符号执行属于白盒测试技术,其符号化地遍历程序的整个执行树.当分支所在路径的深度较深时,符号执行遍历到该分支的概率也会降低,并且约束求解的难度也会提高,符号执行此时会比较低效.而模糊测试属于灰盒测试技术,其本质是在程

序的输入集内进行随机枚举,此时,基于执行树上的路径,把输入集划分为不同的区域(即不同的输入会执行不同的路径).路径条件复杂可以理解为该路径可执行的输入集较小,因此模糊测试变异到的概率也会变小,此时,模糊测试会比较低效.

如图 2 所示,本文将求解难度分为 4 类:当该分支所在位置较浅并且路径条件复杂度较低时,符号执行和模糊测试都能快速生成相应的测试用例;当该路径条件复杂度较高但是该分支位置较浅时,符号执行更适合为其生成测试用例;当路径条件复杂度较低但是分支位置较深时,模糊测试更适合为其生成测试用例.对于分支位置较深并且路径复杂度较高的情况(如图 2 中的蓝色方块所示),此时两种方法都会遇到瓶颈.本文从这点出发,结合了这两种方法的优点,从而可以覆盖到更深、更难的分支条件.

Fig.2 Analysis of branch coverage on program execution tree

图 2 程序执行树中的分支覆盖分析

3 主要方法

3.1 方法框架

本文所提出的 Afleer 方法的框架如图 3 所示,该方法主要包括两个阶段:左边是编译阶段,右边是运行阶段.给定一个程序,首先我们将其编译为插桩后的 LLVM 位码以及插桩后的可执行文件,其中,LLVM 位码用于符号执行,而可执行文件用于模糊测试.需要注意的是,LLVM 位码和可执行文件必须一致,亦即在优化后,最终的控制流图相同并且插桩信息保持一致.在运行阶段,Afleer 分别运行模糊测试以及符号执行,其中模糊测试不断地生成测试用例并更新覆盖信息(例如,AFL 中的 virgin_bits),该信息反映了哪些分支已经被覆盖到或者尚未被覆盖到.而在另一端,符号执行系统性地搜索程序执行树,当发现某个分支(p,c)未被覆盖时,则生成相应的测试用例 T 并加入到测试用例集中.模糊测试监控新增测试用例集,当有新的测试用例 T 时,将其读入并加入到自己的测试用例队列中,此时,(p,c)已被覆盖.基于新的测试用例,模糊测试可以继续探索(p,c)后面的分支.通过不断的迭代,最终 Afleer 可以覆盖到更多的分支.下面,我们将详细介绍 Afleer 方法的实现细节.

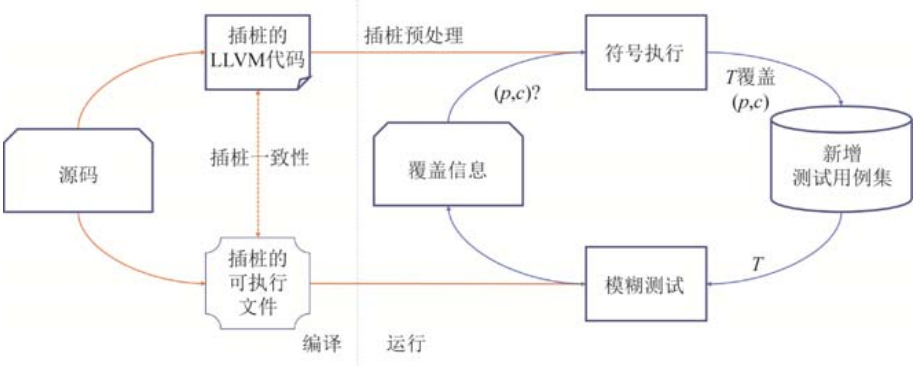


Fig.3 Overview of Afleer

图 3 Afleer 的主要框架

3.2 代码插桩处理

插桩一致性.符号执行与模糊测试结合的前提是其双方版本一致.具体来说,在源文件以及可执行文件这两

个版本中,需要保证控制流图以及每个基本块的插桩编号相同,使得双方在覆盖信息上保持一致.

例如,KLEE 的输入是 LLVM 位码,而 AFL 可以基于 LLVM 进行插桩,为了保证插桩信息一致,该项工作修改了编译工具 Whole Program LLVM[24](简写为 WLLVM).WLLVM 是一个可以为项目生成 LLVM 位码以及可执行文件的工具,但是,由于 WLLVM 生成 LLVM 位码与可执行代码是分两次编译的,从而导致同一个基本块在两次编译中随机生成两个不同的编号.最终,模糊测试与符号执行无法同步覆盖信息.修改该工具如下:首先插桩并编译为LLVM位码,之后再将位码编译为目标代码(object文件),最终将目标文件链接为可执行文件.修改后的 WLLVM 中仅插桩一次,从而保证了 LLVM 位码与可执行代码的插桩信息保持一致.

插桩预处理.经过模糊测试插桩后,每个基本块都加入了统计覆盖信息的代码.例如,在 AFL 插桩后会引入两个外部全局变量,并且 LLVM 位码中的每个基本块会多出 8 条计算指令.而符号执行不需要这些动态计算覆盖信息,因此,这些指令会极大地影响符号执行的执行效率.我们给出算法来获得插桩信息中的编号,并删除其他无用指令.

算法 1 给出了针对 AFL 插桩的预处理算法,其输入是 LLVM 的模块 *M*.在第 1 行删除 AFL 引入的外部全局变量(即 afl_prev_loc 与 afl_area_ptr),由于这两个全局变量都是外部变量,会造成符号执行运行时无法找到变量而报错,第 1 行将其删除.其次,在模块 *M* 中的每一个基本块中,删除 AFL 插桩的额外代码.对于每个基本块 *B*,首先解析插桩指令,并得到随机插桩编号 ID(第 3 行),然后删除插桩指令(第 4 行),并插入一条指令(第 5 行)以保存 ID 的值.针对 KLEE,我们选择加入一条 KLEE 未支持的指令 AtmoicRMW 来保存 ID.因此,在经过预处理后,在每个基本块中 8 条指令将减少为 1 条.

算法 1. 插桩预处理.

Input: *M*: LLVM 模块(Module).

1 删除 *M* 中模糊测试引入的全局变量;

2 **foreach** $B \in M$ **do**

3 ID=从插桩代码中解析基本块的编号;

4 删除所有插桩的代码;

5 插入一条指令以保存 ID;

6 **end**

3.3 Afleer模糊测试模块

算法 2 展示了 Afleer 中模糊测试模块.Afleer 主要扩展了 AFL 的算法,输入为已有测试用例 *Seeds*、待测程序 *P* 以及来自符号执行的新增测试用例集 *SE_Seeds*.输出是生成的所有测试用例,其中,包含可以触发漏洞的输入.AFL 依次遍历 *Seeds* 中的每一个测试用例 *seed*.首先对其进行变异,得到新的测试用例 *newseed* (第 2 行).注意,一个测试用例可以通过多种变异操作得到多个测试用例,不失一般性,这里假设其结果为 *newseed*.然后程序 *P* 执行新的测试用例 *newseed*(第 3 行),得到覆盖信息以及运行结果.如果运行结果发现异常,则找到了触发漏洞的测试用例(第 4 行).如果该测试用例使得新的分支被覆盖(第 7 行),则将其加入测试用例集合(第 8 行)并更新覆盖信息(第 9 行).在完成一个测试用例的处理后,将检查是否符号执行探索到新的未覆盖分支,如果发现(第 11 行)新的测试用例,则将其加入 *Seeds* 的最前列,并将 *SE_Seeds* 清空.这样,在符号执行发现新的测试用例后,模糊测试将对其进行变异,以探索该分支下更深位置的分支.

算法 2. Afleer 模糊测试模块.

Input: *Seeds*:已有测试用例,*P*:待测程序,*SE Seeds*:新增测试用例;

Output:测试用例.

1 foreach $seed \in Seeds$ do

2 *newseed*=*mutate*(*seed*);

3 (*trace_bits*,*result*)=*Execute*(*P*,*newseed*);

4 **if** *result*==*crash* **then**

```
5          发现漏洞;
6        end
7        if hasNewCov(trace_bits,virgin_bits) then
8          Seeds=Seeds∪{newseed};
9          update(virgin_bits,trace_bits);
10     end
11     if SE_Seeds≠∅ then
12       Seeds=Seeds∪SE_Seeds;
13       SE_Seeds=∅;
14     end
15   end
```

3.4 Afleer符号执行模块

算法 3 展示了 Afleer 中符号执行模块的主要思想.Afleer 主要扩展了 KLEE 的算法,算法输入是 LLVM 的模块 *M*,覆盖信息 *virgin_bits*,输出为新生成的测试用例 *SE_Seeds*.首先创建一个初始状态(第 1 行),并将其加入到状态集合 *States* 中(第 2 行).此时,从初始状态开始进行遍历,在集合 *States* 中选择一个状态(第 4 行),这里,搜索方法可以采取不同的策略(例如,深度优先策略或者广度优先策略等).接下来得到当前状态的指令 inst,根据指令的不同类型对其进行解释执行(第 6 行~第 30 行).例如,当指令为分支条件时(第 7 行),当前状态分叉为两个状态并更新路径条件 *state.pc*.如果当前分支的路径条件可满足(第 8 行与第 12 行),则将生成的新状态加入状态集合.当遇到指令 *AtmoicRMW*(来自算法 1)时解析得到基本块编号(第 18 行),基于该编号可以检查当前分支是否被覆盖.第 18 行~第 20 行计算当前分支的哈希值(与 AFL 中的插桩计算等价).如果覆盖到新的分支(第 21 行),则生成测试用例(第 22 行),并将其加入到测试用例集合 *SE_Seeds*(第 23 行).当该路径生成新的分支后,我们将该状态优先级降低(第 24 行),因为在符号执行突破新的分支后,我们期待模糊测试基于该测试用例进行更深位置的探索.因此,为了提高效率,此时,符号执行将优先探索其他分支,从而避免与模糊测试进行同位置的探索.第 4 行的搜索算法将同时考虑各个状态的优先级.对于其他类型的指令,符号执行将根据相应的语义进行解释与执行(第 28 行).

算法 3. Afleer 符号执行模块.

Input:*M*:LLVM 模块,*virgin bits*:覆盖信息;

Output:*SE_Seeds*:新增测试用例.

```
1   start=createNewState(M);
2   States={start};
3   while States≠∅ do
4     state=select(States);
5     inst=state.ins;
6     switch inst do
7       case BranchInst
8         if state.pc∧pathCond(inst,true) then
9                                   state1=fork(state,inst,true);
10                                  States=State∪{state1};
11        end
12        if state.pc∧pathCond(inst,false) then
13                                  state2=fork(state,inst,false);
14                                  States=State∪{state2};
```

```
15              end
16            endsw
17            case AtmoicRMWInst
18              cur=getID(inst);
19              hash=cur⊕state.pre;
20              state.prev=cur>>1;
21              if hasNew(virgin bits,hash) then
22                seed=generateTestCase(state);
23                SE_Seedss=SE_Seed∪{seed};
24                state.priority−−;
25              end
26            endsw
27            otherwise
28                handle(state,inst);
29            endsw
30        endsw
31    end
```

4 实证研究

本节介绍本文的实证研究,首先提出研究问题,接着介绍实验设计的具体方法,然后总结并分析实验结果,最后对 Afleer 的优缺点进行讨论,并对符号执行及模糊测试相结合的未来研究工作进行分析.

4.1 实验设计

基准方法的选择.Afleer 方法的目的是通过结合符号执行帮助模糊测试能够覆盖到更多的分支,从而发现更多的漏洞.在实验中我们选择 AFL(本文提供的方法是通用的,可以很容易地扩展到其他 AFL 优化的方法上,例如 AFLfast[25]、Steelix[26]以及 Skyfire[27]等)作为基准方法,而未选择 KLEE 作为基准方法,其主要原因总结如下.

(1) 在 Afleer 中,主要使用 KLEE 增强 AFL 寻找未覆盖边的能力.而不是用 AFL 去辅助 KLEE 生成测试用例,因此,本文将侧重点放在对模糊测试方法的改进上;

(2) 为了提高效率,Afleer 中 KLEE 未对 AFL 已覆盖的边生成测试用例,因此,Afleer 生成的测试用例更适合与 AFL 生成的测试用例进行对比;

(3) 即使将 Afleer 中的 KLEE 按标准方式加以运行,其生成的测试用例一定是 Afleer 生成的测试用例的子集,因此,将两者进行比较没有意义.

除此之外,将符号执行与动态测试结合的技术还有 Driller[20]以及配套工具 HCT[2]等,然而,由于 HCT 工具没有开源,Driller 仅能够运行在 DECREE 的二进制程序上,因此,在我们设计的实验中并没有与其直接进行比较,在相关工作中,我们将从方法上进行比较.

研究问题.为了验证 Afleer 的有效性,本文尝试回答如下研究问题.

- RQ1. Afleer 在漏洞检测上的效果如何?与 AFL 相比,其是否可以在相同的指定时间内检测到更多的漏洞?或者对于同样的漏洞,其是否可以更快地检测到?
- RQ2. Afleer 在测试覆盖能力上的效果如何?是否借助符号执行可以有效辅助 Afleer 覆盖到更多的分支?

实验数据集.为了回答上述研究问题,本文选择了标准程序集 LAVA-M[22]以及一个实际项目 oSIP-4.0.0[22]作为实验对象.其中,LAVA-M 中的每个程序均包含了大量的已知漏洞,因此可以用来公平地比较各种方法的漏洞检测能力,实际项目 oSIP 则可用来验证 Afleer 的可扩展性以及有效性.标准程序集 LAVA-M 和项目 oSIP 的

具体信息总结如下.

• LVAM-M 包含 4 个带有漏洞的 Linux 系统工具程序:base64、md5sum、uniq 以及 who.数据集的搜集人员按照一定标准在每个程序中插入了大量的难以检测到的漏洞.其中,每个漏洞都有一个唯一编号,当该漏洞被触发时则打印对应漏洞编号.我们使用 LAVA-M 来比较 Afleer 以及 AFL 的漏洞检测能力,由于这些程序都来自实际项目,因此,它们同时也被用来比较不同方法的代码覆盖能力.注意,在运行 base64 与 md5sum 时,我们分别使用了参数-d 与-c.

• oSIP 实现了 SIP 协议[28],其为用户提供了初始化以及控制 SIP 会话的接口实现.由于其包含了 SIP 协议的语法解析器,因此,AFL 难以生成可以通过复杂语法检查的测试用例,我们将分析 Afleer 在该项目中的代码覆盖能力.

评测指标.针对漏洞检测能力以及代码覆盖能力,本文通过以下指标来评测不同方法.

• 漏洞检测数(#bugs):在指定时间内找到的漏洞总数.

• 边覆盖数(#edges):AFL 中的边指的是分支,边覆盖数表示在指定时间内的分支覆盖数.

• 路径覆盖数(#path):路径数表示 AFL 在指定时间内生成的测试用例数.注意,AFL 采用了循环桶(loop bucket)的思想,部分路径的测试用例不会被保留.

运行实验的机器配置信息总结如下:CPU Intel Xeon E5-1650 v3,内存 16GB,操作系统 64 位 Ubuntu 16.04LTS,并且运行时间固定为 5 小时.

4.2 实验结果分析

4.2.1 LAVA-M 程序集上的漏洞检测能力比较(RQ1)

表 1 总结了在 LAVA-M 程序集上 Afleer 与 AFL 的比较结果.其中,前两行表示 LAVA-M 数据集的相关信息,第 1 行表示项目名字,第 2 行总结了每个项目中插入的漏洞数(即已知的漏洞数).随后两行总结了 Afleer 检测出的漏洞数,第 4 行(#afleerbugs)总结了 Afleer 在每个项目中检测到的漏洞数以及占所有已知漏洞数的比例.如果一些漏洞在 KLEE 中被直接发现并发送给 AFL,则将其总结在第 3 行(#kleebugs).最后一行(#aflbugs)总结了在执行标准 AFL 时所检测到的漏洞数.

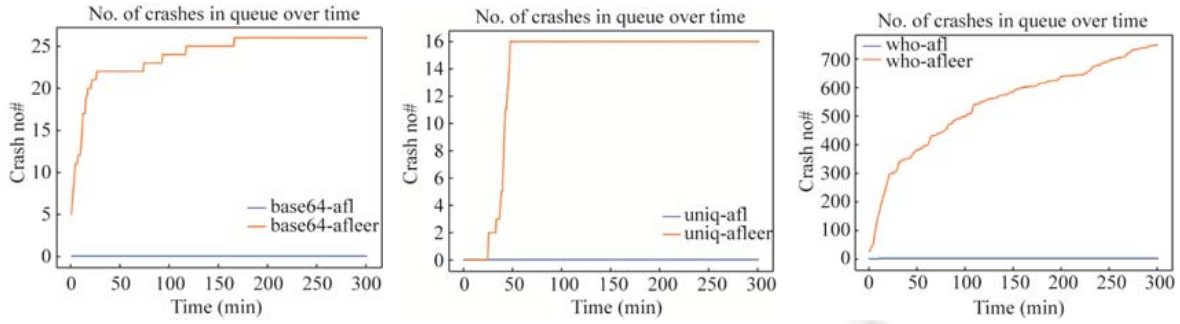
Table 1 Results of bugs found on benchmark LAVA-M

表 1 LAVA-M 程序集上的漏洞检测结果比较

	LAVA-M	base64	md5sum	uniq	who	Total
	Total Bugs	44	57	28	2 136	2 265
Afleer	#kleebugs	6	0	8	9	23
	#afleerbugs	11(25%)	0	8(28.57%)	736(34%)	755(33%)
AFL	#aflbugs	0	0	0	1	1

基于表 1 可以看出,Afleer 总共检测出 755(33%)个漏洞,其中,在 base64、md5sum、uniq 以及 who 上分别检测出 11(25%)、0、8(28.57%)以及 736(4%)个漏洞.而标准的 AFL 基本没有检测出漏洞,其中,仅在 who 上检测出 1 个漏洞.另外,值得注意的是,KLEE 在依据覆盖信息进行搜索遍历时也可以直接检测出漏洞,例如 KLEE 分别为 4 个项目检测到了 6、0、8、9 个漏洞.除去 23(1%)个 KLEE 发现的漏洞,Afleer 中的 AFL 仍然发现了 722(32%)个漏洞,其漏洞检测能力要显著优于 AFL.

图 4 给出了随着时间的推移,Afleer 与 AFL 在 3 个项目(因为 md5sum 上均未检测到漏洞,因此未给出趋势图)中漏洞检测的趋势图,其中,横坐标为方法运行的时间,纵坐标为检测到的漏洞数.注意,在 AFL 执行过程中对一个漏洞可能会产生多个测试用例,因此,图 4 所示中纵坐标显示的漏洞数(crashes no)要多于表 1 中检测到的漏洞数.从图 4 可以看到,3 个项目在开始时,检测到的漏洞数都会快速增长,因为开始时,KLEE 会快速突破某些分支并找到一定的漏洞,使得漏洞数快速增长.之后,一些处于较深位置的漏洞将无法被 KLEE 探索到,而 AFL 在此基础上将进一步找到一些更深位置的漏洞.而在 uniq 上,当 KLEE 检测到所有漏洞后,Afleer 并没有检测到更新的漏洞.与 AFL 相比,其在没有 KLEE 提供漏洞的情况下,几乎检测不到任何漏洞.



(a) base64 漏洞检测趋势 (b) uniq 漏洞检测趋势 (c) who 漏洞检测趋势

Fig.4 Increasing of bugs finding on LAVA-M

图 4 LAVA-M 漏洞检测趋势图

通过查看 LAVA-M 中漏洞插入的代码,我们分析了 AFL 难以检测到漏洞而 Afleer 可以检测到漏洞的原因.LAVA-M 中插入的漏洞的触发条件主要是针对魔术字节(magic byte)的比较,例如,0*x*6*c*617564==*lava_get*(253).因此,普通 AFL 很难突破这些分支,从而无法找到漏洞.而在 Afleer 中,KLEE 可以为其生成相应的测试用例 0*x*6*c*617564.并且,这些漏洞之间是有一定的关系的,例如,AFL 读取 KLEE 发送的测试用例 0*x*6*c*617564 后,在其基础上很容易变异出一个可以覆盖条件 0*x*6*c*617562==*lava_get*(255)的测试用例(其只有一个字节是不同的),并且不受该条件所处深度的影响.例如,在 uniq 中,KLEE 提供了 8 个触发漏洞的测试用例,而 AFL 基于该测试用例并没有发现新的漏洞.其主要原因是,触发其他漏洞的测试用例与 KLEE 已生成的测试用例在字节上相差很大,因此,基于变异策略的 AFL 无法更快地生成新的漏洞.而如果其处于深位置,则 KLEE 也将很难搜索到该位置.该结果表明了 Afleer 的有效性.

同时,通过分析现有测试用例的覆盖率,我们进一步获得了其他漏洞未被检测到的原因,主要原因是由于符号执行与模糊测试本身的弱点导致某些分支难以被覆盖.具体地,我们从符号执行与模糊测试两方面来进行分析.

(1) 从符号执行方面进行分析,其主要由两个原因导致.首先,符号执行面临状态爆炸问题,由于一些分支在有限时间内无法被遍历到,因此无法生成相应的测试用例.其次,某些分支条件依赖于其相应的前置条件,即使这些分支可以被覆盖到,但在某些前置条件下,分支条件永远无法得到满足,因此也会导致无法生成相应的测试用例.例如,KLEE 与 AFL 在 md5sum 程序上都没有检测到漏洞,其原因主要是无法突破开始时的哈希函数,因此难以覆盖到含有漏洞的分支.

(2) 在模糊测试方面,由于 LAVA 数据集中都是较难的数字比较条件,因此,模糊测试通过随机变异很难生成相应的测试用例,从而无法发现漏洞.

在 Afleer 总共发现的 755 个漏洞中,通过符号执行 Aflee_KLEE 找到了 23 个,而通过模糊测试 Afleer_Afl 找到了 722 个.因此,与单独使用符号执行工具 KLEE 相比,Afleer 多发现了 722(超过 31 倍)个漏洞.

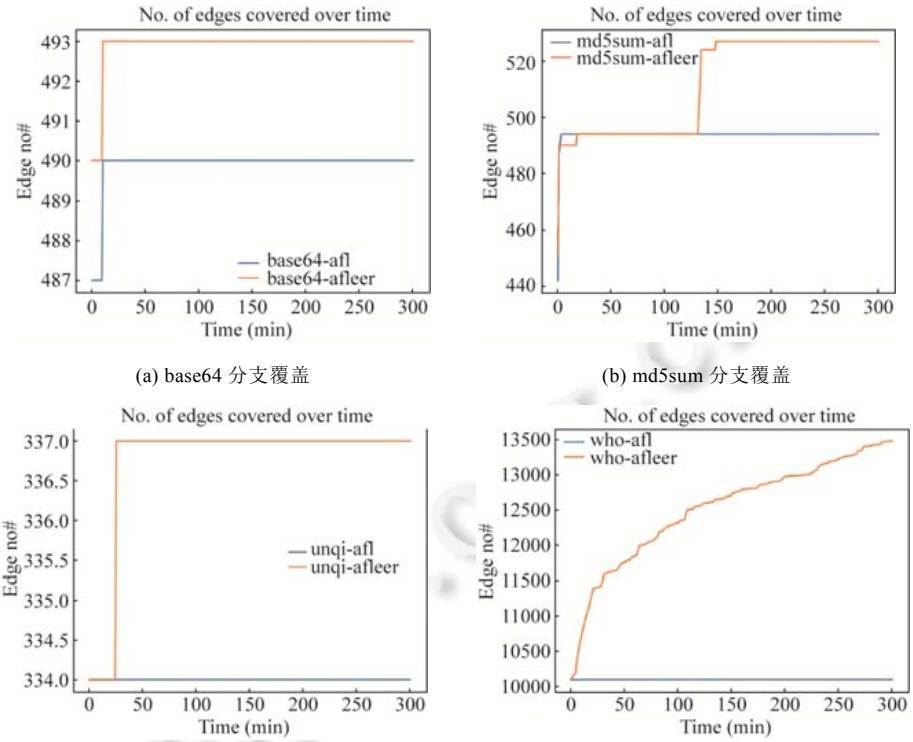
基于上述分析我们认为,通过结合 KLEE 与 AFL,Afleer 与单独使用 KLEE 及 AFL 相比可以显著提高漏洞检测的能力(即基于 LAVA-M 评测集可以比 AFL 多发现 754 个漏洞,比 KLEE 多发现 722 个漏洞).

4.2.2 LAVA-M 程序集上的代码覆盖能力比较(RQ2)

本节我们将依次比较 Afleer 与 AFL 在 LAVA-M 程序集上的分支覆盖能力以及路径覆盖能力.

图 5 给出了 4 个程序中不同方法的分支覆盖趋势图,其中,横坐标为时间,纵坐标为覆盖的分支数.从中可以看到,base64、md5sum 以及 uniq 上,Afleer 无明显提高,通过观察代码,我们分析了其主要原因在于:(1) 代码量较小,例如 base64、uniq 以及 md5sum 分别包含 350 行、700 行以及 1 000 行左右,此时 Afleer 与 AFL 在很短的时间内就可以收敛;(2) 一些较难的分支无法被 KLEE 突破,因此 Afleer 此时会与 AFL 的分支覆盖能力相似,例如 md5sum 包含哈希计算,导致后面的分支都无法突破,并且也没有检测到任何漏洞(见表 1).而对于 who,其包

含 4 600 多行代码,此时,Afleer 明显可以突破更多的分支条件,并可以额外覆盖 3 500(35%)个新分支.



(a) base64 分支覆盖 (b) md5sum 分支覆盖

(c) uniq 分支覆盖 (d) who 分支覆盖

Fig.5 Increasing of branch coverage on LAVA-M

图 5 LAVA-M 分支覆盖趋势图

表 2 显示了 Afleer 与 AFL 在 4 个项目中分别生成的测试用例数.表格左边为 Afleer 的结果,右边为 AFL 的结果.同时我们也统计了 Afleer 中 AFL 与 KLEE 分别贡献的测试用例总数(即 Afleer_AFL 与 Afleer_KLEE),可以发现,AFL 生成的测试用例要显著多于 KLEE 生成的测试用例(KLEE 在 4 个项目中分别贡献 12%、3%、21%以及 3%),即分别生成了 24、10、20 以及 26 个测试用例.需要注意的是,在 Afleer 刚开始运行时,KLEE 和 AFL 同时运行,此时,一些容易覆盖的分支还未被 AFL 覆盖,但是优先被 KLEE 搜索到从而生成测试用例.因此,列 Afleer_KLEE 包含的测试用例不全是 AFL 难以覆盖到的路径.通过与 AFL 的结果进行比较(最后一列)可以发现,Afleer 在 base64 上提升了 42(28%)、在 md5sum 上提升了 94(35%)、在 uniq 上提升了 4(4%)、在 who 上提升了 699(大约 5 倍),上述结果表明,Afleer 在路径覆盖能力上有明显的提升.此外,图 6 中给出了路径覆盖的趋势图,横坐标表示时间,纵坐标表示路径覆盖数.

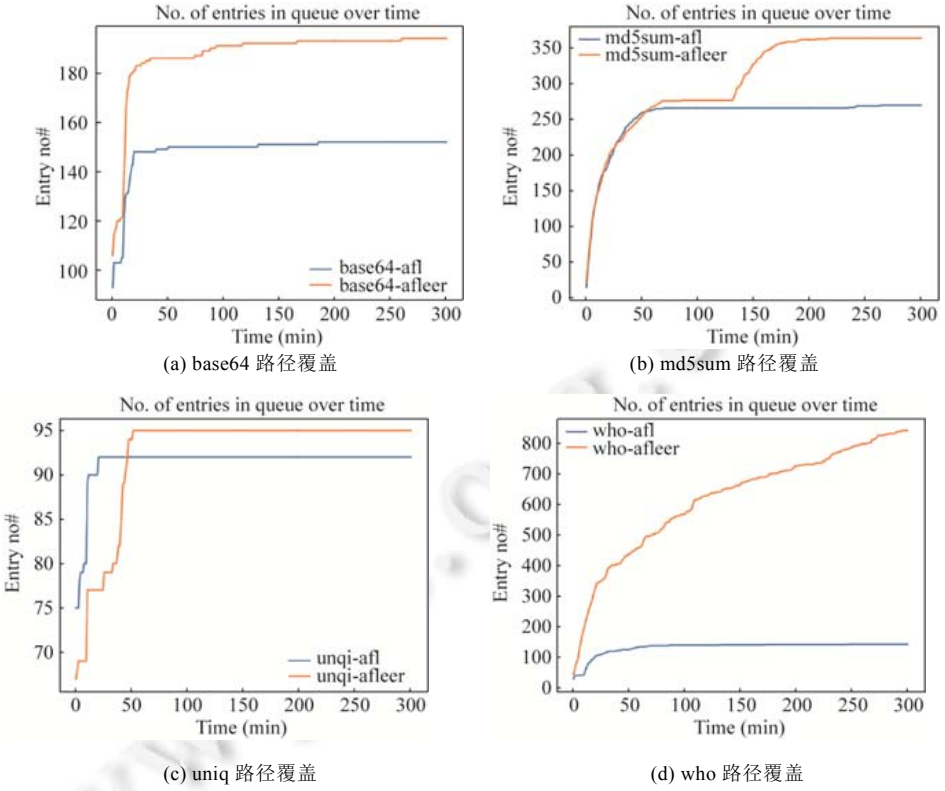
Table 2 Comparison of the total number of test cases generated by Afleer and AFL

表 2 Afleer 与 AFL 生成的测试用例总数比较结果

	Afleer_AFL	Afleer_KLEE	Afleer Total	AFL
base64	171(88%)	24(12%)	195(+28%)	153
md5sum	353(97%)	10(3%)	363(+35%)	269
uniq	75(79%)	20(21%)	95(+4%)	91
who	815(97%)	26(3%)	841(+492%)	142

基于上述及表 2 和图 6 的分析,Afleer 在分支覆盖能力与路径覆盖能力上都有不同程度的提升.在某些项目上,即使分支覆盖能力没有明显提升,但路径覆盖能力仍有明显提升.因此,Afleer 可以大幅度提高程序的覆盖能力,尤其是在大规模程序上.与 KLEE 相比,可以发现,Afleer 中 AFL 贡献了更多的测试用例(4 倍~32 倍).该结果

符合我们的预期,即 AFL 可以快速生成大量较容易的测试用例,而 KLEE 由于可扩展性差,因此主要用于生成较难的测试用例.



(a) base64 路径覆盖 (b) md5sum 路径覆盖

(c) uniq 路径覆盖 (d) who 路径覆盖

Fig.6 Increasing of branch coverage and bug finding on oSIP

图 6 LAVA-M 路径覆盖趋势图

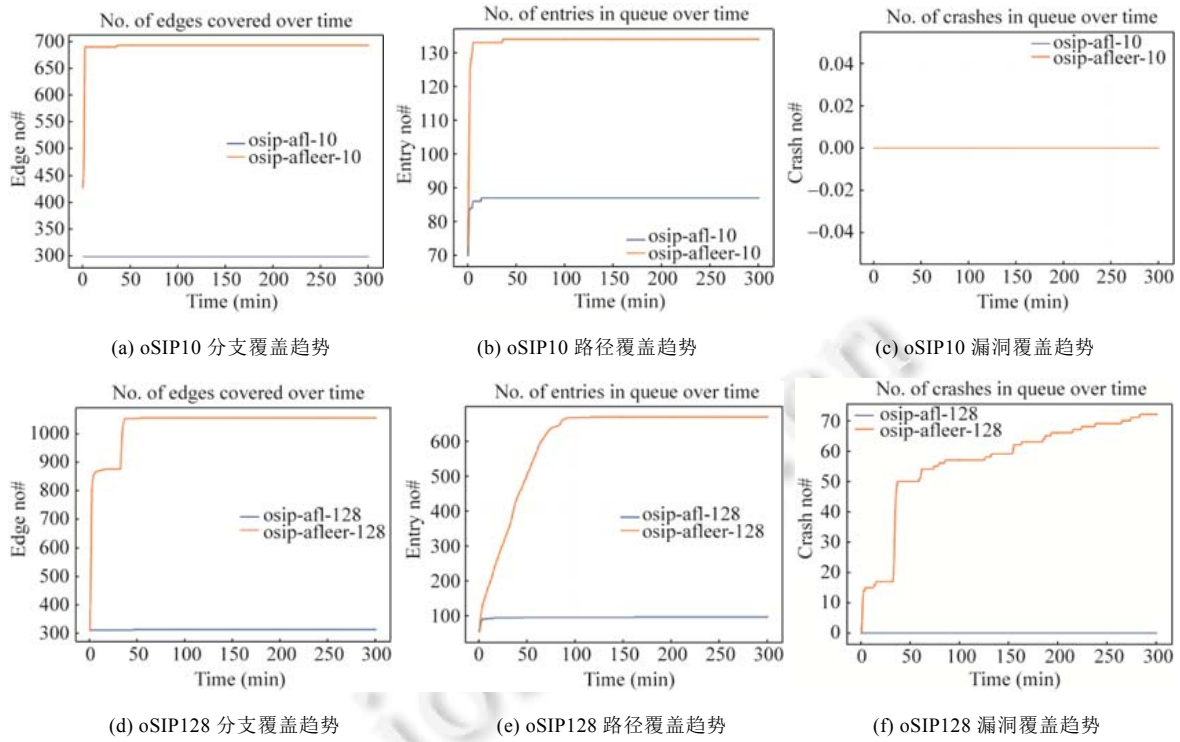
4.2.3 基于实际项目的比较(RQ1 和 RQ2)

我们将 Afleer 应用在实际项目 oSIP 中,并发现了一个新的漏洞,该漏洞目前已被提交给 GNU 处理[29].在实验中,我们对输入规模进行了限制(即将输入规模限定为 10 字节或者 128 字节),并通过不同的输入规模来分析其对覆盖能力以及漏洞检测能力的影响.

图 7 显示了 oSIP 在输入规模为 10 字节与 128 字节时的分支覆盖趋势图、路径覆盖趋势图以及漏洞检测趋势图.与 AFL 相比,Afleer 在输入规模为 10 字节时分支覆盖能力提升了 1.3 倍,路径覆盖能力提升了 54%;当输入规模为 128 字节时,分支覆盖能力提升了 2.4 倍,路径覆盖能力提升了 6.1 倍.在漏洞检测能力上,当输入为 10 字节时,由于输入规模太小,因此都未检测出漏洞.而当输入规模变为 128 字节时,此时 Afleer 可以检测到漏洞,而 AFL 仍然未检测到.注意,图 7(f)所示的 70 多个可以触发漏洞的测试用例都属于触发同一个漏洞.同时,通过比较不同输入规模的结果可以发现,随着输入规模的扩大,其可以生成的测试用例数也在增加,因此,可以覆盖的分支、路径以及可检测出的漏洞都会提高.

同时,我们也分析了 Afleer 中 AFL 与 KLEE 的各自贡献率(记为 KLEE/AFL).其中,在测试用例生成方面,当输入长度为 10 字节时为 60/78,当输入长度为 128 字节时为 2/680.可以看到,在输入长度较小时,KLEE 会快速生成更多的测试用例,因为当输入长度较小时,KLEE 的约束复杂度较低,从而使得求解速度快速提高.在漏洞发现方面,KLEE 没有发现该漏洞.

总的来说,Afleer 在实际项目中比 AFL 和 KLEE 在代码覆盖能力及漏洞检测能力上都具有显著的提升,因此证明了 Afleer 的可扩展性和有效性.



(a) oSIP10 分支覆盖趋势 (b) oSIP10 路径覆盖趋势 (c) oSIP10 漏洞覆盖趋势

(d) oSIP128 分支覆盖趋势 (e) oSIP128 路径覆盖趋势 (f) oSIP128 漏洞覆盖趋势

Fig.7 Increasing of branch coverage and bug finding on oSIP

图 7 oSIP 分支覆盖以及漏洞检测趋势图

4.3 进一步讨论

本节将结合上述实验结果进一步讨论影响 Afleer 有效性的因素,在 Afleer 中,KLEE 的方式是搜索执行树而 AFL 的方式是变异测试用例,因此,Afleer 的有效性受制于 KLEE 与 AFL 本身在目标程序上的搜索效率与变异效率.

从 KLEE 角度出发,其在搜索时会存在状态爆炸问题.当目标分支在目标程序的较浅位置时,KLEE 比较容易搜索到该分支,从而生成相应的测试用例.例如,图 4 中,KLEE 在开始时会很快找到较浅位置的漏洞.相反,即使已知某个分支难以覆盖到,但是,如果其处于较深位置,则 KLEE 仍将难以找到一条可以满足该分支条件的路径.例如,对于其他未检测到的漏洞,KLEE 在有限时间内也难以检测到.

从 AFL 的角度出发,测试用例在内容上的关联程度对其影响较大.假设 KLEE 已为某个目标分支生成测试用例并提交给 AFL,如果剩余未覆盖分支的测试用例与当前测试用例关联程度较强(即相似度较高),则 AFL 通过变异可以很快覆盖到其他分支,并且不受深度的影响.例如,在上一节所述内容中,KLEE 生成了测试用例 0x6c617564 后,对于另一个处于较深位置的未覆盖分支 0x6c617562==*lava_get*(255)则很难搜索到.但是,AFL 通过变异可以很快地为其生成测试用例,在项目 who 中,KLEE 为其找到了 9 个漏洞,在此基础上,AFL 仍多检测到 727 个漏洞(见表 1).相反,如果其他分支的测试用例与当前测试用例完全不同,此时,AFL 难以通过变异当前测试用例来覆盖到新分支.例如,在 uniq 中,Afleer 找到的漏洞都来自 KLEE.

总的来说,当符号执行发现某个未覆盖的分支,而模糊测试基于该分支可以找到更多的未覆盖分支时,双方结合的方式将会更加高效.当 KLEE 无法搜索到未覆盖的分支,或者 AFL 无法基于 KLEE 生成的测试用例找到更多的测试用例时,双方结合的方式将会低效.

此外,在 Afleer 的设计中,KLEE 的不同之处在于:(1) 对于 AFL 已经覆盖到的边,KLEE 仍然可以到达该边但是不会生成测试用例;(2) 对于 AFL 未覆盖的边,则生成测试用例并发送给 AFL.KLEE 在 Afleer 中执行的先

后顺序仅影响 AFL 与 KLEE 中都容易生成的测试用例,而不影响整体效率.具体来讲,如果 KLEE 先执行,则 KLEE 将会优先生成多个 AFL 未遍历的测试用例(假设测试用例集合为 $P+Q$,P 表示对 AFL 容易覆盖到的测试用例集合,Q 表示 AFL 较难覆盖到的测试用例集合).由于此时 AFL 还未执行,P 与 Q 执行的路径并未被覆盖,因此,将由 KLEE 优先生成.而如果 AFL 先执行,则 P 将由 AFL 快速生成,此时,KLEE 在遍历时如果遇到这些边则不再需要生成 P,仅只生成 AFL 难以生成的测试用例 Q.总的来说,KLEE 与 AFL 的先后执行顺序仅影响 P 由哪部分优先生成;而对于 AFL 难以生成的测试用例 Q,理论上无论哪部分先执行,其基本上都将被 KLEE 生成.

为了解决 Afleer 现有的不足,未来可以从以下几个方面进行改进.在符号执行方面,可以采用加入静态分析(例如文献[31])来引导符号执行更高效的搜索.循环与递归是影响符号执行状态爆炸的主要因素,通过加入循环(递归)分析与总结可以缓解该问题.利用程序切片分析发现不相关的状态,在符号执行中略过这些状态也是另外一种方法.对于位置较深使得符号执行难以到达的分支,可以通过混合执行来快速达到,或者进行目标驱动的数据流分析来寻找到达该分支的路径.在模糊测试方面,可以加入现有的技术,如污点分析(Steelix[26], Angora[30])、定向模糊测试(AFLgo[31],Hawkeye[32])以及其他策略(Skyfire[25],AFLFast[27])来提高模糊测试的能力,从而提高混合测试的整体性能.

5 相关工作

本节将分别从模糊测试、符号执行以及混合测试等角度对已有研究工作进行总结.

5.1 模糊测试

目前存在很多基于遗传算法并基于覆盖率引导的模糊测试工具,例如:AFL[4]、libFuzzer[7]以及 honggfuzz[6].污点分析(taint analysis)用于分析程序输入与程序逻辑之间的关系,目前,很多基于污点分析的方法已被用来提高模糊测试的能力.BuzzFuzz[33]以及 FairFuzz[35]使用动态污点分析的方法来定位输入中感兴趣的字节,随后重点对这些字节进行变异,从而提高效率.Taintscope[36]通过污点分析来识别验证校验和(checksum)数字的分支,并通过改变控制流来绕过这些难以通过的分支.与 BuzzFuzz 和 FairFuzz 类似,在绕过校验和的检验后,其识别出输入中与危险操作相关的字节并进行重点变异.Dowser[38]对缓冲区溢出漏洞进行检查,通过污点分析检查输入中影响到数组索引变化的字节.污点分析虽然可以识别出重要字节,但同时也会付出很高的性能代价,并减缓模糊测试的执行速度.VUzzer[37]通过污点分析识别出魔术字节(magic bytes)在输入中的位置,从而对这些字节进行变异,其缺点是魔术字节在输入中的位置是固定的.Steelix[26]在固定位置通过细粒度的插装方式来拆分程序中魔术字节的比较状态.例如,一个 32 位的整数比较可以记录为 4 个 8 位整型的比较,从而加快魔术字节的匹配速度,Steelix 适用于魔术字节比较中变量直接来自输入的场景.Angora[30]提供了一种不通过符号执行来求解路径约束的方法,通过污点分析,借助梯度下降来搜索可以满足约束条件的测试用例.基于上述分析可以看出,基于污点分析的模糊测试与本文所提出的 Afleer 的目标相似,都是针对难以覆盖的分支来生成测试用例.其优势在于,当分支条件中的变量直接来自输入的某些字节时,即使该分支处于较深的位置都可以快速生成测试用例.而如果分支条件中的变量依赖于输入变量的所有字节或者依赖于部分字节的复杂计算,则基于污点分析的方法将会失效,而 Afleer 通过符号执行则可以有效解决该问题.

除此之外,研究人员还提出了一些用来提高模糊测试效率的方法.例如,Rebert[38]等人以及 Woo 等人[39]在模糊测试中,借助测试用例选择以及调度策略来提升漏洞检测能力.AFLFast[25]发现模糊测试在大部分时候都在尝试高概率路径,而很难触发到新的低概率路径,因此其通过给覆盖低概率路径的测试用例赋予更多的时间进行变异来提升效率.AFLgo[31]是一种针对特定目标点的模糊测试技术,其通过为测试用例排序来进行高效率的有针对性的测试.Skyfire[27]从现有测试用例中学习出基于概率的上下文敏感语法(probabilistic context sensitive grammar),以指导生成高质量的结构型测试用例.CollAFL[40]提出了一种可以解决模糊测试中路径碰撞(path collisions)问题的方法,路径碰撞问题主要由插装过程中的基本块编号碰撞所导致,该问题会对新漏洞检测能力产生影响.不难看出,这些方法与 Afleer 存在正交关系,在下一步工作中,我们可以进一步融合这些方法来提升 Afleer 中模糊测试的能力.

5.2 符号执行

符号执行已得到广泛研究,并在工业界与学术界中涌现出了很多优秀工具,例如:KLEE[9]、JPF[10]、CUTE[11]、S2E[41]、PEX[12]以及 Angr[20]等.符号执行时最大的问题是状态爆炸或者路径爆炸,目前,存在很多针对该问题的研究工作.

使用更好的搜索策略是解决状态爆炸问题的一种方法,例如:随机路径探索[9]、深度优先探索[11]、广度优先探索[42]、基于覆盖信息的探索[43].谢涛等人[44]提出了一种基于适应度函数的路径探索方法.陈振邦等人[45]提出了一种基于待验证性质引导的搜索策略.李宣东等人[46]提出了一种优先搜索最少被遍历过的路径从而提高整个覆盖率的方法.王海军等人[47]提出了一种基于路径依赖分析以删除冗余路径的方法,从而缓解了状态爆炸问题.实证研究结果表明,这些策略可以在一定程度上缓解路径爆炸的问题.

通过状态约减[48]也是解决路径爆炸问题的一种方法,如果一条路径不可达或者约束条件与之前遍历的路径的约束条件相同,则可以不必执行该路径.甘水滔等人[49]提出了一种根据程序功能进行切片分析的方法,进而引导符号执行并约减无关状态.Trabish 等人[50]提出的方法,可以在符号执行过程中实时地过滤无关代码从而提高效率.此外,合并路径[51,52]也是一种提高符号执行效率的方法.例如,函数总结[53]以及循环总结技术[54−56]可以被用来合并路径,从而避免函数在循环或者函数中不断被搜索.除此之外,一些研究针对混合执行(concolic execution)的优化问题[57,58]并提出了相应解决方案[59].

不难看出,上述针对符号执行的研究工作与本文所提出的 Afleer 同样存在正交关系,在下一步工作中,我们可以进一步融合这些方法,帮助 Afleer 更高效地搜索到未覆盖的分支,从而提升 Afleer 的效率.

5.3 符号执行与模糊测试结合的混合测试

目前已有很多关于符号执行与模糊测试(或随机测试)结合的混合测试.DART[16]与 SAGE[17]通过随机测试产生一定数量的测试用例,之后选取测试用例并运用混合执行来系统性地遍历程序.与 Afleer 相比,DART 与 SAGE 使用随机测试以及混合执行,而 Afleer 使用符号执行与基于覆盖率的灰盒测试.DART 与 SAGE 是单向的,其仅将随机测试生成的测试用例用于混合执行,而 Afleer 是双向交互的,因此能够更高效地生成测试用例.SYMFUZZ[18]通过符号执行分析两个程序输入之间的依赖关系,基于依赖信息计算测试用例的变异率,并用随机测试来进行变异.SYMFUZZ 不是直接将符号执行与随机测试结合,而是使用符号执行计算出一定的信息来提高随机测试的效率.Brian 等人[60]提出了一种结合符号执行与模糊测试的混合测试方法.与 Afleer 不同的地方在于:(1) 该方法使用随机测试生成测试用例来发现哪些基本块较容易被覆盖.基于该信息,利用符号执行来搜索难以覆盖的基本块.而 Afleer 直接使用模糊测试的覆盖信息来指导符号执行的搜索过程.(2) 不同于传统模糊测试,在该方法中,模糊测试不进行测试用例的变异操作,而仅用来运行符号执行生成的测试用例,以检测程序是否出现崩溃,因此,其并未充分利用模糊测试的信息及优势.

Majumdar 等人[2]提出了一种混合执行与随机测试交互执行的方法(简称 HCT),其通过随机测试产生测试用例,当随机测试收敛时(即无法生成新的测试用例)转到混合执行.一旦混合执行找到新的测试用例,则返回随机测试继续进行.Yun 等人[61]提出了一种适合混合测试的混合执行工具 QSYM,他们深入分析了影响混合测试的主要性能瓶颈:即混合执行测试.该方法与我们的方法是正交关系,通过引入其改进的混合执行测试方法 QSYM,可以提升 Afleer 的整体效率.Chao-Chun 等人[62]提出了一种以发现漏洞为目标的混合执行方法 CRAXfuzz.首先对一些与安全密切相关的函数进行 hook,例如 malloc、strcpy 以及 printf 等.之后基于某个种子进行混合执行测试,CRAXfuzz 采用目标驱动的搜索策略来检测被 hook 的函数是否具有安全问题.如果当前种子未发现漏洞,则用模糊测试或者符号执行生成新的种子进行尝试.

与 Afleer 比较类似的工作是 Saahil 等人[19]提出的方法 Munch.Munch 也是使用 KLEE 与 AFL 进行结合来提高测试的覆盖率,但是,这两种方法具有很大的不同:(1) 两者最大的不同在于目标不同,Afleer 侧重于提高分支覆盖率,而 Munch 侧重于提高函数覆盖率.因此,Afleer 采用细粒度的覆盖信息(即分支覆盖)来指导 KLEE 的搜索,而 Munch 采用粗粒度的函数覆盖来引导 KLEE 搜索(即 sonar-search).(2) 由于 AFL 默认也使用分支覆盖

引导并生成分支覆盖信息,因此,通过分支覆盖则可以更直接、更有效地将 KLEE 与 AFL 结合.例如,Afleer 中 KLEE 的搜索利用 AFL 的分支覆盖信息进行引导,从而避免搜索 AFL 已覆盖的分支.因此,KLEE 新生成的测试用例一定是 AFL 感兴趣的测试用例(该测试用例会覆盖 AFL 未覆盖的分支).因此,Afleer 中 AFL 与 KLEE 通过多次双向交互,可以更高效地提高分支覆盖率.在 AFL 中未考虑函数覆盖信息,Munch 中的两种策略都分别属于单向交互(即 KLEE 到 AFL 或者 AFL 到 KLEE).而在 Munch 的实验结果中(原文图 3)可以发现,由于覆盖率的反馈信息较粗,随着函数深度的逐渐加深,Munch 的两种策略并未比 AFL 有明显的提升.而通过细粒度的覆盖信息反馈,Afleer 在分支覆盖上有明显提升.(3) 与 Munch 相比,Afleer 在漏洞发现上更直接并且更有优势.因为漏洞通常处于难以覆盖的分支下,通过尽可能地覆盖更多分支则会更容易发现潜在的漏洞.而函数覆盖与漏洞发现则没有直接的关系.例如在图 1(a)中,Munch 的主要目标是函数覆盖,因此,任何一个不等于 123456789 的 input 都可以得到 100%的函数覆盖率(即当前函数与 func 函数都可以被覆盖),但不能发现条件 input==123456789 下的漏洞.Afleer 通过分支覆盖的引导,则可以直接检测该漏洞.(4) 两种方法虽然不同,但在某种程度上又具有一定的互补性.通过 Munch 首先覆盖难覆盖的函数,然后在该函数内利用 Afleer 更系统性地进行测试,从而可以尽可能地覆盖函数内全部分支.例如,在文献[19]的图 1 中,如果某个漏洞处于最深层函数(如 b0),则 Afleer 中 KLEE 会首先搜索父节点的函数,难以覆盖到深处的函数.而如果通过 Munch 先搜索到 b0,再进行分支覆盖搜索,则会更快地发现该漏洞.

Driller[21]将混合执行工具 Angr 与基于覆盖率的 AFL 相结合.首先运行 AFL,当 AFL 无法生成新的测试用例时,则 Angr 选取一个测试用例进行混合执行,直到发现新的测试用例,之后将新的测试用例再转交给 AFL 来继续进行.这两种方法与 Afleer 类似,都是通过符号执行/混合执行与灰盒测试/随机测试结合,并且双向交互.与 HTC 不同的是,Afleer 与 Driller 使用了灰盒测试,因此可以利用覆盖率信息来指导符号执行或混合执行的搜索,进而效率更高.而 Afleer 与 Driller 的不同之处在于:Afleer 运行在源码上,而 Driller 运行在二进制代码上;并且 Afleer 使用的是符号执行,而 Driller 使用的是混合执行.混合执行的优势是,如果从 AFL 的结果中选取高质量的测试用例执行,则有助于找到新分支.但在实际执行过程中,不是所有测试用例都是高质量的,例如:即使某个测试用例在控制流图上可以到达未覆盖的分支,但该测试用例在该路径下产生的约束条件仍可能无法使分支条件得到满足,此时就会影响效率.因此,其是一个平衡的过程,引入混合执行在某些情况下可以快速找到并覆盖处于深处的分支,但在某些条件下会出现大量的可到达但无法覆盖深分支的情况.基于上述分析,Afleer 没有采用混合执行方式,而是使用符号执行并依据覆盖信息(即 AFL 实时更新的文件)进行系统性的搜索,该策略首先可以保证更快速地覆盖浅位置的新分支.同时,基于覆盖信息以及控制流图可以运用更通用的搜索算法来遍历程序执行树.

6 总结与展望

本文深入分析了现有符号执行与模糊测试的优缺点,并提出了一种将符号执行与模糊测试相结合的新颖方法 Afleer.该方法综合利用了符号执行可以解决复杂条件的能力,模糊测试可以覆盖深分支的能力进而生成具有高覆盖率的测试用例.具体来说,模糊测试可以快速生成大量的测试用例,符号执行基于模糊测试的覆盖信息进行搜索,当搜索到新的未覆盖分支时则生成测试用例,模糊测试基于符号执行生成的测试用例继续变异.本文将 Afleer 用于标准程序集 LAVAM-M 以及实际项目 oSIP,最终结果验证了该方法的有效性.

我们在下一步工作中,将从符号执行与模糊测试本身存在的不足之处出发,进一步增强 Afleer 的能力.首先,将现有的基于污点分析以及有目标性的策略集成到模糊测试中,其可以提高模糊测试处理复杂分支的能力.其次,将前期的循环分析与总结工作[58−60]集成到符号执行中,以加速符号执行的效率.最后,如何基于模糊测试的动态信息来指导符号执行的搜索也将是下一步需要重点关注的课题.

References:

[1] Flexera. 2018. https://resources.flexera.com/web/pdf/Research-SVM-Vulnerability-Review-2018.pdf

[2] Majumdar R, Sen K. Hybrid concolic testing. In: Proc. of the Int'l Conf. on Software Engineering. IEEE, 2007. 416–426.

[3] Myers GJ, Sandler C, Badgett T. The Art of Software Testing. John Wiley & Sons, 2011.

[4] American fuzzy lop. http://lcamtuf.coredump.cx/afl/

[5] Peach fuzzer platform. http://www.peachfuzzer.com/products/

[6] honggfuzz. https://github.com/google/honggfuzz

[7] libFuzzer. http://llvm.org/docs/LibFuzzer.html

[8] King JC. Symbolic execution and program testing. Communications of the ACM, 1976,19(7):385–394.

[9] Cadar C, Dunbar D, Engler DR, *et al*. KLEE: Unassisted and automatic generation of highcoverage tests for complex systems programs. In: Proc. of the 8th USENIX Symp. on Operating Systems Design and Implementation. 2008. 209–224.

[10] Anand S, Pasâareanu C, Visser W. JPF-se: A symbolic execution extension to Java pathfinder. In: Proc. of the Int'l Conf. on Tools and Algorithms for the Construction and Analysis of Systems. Springer-Verlag, 2007. 134–138.

[11] Sen K, Agha G. Cute and Jcute: Concolic unit testing and explicit path model-checking tools. In: Proc. of the Int'l Conf. on Computer Aided Verification. Springer-Verlag, 2006. 419–423.

[12] Tillmann N, De Halleux J. Pex-white box test generation for .net. In: Proc. of the Int'l Conf. on Tests and Proofs. Springer-Verlag, 2008. 134–153.

[13] De Moura L, Bjørner N. Z3: An efficient smt solver. In: Proc. of the Int'l Conf. on Tools and Algorithms for the Construction and Analysis of Systems. Springer-Verlag, 2008. 337–340.

[14] Dutertre B. Yices 2.2. In: Proc. of the Int'l Conf. on Computer Aided Verification. Springer-Verlag, 2014. 737–744.

[15] Ganesh V, Dill DL. A decision procedure for bit-vectors and arrays. In: Proc. of the Int'l Conf. on Computer Aided Verification. Springer-Verlag, 2007. 519–531.

[16] Godefroid P, Klarlund N, Sen K. Dart: Directed automated random testing. In: Proc. of the 2005 ACM SIGPLAN Conf. on Programming Language Design and Implementation. ACM, 2005. 213–223.

[17] Godefroid P, Levin MY, Molnar DA, *et al*. Automated whitebox fuzz testing. In: Proc. of the Annual Network and Distributed System Security Symp., Vol. 8. 2008. 151–166.

[18] Cha SK, Woo M, Brumley D. Program-adaptive mutational fuzzing. In: Proc. of the IEEE Symp. on Security and Privacy. IEEE, 2015. 725–741.

[19] Ognawala S, Hutzelmann T, Psallida E, Pretschner A. Improving function coverage with Munch: A hybrid fuzzing and directed symbolic execution approach. In: Proc. of the 33rd Annual ACM Symp. on Applied Computing, SAC 2018. 2018. 1475–1482.

[20] Stephens N, Grosen J, Salls C, Dutcher A, Wang R, Corbetta J, Shoshitaishvili Y, Kruegel C, Vigna G. Driller: Augmenting fuzzing through selective symbolic execution. In: Proc. of the Annual Network and Distributed System Security Symp. 2016. 1–16.

[21] Dolan-Gavitt B, Hulin, Kirda EP, Leek T, Mambretti A, Robertson W, Ulrich F, Whelan R. LAVA: Large-scale automated vulnerability addition. In: Proc. of the IEEE Symp. on Security and Privacy. IEEE, 2016. 110–121.

[22] oSIP. https://www.gnu.org/software/osip/

[23] Lattner C, Adve V. LLVM: A compilation framework for lifelong program analysis & transformation. In: Proc. of the Int'l Symp. on Code Generation and Optimization: Feedback-directed and Runtime Optimization. IEEE Computer Society, 2004. 75.

[24] Whole Program LLVM. https://github.com/travitch/whole-program-llvm

[25] Bohme M, Van-Thuan P, Roychoudhury A. Coverage-based greybox fuzzing as Markov chain. In: Proc. of the 2016 ACM SIGSAC Conf. on Computer and Communications Security. ACM, 2016. 1032–1043.

[26] Li YK, Chen BH, Chandramohan M, Lin SW, Liu Y, Tiu A. Steelix: Program-state based binary fuzzing. In: Proc. of the 11th Joint Meeting on Foundations of Software Engineering. ACM, 2017. 627–637.

[27] Wang JJ, Chen BH, Wei L, Liu Y. Skyfire: Data-driven seed generation for fuzzing. In: Proc. of the IEEE Symp. on Security and Privacy. IEEE, 2017. 579–594.

[28] RFC 3261. https://www.ietf.org/rfc/rfc3261.txt

[29] oSIPBug. https://savannah.gnu.org/bugs/?54227

[30] Chen P, Chen H. Angora: Efficient fuzzing by principled search. arXiv Preprint arXiv: 1803.01307, 2018.

[31] Bohme M, Van-Thuan P, Nguyen MD, Roychoudhury A. Directed greybox fuzzing. In: Proc. of the 2017 ACM SIGSAC Conf. on Computer and Communications Security. ACM, 2017. 2329–2344.

[32] Chen HX, Xue YX, Li YK, Chen BH, Xie XF, Wu XH, Liu Y. Hawkeye: Towards a desired directed grey-box fuzzer. In: Proc. of the 2018 ACM SIGSAC Conf. on Computer and Communications Security. ACM, 2018. 2095–2108.

[33] Ganesh V, Leek T, Rinard M. Taint-based directed whitebox fuzzing. In: Proc. of the 31st Int'l Conf. on Software Engineering. IEEE Computer Society, 2009. 474–484.

[34] Lemieux C, Sen K. Fairfuzz: Targeting rare branches to rapidly increase greybox fuzz testing coverage. arXiv Preprint arXiv: 1709.07101, 2017.

[35] Wang TL, Wei T, Gu GF, Zou W. Taintscope: A checksum-aware directed fuzzing tool forautomatic software vulnerability detection. In: Proc. of the IEEE Symp. on Security and Privacy. IEEE, 2010. 497–512.

[36] Haller I, Slowinska A, Neugschwandtner M, Bos H. Dowsing for overflows: A guided fuzzer to find buffer boundary violations. In: Proc. of the 22nd USENIX Security Symp. 2013. 49–64.

[37] Rawat S, Jain V, Kumar A, Cojocar L, Giuffrida C, Bos H. Vuzzer: Application-aware evolutionary fuzzing. In: Proc. of the Network and Distributed System Security Symp. (NDSS). 2017.

[38] Rebert A, Cha SK, Avgerinos T, Foote JM, Warren D, Grieco G, Brumley D. Optimizing seed selection for fuzzing. In: Proc. of the 23rd USENIX Security Symp. 2014.

[39] Woo M, Cha SK, Gottlieb S, Brumley D. Scheduling black-box mutational fuzzing. In: Proc. of the 2013 ACM SIGSAC Conf. on Computer & Communications Security. ACM, 2013. 511–522.

[40] Gan ST, Zhang C, Qin XJ, Tu XW, Li K, Pei ZY, Chen ZN. Collafl: Path sensitive fuzzing. In: Proc. of the IEEE Symp. on Security and Privacy. IEEE, 2018.

[41] Chipounov V, Kuznetsov V, Candea G. S2e: A platform for in-vivo multi-path analysis of software systems. In: Proc. of the 16th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems. ACM, 2011. 265–278.

[42] Cadar C, Ganesh V, Pawlowski PM, Dill DL, Engler DR. Exe: Automatically generating inputs of death. ACM Trans. on Information and System Security, 2008,12(2):10.

[43] Burnim J, Sen K. Heuristics for scalable dynamic test generation. In: Proc. of the 23rd IEEE/ACM Int'l Conf. on Automated Software Engineering. IEEE Computer Society, 2008. 443–446.

[44] Xie T, Tillmann N, de Halleux J, Schulte W. Fitness-guided path exploration in dynamic symbolic execution. In: Proc. of the Int'l Conf. on Dependable Systems & Networks. Citeseer, 2009. 359–368.

[45] Zhang YF, Chen ZB, Wang J, Dong W, Liu ZM. Regular property guided dynamic symbolic execution. In: Proc. of the 37th Int'l Conf. on Software Engineering. IEEE Press, 2015. 643–653.

[46] Li Y, Su ZD, Wang LZ, Li XD. Steering symbolic execution to less traveled paths. In: Proc. of the 2013 ACM SIGPLAN Int'l Conf. on Object Oriented Programming Systems Languages & Applications, OOPSLA 2013. 2013. 19–32.

[47] Wang HJ, Liu T, Guan XH, Shen C, Zheng QH, Yang ZJ. Dependence guided symbolic execution. IEEE Trans. on Software Engineering, 2016,43(3):252−271.

[48] Bugrara S, Engler D. Redundant state detection for dynamic symbolic execution. In: Proc. of the 2013 USENIX Conf. on Annual Technical Conf. USENIX Association, 2013. 199–212.

[49] Gan ST, Wang LZ, Xie XH, Qin XJ, Zhou L, Chen ZN. Guided symbolic execution method based on program function label slice. Ruan Jian Xue Bao/Journal of Software, 2019 (in Chinese with English abstract). [doi: 10.13328/j.cnki.jos.005562]

[50] Trabish D, Mattavelli A, Rinetzky N, Cadar C. Chopped symbolic execution. In: Proc. of the Int'l Conf. on Software Engineering. 2018. 5.

[51] Kuznetsov V, Kinder J, Bucur S, Candea G. Efficient state merging in symbolic execution. In: Proc. of the 33rd ACM SIGPLAN Conf. on Programming Language Design and Implementation. ACM, 2012. 193–204.

[52] Sen K, Necula G, Gong L, Choi W. MultiSE: Multi-path symbolic execution using value summaries. In: Proc. of the 10th Joint Meeting on Foundations of Software Engineering. ACM, 2015. 842–853.

[53] Godefroid P. Compositional dynamic test generation. In: Proc. of the 34th Annual ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages. ACM, 2007. 47–54.

[54] Xie XF, Chen BH, Zou L, Liu Y, Le W, Li XH. Automatic loop summarization via path dependency analysis. IEEE Trans. on Software Engineering, 2019,45(6):537−557.

[55] Xie XF, Chen BH, Liu Y, Le W, Li XH. Proteus: Computing disjunctive loop summary via path dependency analysis. In: Proc. of the 24th ACM SIGSOFT Int'l Symp. on Foundations of Software Engineering. ACM, 2016. 61−72.

[56] Xie XF, Liu Y, Le W, Li XH, Chen HX. S-looper: Automatic summarization for multipath string loops. In: Proc. of the 2015 Int'l Symp. on Software Testing and Analysis. ACM, 2015. 188−198.

[57] Wang XY, Sun J, Chen ZB, Zhang PX, Wang JY, Lin Y. Towards optimal concolic testing. In: Proc. of the Int'l Conf. on Software Engineering. 2018.

[58] Cui ZQ, Wang LZ, Li XD. Target-directed concolic testing. Chinese Journal of Computers, 2011,34(6):953−964 (in Chinese with English abstract).

[59] Jia XY, Ghezzi C, Ying S. Enhancing reuse of constraint solutions to improve symbolic execution. In: Proc. of the 2015 Int'l Symp. on Software Testing and Analysis. ACM, 2015. 177−187.

[60] Pak BS. Hybrid fuzz testing: Discovering software bugs via fuzzing and symbolic execution [Ph.D. Thesis]. School of Computer Science, Carnegie Mellon University, 2012.

[61] Yun I, Lee S, Xu M, Jang Y, Kim T. QSYM: A practical concolic execution engine tailored for hybrid fuzzing. In: Proc. of the 27th USENIX Security Symp. (USENIX Security 18). Baltimore: USENIX Association, 2018. 745−761.

[62] Yeh CC, Chung H, Huang SK. Craxfuzz: Target-aware symbolic fuzz testing. In: Proc. of the 39th IEEE Annual Computer Software and Applications Conf. (COMPSAC), Vol. 2. IEEE, 2015. 460−471.

附中文参考文献:

[49] 甘水滔,王林章,谢向辉,秦晓军,周林,陈左宁.一种基于程序功能标签切片的制导符号执行分析方法.软件学报,2019. [doi: 10.13328/j.cnki.jos.005562]

[58] 崔展齐,王林章,李宣东.一种目标制导的混合执行测试方法.计算机学报,2011,34(6):953−964.

谢肖飞(1989－),男,山西运城人,博士,主要研究领域为程序分析、测试,形式化验证.

孟国柱(1987－),男,博士,副研究员,CCF专业会员,主要研究领域为软件与系统安全.

李晓红(1965－),女,博士,教授,博士生导师,CCF 高级会员,主要研究领域为安全软件工程,可信软件,信息安全.

刘杨(1982－),男,博士,副教授,主要研究领域为 Formal Methods,Security,Software Engineering,Multi-agent Systems.

陈翔(1980－),男,博士,副教授,CCF 高级会员,主要研究领域为软件缺陷预测,软件缺陷定位,回归测试,组合测试.