软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software, 2022,33(11):3967–3982 [doi: 10.13328/j.cnki.jos.006331]

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

有效覆盖引导的定向灰盒模糊测试*

杨 克[1,2], 贺也平[1,2,3], 马恒太[1,2], 蔡春芳[1,2], 谢 异[1,2], 董 柯[1,2]

[1](基础软件国家工程研究中心(中国科学院 软件研究所), 北京 100190)

[2](中国科学院大学, 北京 100049)

[3](计算机科学国家重点实验室(中国科学院 软件研究所), 北京 100190)

通信作者: 贺也平, E-mail: yeping@iscas.ac.cn; 马恒太, E-mail: hengtai@iscas.ac.cn

摘 要: 定向灰盒模糊测试技术在度量种子对目标执行状态的搜索能力时, 除了考虑种子逼近目标代码的程度之外, 还需要分析种子对多样化执行状态的发现能力, 从而避免陷入局部最优. 现有的定向灰盒模糊测试主要根据全程序的覆盖统计来度量种子搜索多样化执行路径的能力. 然而, 目标执行状态仅依赖于部分程序代码. 如果带来新覆盖的种子并未探索到目标状态计算所依赖的新执行状态, 其不仅不能扩大种子队列对目标执行状态的搜索能力, 而且会诱导测试目标无关的代码和功能, 阻碍定向测试向目标代码的收敛. 为了缓解该问题, 从待发现目标执行状态依赖代码的覆盖统计着手, 提出了一种有效覆盖引导的定向灰盒模糊测试方法. 利用程序切片技术提取影响目标执行状态计算的代码. 通过能量调度(即控制种子后代生成数量), 提升引发该部分代码控制流新覆盖变化的种子能量, 降低其他冗余种子的能量, 使定向灰盒模糊测试专注于搜索目标相关的执行状态. 在测试集上的实验结果显示, 该方法显著提升了目标状态发现效率.

关键词: 定向模糊测试; 有效覆盖; 冗余种子; 能量调度; 程序切片

中图法分类号: TP311

中文引用格式: 杨克, 贺也平, 马恒太, 蔡春芳, 谢异, 董柯. 有效覆盖引导的定向灰盒模糊测试. 软件学报, 2022, 33(11): 3967–3982. http://www.jos.org.cn/1000-9825/6331.htm

英文引用格式: Yang K, He YP, Ma HT, Cai CF, Xie Y, Dong K. Guiding Directed Grey-box Fuzzing by Target-oriented Valid Coverage. Ruan Jian Xue Bao/Journal of Software, 2022, 33(11): 3967–3982 (in Chinese). http://www.jos.org.cn/1000-9825/6331.htm

Guiding Directed Grey-box Fuzzing by Target-oriented Valid Coverage

YANG Ke[1,2], HE Ye-Ping[1,2,3], MA Heng-Tai[1,2], CAI Chun-Fang[1,2], XIE Yi[1,2], DONG Ke[1,2]

[1](National Engineering Research Center of Fundamental Software (Institute of Software, Chinese Academy of Sciences), Beijing 100190, China)

[2](University of Chinese Academy of Sciences, Beijing 100049, China)

[3](State Key Laboratory of Computer Science (Institute of Software, Chinese Academy of Sciences), Beijing 100190, China)

Abstract: Directed grey-box fuzzing measures the effectiveness of seeds for detecting the execution path towards the target. In addition to the closeness between the triggered execution and the target code lines, the ability to explore diversified execution paths is also important to avoid local optimum. Current directed grey-box fuzzing methods measure this capability by coverage counting of the whole program. But only a part of the program is responsible for the calculation of the target state. If the new seed brings target irrelevant state changes, it cannot enhance the queue for state exploration. What is worse, it may distract the concentration of the fuzzer and waste time on exploring target irrelevant code logic. To solve this problem, this study provides a valid coverage guided directed grey-box fuzzing method. The static program slicing technique is used to locate the code region that can affect the target state and detect interesting seeds that bring new differences in coverage of this code region. By enlarging the energy of these seeds and reducing others (adjusting power schedule), the

* 基金项目: 中国科学院战略性先导科技专项(XDA-Y01-01, XDC02010600)

收稿时间: 2020-11-09; 修改时间: 2021-01-06; 采用时间: 2021-03-03; jos 在线出版时间: 2021-08-03

fuzzer can be guided to focus on seeds that can help explore different control flow that target depends and mitigate the interference of redundant seeds. The experiment on the benchmark provided shows that this strategy brings significant performance improvement for AFLGO.

Key words: directed fuzzing; valid coverage; redundant seed; power schedule; program slicing

定向灰盒模糊测试[1]在覆盖率引导的灰盒模糊测试基础上增加对待测代码区域的引导机制, 从而实现对所关注代码片段的集中测试或者搜索触发特定执行状态的测试输入. 其继承了灰盒模糊测试开发维护成本低、使用简单、可扩展性好等优点, 可应用于补丁/变更测试、错误复现、疑似缺陷或漏洞状态的触发验证等应用场景.

覆盖率引导的灰盒模糊测试主要使用遗传算法搜索执行路径, 即将输入作为种子, 通过交叉变异产生后代, 借助覆盖统计等手段识别触发新执行路径的种子. 如此不断繁衍, 从而实现对执行路径的探索. 种子队列保留了进化过程中的所有代的种子. 定向灰盒模糊测试仅希望发现抵达目标代码或目标执行状态的执行路径, 因此重点选择哪些种子进行交叉变异更有利于目标执行状态的搜索, 是定向模糊测试的关键问题.

基于种子调度实现测试聚焦包含两个方面: 一是度量和发挥种子对目标相关状态的搜索能力; 二是去除或抑制种子对目标无关状态的搜索, 即排除或抑制冗余种子. 现有的定向灰盒模糊测试研究大都仅关注于前者, 忽视了后者的重要性.

现有的定向灰盒模糊测试研究从距离目标代码的控制流距离[1,2]、序列覆盖率[3,4]以及触发目标代码区域的输入内容特征[5]等角度度量种子触发执行路径与目标代码的逼近程度, 从执行路径覆盖差异和分支触发频率[6]等角度度量种子对深度执行状态的搜索能力. 根据对种子的评估结果, 控制种子在每轮迭代中的后代生成数量(也称为能量(energy))[1−4,6], 或过滤掉明显无法抵达目标的种子[5]来实现对目标代码的聚焦测试. 但是在覆盖统计时, 它们并未分析新覆盖的代码段或新触发的执行状态与目标执行状态的关系, 所有带来新覆盖的种子都被加入测试队列中.

然而, 种子队列中的种子并非越多越好[7]. 如果新加入的种子触发的新执行状态与目标执行状态的计算无关, 其不仅不能提升种子队列对目标状态的搜索能力, 反而会带来副作用. 即占用测试时间, 诱导定向测试探索目标无关执行状态, 阻碍测试聚焦. 因此, 识别和排除冗余的种子对于定向测试的聚焦非常重要.

新种子若能提升现有种子集合对目标状态的搜索能力, 则称其为有效增益种子(又称为有效覆盖增益种子),否则称其为冗余种子. 有效增益种子触发了影响目标状态计算的新执行状态, 有助于提升种子队列对目标状态的搜索能力. 而冗余种子则无助于目标的发现, 且会带来前面提到的副作用. 目标依赖的执行状态变化是目标依赖的代码形成的, 如果能够确定新的覆盖变化发生于目标依赖的代码, 就可以判定种子是否带来有效覆盖增益.

目标依赖的代码可以借助静态分析获取, 结合现有的覆盖统计技术, 可以监控新种子是否带来对目标搜索有帮助的覆盖变化. 通过对新种子按是否带来有效覆盖增益加以区分, 并减少冗余种子的能量, 可以缓解目标无关状态变化对测试精力的分散作用, 提升定向测试效率.

为此, 本文从种子带来的覆盖增益的有效性角度来进一步对种子优劣进行度量. 利用程序切片技术识别目标状态的数据依赖和控制依赖, 获得对目标状态有影响的代码段. 在此基础上, 本文提出了一种有效覆盖引导的定向灰盒模糊测试方法, 增加有效覆盖增益种子的能量, 减少其他冗余种子的能量, 从而提升定向模糊测试的效率.

主要贡献如下:

(1) 提出了一种基于程序切片的有效覆盖识别和有效覆盖增益种子判定方法. 从覆盖变化能否影响目标依赖的计算过程的角度给出了一种有效性度量, 并给出了一种该方法的具体实现.

(2) 基于该度量, 提出了一种增加有效覆盖增益种子测试时间占比的定向模糊测试的能量分配策略. 重点对触发目标依赖的新执行状态的种子进行交叉变异, 使得定向模糊测试更快地收敛到目标区域, 提升对目标执行状态的发现效率;

(3) 实现了原型系统 VCov, 在 6 款开源应用软件构造的测试数据集上取得了平均 1.5 倍于 AFLGO 目标执行状态检测效率提升, 并能够保持相同的多样性路径发现能力.

1 研究动机

本节通过一个案例来说明在对代码位置明确的目标状态生成测试输入的任务中, 冗余种子对目标执行状态的发现时间的影响.

例 1: 目标无关种子影响格式字段字节判断的求解时间

本例是在Hawkeye的评估中选择的测试案例. 该案例是从Google的测试套件(fuzzer test suite)中选取的对JPEG 图片处理库 libjpeg-turbo-2017 中特定代码行 jdmarker:659(图 1 第 16 行)的可达性测试. Google 测试套件提供了一个 JPEG 初始种子. 图 1 显示了目标行在 *examine_app*0 函数中所依赖的关键条件. *examine_app*0 函数在解析符合条件的 JPEG 文件时只执行一次. 下划线标记的部分是初始种子不匹配的条件, 也就是 *data*[2], *data*[3], *data*[5], 模糊测试器必须将相应的输入更正才能触发目标行代码.

```
1   LOCAL(void)
2   examine_app0 (j_decompress_ptr cinfo,JOCTET*data,
3                 unsigned int datalen,JLONG remaining)
4   { ...
5       } else if (datalen≥6 &&
6           GETJOCTET(data[0])==0x4A &&
7           GETJOCTET(data[1])==0x46 &&
8           GETJOCTET(data[2])==0x58) &&
9           GETJOCTET(data[3])==0x58) &&
10          GETJOCTET(data[4])==0) {
11        switch (GETJOCTET(data[5])) {
12        case 0x10:
13          TRACEMS1(cinfo,1,JTRC_THUMB_JPEG,(int) totallen);
14          break;
15        case 0x11:
16          //target
17  ... }
```

图 1 Google 测试套件中关于 libjpeg-turbo 用于定向测试的一个目标行

使用 AFLGO 对抵达该目标的执行进行复现, 实验环境: Ubuntu16.04 3GB 内存, 物理 CPU Inter Corei5 4200U 的虚拟机. 该例目标状态依赖的计算是顺序执行的, 因此 AFLGO 的距离估算比较准确. AFLGO 使用模拟退火算实现种子到目标代码的距离优化, 冷却时间设置得越小, 越早收敛至对目标较近的种子的测试. 本例中将冷却时间设置为 1 min, 这是 AFLGO 提供的最小冷却时间单位. 在发现目标前, 对于每一个保留在模糊测试队列中的种子(interesting seed), 我们记录了 3 个值: 生成时间、与目标的距离、当前队列种子中的最小距离(全局最短距离). 图 2 绘制了对本例一次测试的记录情况.

其中, 实心点记录了 AFLGO 在发现抵达目标的种子前, 每一个新发现的有趣种子的生成时间(横轴, 单位: ms)以及与目标的距离(左侧纵轴); 最下方的黑点对应的种子距离为–1, 即 AFLGO 认定为无穷远的种子; 蓝线连接的空心点记录了队列中所有输入的最短距离(右侧纵轴)随时间(与实心点共享横轴)的变化. 本次测试过程在发现目标前一共生成了 394 个“有趣(interesting)”的种子. 由于图 1 中的代码仅被执行一次, 且触发这些分支前无复杂计算, 因此最短距离反映了这些关键的数值比较的求解过程. 从最短距离随时间的变化上可以看出, 本轮测试中超过 90%的时间都花费在最后一个分支条件(*data*[5]的比较)的突破上面. 问题是, 前两个分支的条件也是字节比较, 为什么突破它们花费的时间差别如此巨大呢?

通过分析记录的数据, 我们发现, 这个最耗时的分支条件(*data*[5]的比较)的首次到达时间是 7 164 ms, 此后, 新发现的 173 个测试输入中仅有 5 个可以到达对 *data*[5]的比较判断. 这段时间, AFLGO 仍然需要分配至少 1/32 原始 AFL 的能量变异 389 个未抵达 *data*[5]判断条件的种子. AFLGO 对队列种子的选取是轮询式的, 且带来全程序中任意控制流覆盖变化的种子都会被视为有趣的种子加入队列中. 随着带来不相关路径覆盖变

化的新种子不断加入队列，真正有效的种子需要等待更久才被轮到. 因此越到后面，这些关键的比较条件的求解时间就越长.

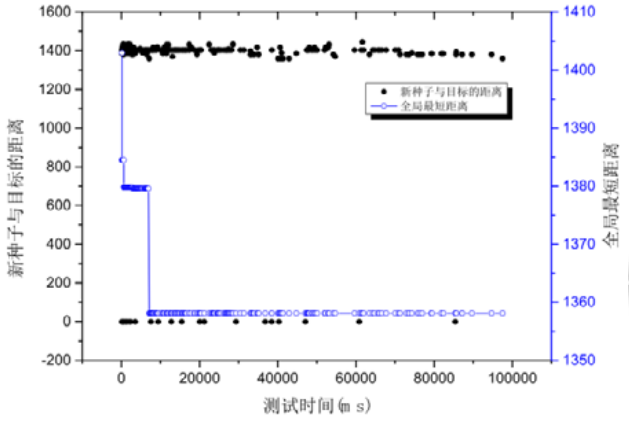

图 2　在触发目标 jdmarker: 659 前 AFLGO 新发现的种子距离与队列中当前的最短距离随时间的变化趋势

虽然总是选择距离目标最近的种子进行变异(例如 Hawkeye[2]近距离种子优先策略)可以提升例 1 的目标行的触发效率，但这种策略是以牺牲执行路径的多样性为代价的，容易陷入局部最优.

为了避免引发目标无关执行状态变化的冗余种子过多地占用测试时间，误导测试方向，本文从种子带来的覆盖增益与目标的相关性角度，识别和重用有效覆盖增益种子，进而提升定向灰盒模糊测试的效率.

2　有效覆盖引导方法

2.1　方法原理

定向灰盒模糊测试在覆盖率引导的灰盒模糊测试上发展而来. 现有的定向灰盒模糊测试仍然依赖于遗传算法，通过提升输入所触发的执行路径的多样性，来实现目标依赖的执行状态的搜索. 其主要算法框架如算法 1 所示，其中，种子选择策略(对应算法 1 中的 *chooseNext*)、能量赋值策略(即控制种子后代生成数量的策略，对应算法 1 中的 *assignEnergy*)、交叉变异方法(对应算法 1 中的 *mutate_input*)以及适应度函数(对应算法 1 中的 *isInteresting*)是覆盖率引导的灰盒模糊测试的关键环节，也是定向灰盒模糊测试引导策略设计所关心的部分. 例如，AFLGO，SCDF 主要通过修改种子的能量赋值策略(*assginEnergy*)来引导定向模糊测试，Hawkeye 和 RDFuzz 除修改能量赋值之外，对变异方法也进行了优化，Hawkeye 还增加了对种子排序(*chooseNext*)的改进.

算法 1. 灰盒模糊测试框架.

输入: Seed Inputs *S*.

```
1:  repeat
2:    s=chooseNext(S)
3:    p=assignEnergy(s)
4:    for i from 1 to p do
5:      s′=mutate_input(s)
6:      if s′ crashes then
7:        add s′ to S_x
8:      else if isInteresting(s′) then
9:        add s′ to S
10:     end if
```

11: **end for**

12: **until** timeout reached or abort-signal

输出: Crashing Inputs $S_{\times}$.

由于输入空间巨大, 灰盒模糊测试需要对输入进行等价类划分. 同类输入仅保留一个(根据 *isInteresting* 判断)加入算法 1 中集合 S 中. 以 AFL 为例, 其根据基本块间控制流迁移的频次统计向量来标识一条执行路径, 将该向量压缩存储于一段固定大小(64 KB)内存中作为该路径的唯一标识. 在测试中, 根据其中值为 1 的比特位对全局覆盖记录中相应的比特位置 0(初始值为 1). 通过监控全局内存是否有新的比特位被置 0(即判定种子是否带来覆盖增益), 进而判定新生成的种子是否触发了未见的执行路径. 通过不断地迭代, 增加 S 中种子的数量, 即可增加整体程序的覆盖率.

然而, 定向灰盒模糊测试的有效测试区域不再是整体程序. 此时, 基于全程序覆盖统计的执行路径等价类划分不再适用. 新种子仅尽管带来了覆盖增益, 但是如果该覆盖增益对应的执行差异不影响目标状态计算, 其对搜索目标无益. 将这些种子加入队列会造成冗余, 影响测试性能[7]. 故应基于目标状态依赖代码的覆盖统计进行种子划分, 即只有目标依赖代码的控制流频次变化形成的覆盖增益才对搜索目标状态才是有效的(即有效覆盖增益). 因此, 本文根据覆盖增益与目标的关联性对新种子进行分类.

本文尝试建立覆盖增益与影响目标状态计算的代码段的联系, 以分辨种子对目标状态的搜索效用, 进一步识别和降低冗余. 借用 AFL 的覆盖统计的机制, 将种子带来的覆盖增益进行再评估. 目标依赖代码上的覆盖增益被认为是有效的. 触发有效覆盖增益的种子被称为有效覆盖增益种子, 简称有效增益种子. 通过能量调度, 对有效覆盖增益种子重点交叉变异, 可以促进定向灰盒模糊测试的聚焦.

为此, 本文利用静态程序切片技术来确定目标依赖的代码, 根据这些代码的覆盖情况识别有效覆盖增益种子, 分配其更多能量, 减少其他种子能量(算法 1 中的 *assginEnergy*), 从而引导定向模糊测试专注于目标依赖执行状态的搜索, 提升目标状态的搜索效率.

2.2 目标执行状态依赖代码的识别

本文使用基于程序依赖图的逆向程序切片技术[8]来识别抵达目标执行状态所需执行的代码. 考虑图 3 中的代码, 假设待触发目标执行状态的触发语句在 processB 函数中并依赖于其参数的取值, 则不需要关心 processA 函数和 print 函数中的计算逻辑, 因为其不会影响目标状态的计算.

图 3 中标记了控制第 6 行 processB 函数调用的可达性以及对其实参 *b* 的值有影响的行. 图 4 显示了图 4 中代码的程序依赖图(PDG). 程序依赖图主要包含控制依赖和数据依赖两部分. 图 4 中的每一个方块表示以 SSA 表示的一条语句, *ENTRY* 和 *EXIT* 是为了控制流分析方便额外添加的入口和出口. 简洁起见, 图 4 并未画出 *EXIT* 对 *ENTRY* 的控制依赖. 图 4 中, 虚线表示数据依赖, 实线表示控制依赖. 数据依赖可以根据控制流上变量的"定义-使用"链分析获得, 而控制依赖主要根据控制流计算出的后主导树分析得到. 根据切片准则, 即 *ProcessB*(*b*)语句和变量 *b* 在程序依赖图上逆向追溯其控制依赖和数据依赖, 进而得到加粗虚线和加粗实线表示的数据和控制依赖关系以及相关的语句(用浅灰色填充的方块). 得到的语句与图 3 阴影部分相同. 结合函数调用图, 可以完成过程间的程序切片, 实现全程序的切片分析.

```
1  int main(·) {
2     int a=Input1(·);
3     char *b=Input2(·);
4     if (b){
5        ProcessA(a);
6        ProcessB(b);
7     }
8     Print(a,b);
9  }
```

图 3 程序切片代码实例

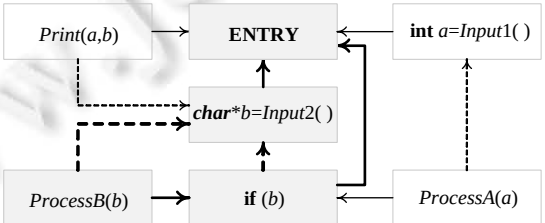

图 4 图 3 代码对应的程序依赖图

在 AFL 中, 以控制流频次的统计向量来记录某输入触发的代码覆盖, 并以该统计向量对全局统计向量的

1 比特位置位情况来判定该种子是否触发了新的执行路径. 因此, 在识别了目标依赖代码的前提下, 只需关注这些代码相关的控制流对应的比特位置位情况, 就可以判定其是否带来了对搜索目标有效的覆盖变化.

2.3 有效覆盖的标识和计算

由于静态切片技术难以保证提取的代码子集的完整性和准确性, 模糊测试仅能根据完整的程序代码构建待测可执行程序, 进行插桩和覆盖统计. 因此需要将切片后的代码与原始程序的对应起来, 才能进行有效覆盖的统计. 图 5 和图 6 显示了图 4 中例子程序切片前后的控制流: 图 5 是原始控制流, 图 6 是切片后的控制流. 很明显, 切片后的控制流和基本块相较于原始程序都有变化. 切片后基本块中的语句变少了, 而且部分控制流已经无法与原始程序直接对应了.

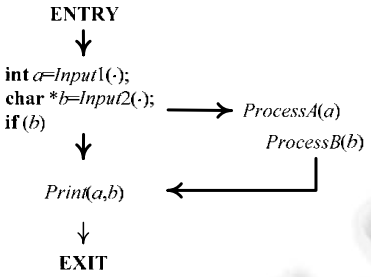

图 5 原始程序的控制流图

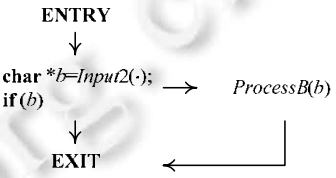

图 6 切片后的控制流图

但是, 切片后的基本块是原始基本块的一部分, 二者是一一对应的. 根据切片后的基本块, 可以定位其原始基本块. 在原始控制流图上, 从这些有对应关系的基本块出发或者抵达这些基本块的控制流边称为有效的控制流. 执行路径对有效控制流的覆盖即为有效覆盖. 如果某个种子触发了以往未见的有效控制流频次, 则认为其对搜索目标有作用.

在具体实现时, 根据 LLVM 字节码中的调试信息完成切片后的基本块与原始程序中基本块的对应. 图 5 中, 加粗的控制流边显示了利用该方法从图 6 中切片后的控制流上的基本块映射到原始基本块后确定的相关控制流. 本文使用的静态切片工具 DG[8]进行目标相关代码的抓取, 该工具是基于 LLVM 的中间码进行程序分析, 将切片得到的控制流信息按照函数名分别保存在不同的文件中. 根据切片后的基本块的调试信息与 AFLGO 插装步骤获得的源程序中的基本块进行对应, 调试信息包含文件名、行号、列号. 根据调试信息确定切片基本块与原程序中基本块的从属关系.

本文方法 VCov 在 AFLGO 的基础上增加了有效覆盖统计功能. AFLGO 使用随机数来标识基本块, 因此需要建立这些随机数标识与控制流图上基本块的索引. 为了保证准确性, 记录了 AFLGO 对基本块插桩(第 2 次编译)时基本块的拓扑信息(与三级前驱和一级后继形成的子图)和为其生成的随机数标识, 从而建立与随机数标识到控制流图上基本块的对应关系. 该对应关系供模糊测试时统计有效覆盖时使用.

首先, 根据切片信息定位到原始控制流$\langle BB_i,BB_j\rangle$(BB_i, BB_j 为该控制流的起始和结束基本块 i, j 为唯一标识基本块的下标); 其次, 根据随机数标识的索引查找到 BB_i 和 BB_j 对应的随机数标识 RID_i, RID_j; 最后, 根据哈希公式$(RID_i>>1)\oplus RID_j$ 确定该控制流在 AFL 共享内存(记录控制流频次的向量)中对应的字节位置. 在实现时, 用一个位向量(8 KB, 算法 2 中的 *mask*)来标记 AFL 共享内存(64 KB)中的有效的控制流.

算法 2 显示了在 AFLGO 基础上更改后的覆盖统计算法. 该算法根据新生成的种子的覆盖记录 *cur*、全局覆盖记录 *vir* 以及有效覆盖掩码 *mask* 来判定新种子带来的覆盖增益有效性. 改动部分主要在第 4–6 行,通过掩码来判定该覆盖变化是否是有效, 其余部分与 AFLGO 相同. AFLGO 沿用了 AFL 的覆盖统计方法, 以控制流频次的压缩向量来表示执行路径. 其将该向量存储在一个 64 KB 的共享内存中, 每条边根据前面提到的哈希公式映射射到一个字节. 每个字节存储取值为 0–255 的频次信息.

算法 2. *has_new_bits*(*s*).

输入: 有效覆盖掩码 *mask*.

全局覆盖更新 *vir*(大小 *MAP_SIZE*=2^{16} 字节);

新生成种子的覆盖 *cur*(大小 *MAP_SIZE*=2^{16} 字节);

```
1:  valid_cov_change=0
2:  i:=0
2:  while (i<MAP_SIZE) {
3:    if cur[i] && (cur[i] & vir[i])
4:      if mask[i>>3] & (0x80>>(i%8))
5:        valid_cov_change=1
6:      end if
7:      if (ret<2)
8:        if cur[i] && vir[i]==0xff
9:          ret:=2
10:       else
11:         ret:=1
12:       end if
13:     end if
14:     vir[i] & =~cur[i]
15:   end if
16:   i:=i+1
17: end while
```

输出: 种子带来的全覆盖变化类型 *ret*(0 丢弃/1 频度变化/2 覆盖新控制流),

valid_cov_change 种子是否带来有效覆盖的变化(0 无变化/1 有变化).

AFL 使用一段 64 KB 内存(对应算法 1 中的 *vir*)用于全局覆盖统计. 初始时, 每字节都是 0xFF. 在测试中, 根据新种子运行后的共享内存(对应算法 1 中的 *cur*)中值为 1 的比特位将全局覆盖统计向量 *vir* 对应位置 0(算法 1 第 14 行). 新种子生成后, 根据其是否引发全局覆盖统计向量(对应算法 1 中的 *vir*)的变化来判断其是否带来新覆盖. 因此, 当发现需要置位时, 根据掩码 *mask* 判断该字节对应的控制流边是否是有效控制流边, 即可识别有效覆盖增益(算法 1 的第 4–6 行).

2.4 降冗策略

理论上, 如果切片结果准确且完整, 降冗策略可以设计为仅保留带来有效覆盖增益的种子, 丢弃冗余种子. 在实际中, 受到目标信息的完整性以及 DG 切片工具自身精度的影响, 所获得的目标依赖的代码可能不完整或不准确. 为了避免在这些特殊场景下产生对种子的误删, 本文通过能量缩放来保证方法的可用性. 在能量分配时, 将有效覆盖增益种子的能量扩大ζ倍, 并将未能带来有效覆盖增益的种子能量缩小ζ倍. 本文的实验中, ζ=2. ζ的取值可以根据冗余种子在相同优劣级种子集合的占比来自适应调节, 本文将其作为未来的研究工作.

2.5 方法框架

图 7 显示了 VCov 的主要框架, 其中, 深灰色部分主要是在 AFLGO 的基础上变更或增加的模块, 浅灰色部分是比 AFLGO 多出的中间结果. VCov 在 AFLGO 的基础上实现, 主要修改了代码插桩部分以及模糊测试器, DG Slicer 是引入的切片模块, 其根据第 1 次编译得到的 LLVM 字节文件(bc 文件)进行切片. VCov Indexer 主要用于匹配两次编译的控制流图, 进而将基本块对应到其随机标识(前文提到的 RID)上. 虚线框起来的部

分是静态分析. 虚线框右侧的“Fuzzer”则是模糊测试主程序. 该部分主要增加了切片控制流读取和映射, 有效覆盖增益的监控和能量缩放. 其余部分与 AFLGO 的结构相同.

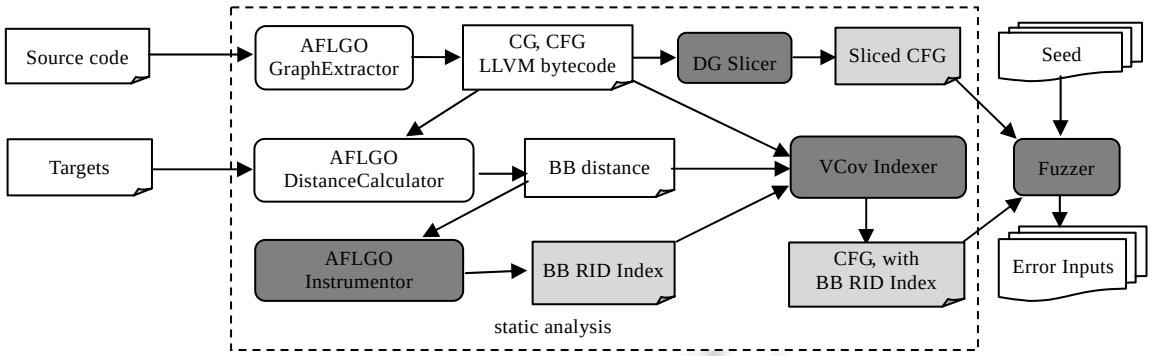

图 7 VCov 模糊测试流程概览

3 实验评估

本文从有效覆盖统计的角度来缓解冗余种子造成的无关状态搜索和失焦问题, 提升目标状态发现效率. 与 Hawkeye, SCDF 或 RDFuzz 从逼近程度或状态随机触发的难易等评价种子优劣维度改进 AFLGO 的思路不同, 覆盖增益与目标的关联性是评价种子对队列搜索效用贡献优劣的另一个维度. 该维度下需要结合已有的测试队列来判定冗余种子, 并抑制其导致的无关状态搜索效用. 为了聚焦在该维度验证本文降冗策略的有效性, 本文以最基本的方法 AFLGO 作为比较参照, 从而避免引入其他维度上改进策略的干扰.

实验设计时, 一方面要验证方法是否有助于减少无关状态的搜索, 提升目标发现效率; 另一方面, 因为在实际实现时, 切片工具和覆盖统计都无法做到完全准确, 降冗策略可能会放大这些误差, 从而影响多样化执行路径的搜索. 因此, 本文也对 VCov 搜索抵达目标多样化执行路径的能力加以评估. 前者主要从目标状态的首次触发时间来考量, 后者则从抵达目标缺陷状态的执行路径多样性两个角度来考量. 具体来说, 本文选择了错误复现应用场景来进行验证.

- RQ1: VCov 的降冗策略是否提升了目标状态的发现效率?
- RQ2: VCov 的降冗策略是否影响抵达目标执行路径的多样性?

(1) 测试程序和测试目标集

对首次目标执行状态发现时间的测试, 本文选用了与 AFLGO 相同的测试集, 外加 Hawkeye 提供的一些额外的定向测试评估基线. 其中包含 C++源程序标识符到 C++ ABI 标识符的转化工具 cxxfilt、JS 解析器 MJS、正则表达式工具 Oniguruma、图片处理库 Libjpeg-turbo, Libpng 以及字体处理库 Freetype-2.0.

(2) 评估指标设置

为了对比 VCov 与 AFLGO 的性能, 本文沿用了 AFLGO 的测评方法. 采用多次实验的方式获得其某一性能指标的数据. 对于 Google 定向测试套件中的程序(Libjpeg, Libpng, Freetype2)、Binutils、Oniguruma 运行 20 次重复实验. 对 MJS 运行 8 次重复实验(错误复现时间较长). 对同一个性能指标 M, 使用 Vargha-Delaney 统计量[9]: A_{12} 来衡量由 VCov 和 AFLGO 生成的两组数据的优劣, 并采用 Mann-Whitney U 检验[10](显著性阈值 p<0.05)衡量其显著性(在实验结果中相关的 A_{12} 数值会被加粗显示).

(3) 实验环境和实验参数

Ubuntu16.04 3GB 内存的虚拟机, 物理 CPU AMD Ryzen 7 Pro 3700U. 实验参数尽量与 AFLGO 和 Hawkeye 中实验参数保持一致. 对于 Google 测试套件中的测试目标, 将冷却时间设定为 1 min, 其余的尽可能设定为复现时间的 3/4 (对 Binutils 和 MJS 设置为 45 min, 对 Oniguruma 设置为 5 min). 除 Google 测试套件提供了初始种子, MJS 使用测试目录(mjs/test)下的输入作为初始种子外, 其余均使用空初始种子(echo

“·”>in/seed). 目标依赖代码的信息是利用切片工具 DG 对含有调试信息的待测程序 LLVM 中间码分析得到.

为了获取尽可能完整的目标依赖代码, 需要充分利用待复现的缺陷现场中的函数语句和相关变量作为逆向切片分析的起始点, 又称为切片准则(slicing criteria). 本文首先以缺陷的触发语句和引起缺陷的变量为切片准则进行切片, 其次以缺陷现场调用栈上的函数为切片准则进行切片. 通过修改 DG 输出, 将这些切片结果合并, 交给模糊测试器进行有效覆盖统计. 待复现缺陷的现场信息从 Github Issue 网页或者报送漏洞的 Bugzilla 网页上获得. 切片的入口函数越靠近主函数, 切片结果越完整. 因此, 实验中, 切片入口函数尽可能地选择调用栈最底层的函数(最底层是 main 函数). 对于因 DG 自身缺陷无法一次性完成的切片工作, 本文实验沿着目标运行状态的调用栈分段切片, 对获得的基本块取并集. 切片得到的基本块信息存放在对应函数名的文本文件中, 将这些文件所在的文件夹提供给 VCov 进行错误复现和定向测试. 在实验中, 对 VCov 设置的能量缩放因子ζ=2.

(4) 表头符号和含义

a) Target: 以代码行位置或 CVE 编号标识的目标执行状态.

b) Fuzzer: 实验采用的定向模糊测试方法, 主要是 AFLGO 和 VCov.

c) Runs: n (n=8 或 n=20)次重复实验下, 目标执行状态复现次数. 每次实验时间上线为在 4 h.

d) μTTE: 平均首次触发时间, 规定时限内未触发的按照规定时限计算.

e) *Avg* (*Num*): n (n=8 或 n=20)次重复实验的平均触发路径数.

f) Factor: AFLGO 与 VCov 在μTTE 或 *Avg* (*Num*)指标上比值.

g) A_{12}: Vargha-Delaney 统计量[9], 用于表征一组数据比另一组数据更大的程度. 当 Mann-Whitney U 检验判定两组数据差异显著(p<0.05)的情况下, A_{12} 统计值被加粗表示, 其表示 AFLGO 与 VCov 两组实验数据(首次触发时间记录或触发目标的执行路径数记录)的差异较为显著.

(5) 实验结果和分析

表 1 中显示的是在 Google 定向测试评估套件上 VCov 与 AFLGO 在 20 次重复实验中, 对目标状态的平均首次触发时间μTTE 的对比. 初始种子和目标代码行由该套件提供. 通过在目标行插入 *abort*(·)语句来通知模糊测试器该行已触发.

表 1 VCov 和 AFLGO 在 Google 定向测试套件中首次触发时间的统计对比

Target	Fuzzer	Runs	μTTE (s)	Factor	A_{12}
jdmaker.c:659	VCov	20	248	–	–
	AFLGO	20	605	2.43	**0.91**
pngread.c:738	VCov	20	1	–	–
	AFLGO	20	1	1.00	0.50
pngrutil.c:3182	VCov	20	1	–	–
	AFLGO	20	1	1.00	0.50
ttgload.c:1711	VCov	20	524	–	–
	AFLGO	20	1195	1.89	**0.75**

其中, 性能提升比较明显的是 jdmarker.c:659 和 ttgload.c:1711, 它们分别是 Libjpeg-turbo-07-2017 和 Freetype2-2017 中的目标行. jdmarker.c:659 即例 1 中的目标. Fuzzer 一列 VCov 和 AFLGO 分别表示使用本文提出的降冗方法和 AFLGO 原始方法得到的实验结果. 对例 1 的测试结果显示: 有效覆盖的引导策略在依赖简单的分支判断条件的求解中依然适用, 并且提升了 2 倍左右的目标发现效率. 得到该结果的原因主要是降冗策略发挥了作用, 因为仅有个别种子在突破例 1 中的关键路障时能带来有效覆盖增益, 因此只有它们能够获得较高的能量, 其余种子能量则被削减为原先的 1/2. 表 1 中 pngrutil.c:3182 和 pngread.c:738 是两个非常容易通过随机变异触发的目标. 由于触发时间较短, VCov 在切片数据读取和有效覆盖统计上的额外处理使其落后了 300 多毫秒, 不过在以 s 为单位的统计中, VCov 与 AFLGO 并无明显差异.

表 2–表 4 分别显示了 VCov 和 AFLGO 在复现 Binutils, MJS 和 Oniguruma 中的缺陷时, 平均首次触发时间μTTE (s)的对比. MJS 中, 缺陷复现的重复实验次数为 8 次, 其余两个软件为 20 次.

表 2 VCov 和 AFLGO 在 Binutils-2.26 CVE 缺陷复现测试集上的首次触发时间的统计

CVE	Fuzzer	Runs	μTTE (s)	Factor	A_{12}
2016-4487, 2016-4488	VCov	20	62	–	–
	AFLGO	20	230	3.72	**0.90**
2016-4489	VCov	20	1261	–	–
	AFLGO	20	3955	3.14	**0.88**
2016-4490	VCov	20	991	–	–
	AFLGO	20	1861	1.88	**0.92**
2016-4491	VCov	20	1311	–	–
	AFLGO	20	1916	1.46	**0.69**
2016-4492	VCov	20	2024	–	–
	AFLGO	20	2008	1.00	0.60
2016-6131	VCov	20	1518	–	–
	AFLGO	20	2487	1.64	**0.72**

表 3 VCov 和 AFLGO 在 MJS 错误复现测试集上首次触发时间的统计

Issue	Fuzzer	Runs	μTTE(s)	Factor	A_{12}
#57	VCov	1	13182	–	–
	AFLGO	1	13228	1.00	0.51
#69	VCov	6	5273	–	–
	AFLGO	2	13656	2.59	**0.83**
#77	VCov	8	5639	–	–
	AFLGO	6	8727	1.55	0.61
#78	VCov	8	124	–	–
	AFLGO	8	266	2.14	**0.81**

表 4 VCov 和 AFLGO 在 Oniguruma 错误复现测试集上首次触发时间的统计

Issue	Fuzzer	Runs	μTTE (s)	Factor	A_{12}
#56	VCov	20	1	–	–
	AFLGO	20	1	1.00	0.50
#57	VCov	20	11	–	–
	AFLGO	20	17	1.50	0.65
#58	VCov	20	14	–	–
	AFLGO	20	24	1.23	0.69
#59	VCov	20	99	–	–
	AFLGO	20	363	3.67	**0.81**
#61	VCov	20	198	–	–
	AFLGO	20	372	1.88	**0.84**

表 2 中显示了 VCov 和 AFLGO 在 Binutils-2.26 CVE 缺陷复现测试集上的首次触发时间的统计. 实验以空种子作为初始输入, 冷却时间设定为 45 min. 可以看到: 在大多数目标中, VCov 取得了较短的平均首次触发时间. 这是因为被测程序 cxxfilt 对 C++名字的转换过程中需要大量的循环以及递归计算, 与目标无关的执行空间较大. VCov 借助有效覆盖统计, 从而聚焦目标依赖的执行状态的探索. 而 AFLGO 采用全程序覆盖统计, 其容易将时间花费在目标无关的执行状态探索上.

表 3 显示了 VCov 和 AFLGO 在 MJS 错误复现测试集上首次触发时间的统计. 该实验选择空种子作为初始输入, 冷却时间设定为 45 min. 其中, Issue 一列表示待复现的缺陷的 Github Issue 编号, Factor 一列为在 μTTE 上 AFLGO 相比于 VCov 的倍数. #57 为整数溢出, #69 为无效读, #77 为堆缓冲区溢出, #78 为释放后重用. 采用缺陷修复前的一个 commit 版本构建被测程序. 被测程序是根据 mjs.c 编译得到的可执行程序. 在对 MJS 的测试中, 设置时间上限为 4 h. 对每个缺陷运行 8 次重复实验. Runs 这一列表示 8 次重复实验中复现该缺陷的次数. 对超时未发现目标的运行按照最大时间 4h 统计缺陷发现时间. VCov 对#69 和#78 的首次触发时间与 AFLGO 相比有明显的缩短. 但是在整数溢出#57 和堆缓冲区溢出#77 这两个缺陷上, VCov 并未显著优于 AFLGO. 对缺陷#57, 8 次 4 h 重复实验中, 仅有一次触发了目标缺陷. 因此, #57 结果的可比性不强. 而对缺陷 #77, 虽然从复现时间上来看, VCov 与 AFLGO 的差异并不显著, 但是 VCov 在 8 次实验中均复现了该缺陷, 而 AFLGO 仅复现了 6 次.

表 4 中显示了 VCov 在 Oniguruma 错误复现基准测试集上的首次缺陷发现时间的统计结果. 只对 AFLGO 能够在 4h 内复现的缺陷进行了统计. 表 4 中, Issue 列表示缺陷对应的 Github 缺陷号. 选择修复缺陷前的一个 commit 版本作为缺陷版本进行测试. 待测程序从 github 上相应的 Issue 的网页上选取, 将 Oniguruma 程序正则表达式匹配的内容字符串修改为从标准输入读取. 以空种子作为初始输入. 冷却时间设定为 5 min. 实验次数为 20 次. 切片准则从 Github 上报告的缺陷调用栈上的函数调用语句中选择. Oniguruma 的#56(CVE-2017-9225), #57(CVE-2017-9224), #58(CVE-2017-9227)这 3 个栈缓冲区溢出缺陷, VCov 对这 3 个缺陷的首次复现时间优势并不明显. 由于这 3 个缺陷比较容易通过随机交叉变异快速发现, 且平均复现时间在十几秒左右, 交叉变异的随机性对首次触发时间的干扰比较大. 而对于#59(CVE-2017-9229, 无效的指针访问缺陷), #61(释放后重用缺陷), 因为抵达缺陷依赖的中间状态的触发条件较为苛刻, 不容易被随机交叉变异发现, 搜索时间较长, 此间产生的冗余种子较多, 对首次触发时间影响明显. 故而, VCov 在这两个例子上有明显的优势.

为了研究 RQ2, 在 Binutils, MJS 和 Oniguruma 测试集上, 针对每个缺陷对 VCov 和 AFLGO 分别进行了 8×4 h, 8×4 h 和 8×1 h 的测试. 统计给定时限内发现的抵达目标缺陷状态的执行路径数量(AFL 的 Unique Crashes 数量), 见表 5–表 7. 用错误现场, 即 valgrind 运行程序获得的调用栈上的函数和错误信号关键词来验证触发缺陷的输入是否触发了期待的错误状态.

表 5 VCov 和 AFLGO 在 Binutils 错误复现测试集上 4 h 发现的对应缺陷路径数量统计结果

Issue	Fuzzer	Runs	*Avg*(*Num*)	Factor	A_{12}
2016-4487	VCov	8	168	1.52	0.50
2016-4488	AFLGO	8	111	–	–
2016-4489	VCov	8	51	0.57	**0.22**
	AFLGO	8	90	–	–
2016-4490	VCov	8	366	1.09	0.72
	AFLGO	8	335	–	–
2016-4491	VCov	8	7	0.88	0.48
	AFLGO	8	8	–	–
2016-4492	VCov	8	14	1.80	0.60
	AFLGO	8	8	–	–
2016-6131	VCov	8	194	2.00	**0.88**
	AFLGO	8	97	–	–

表 6 VCov 和 AFLGO 在 MJS 错误复现测试集上 4 h 发现的缺陷路径数量统计结果

Issue	Fuzzer	Runs	*Avg*(*Num*)	Factor	A_{12}
#57	VCov	1	0.125	1.00	–
	AFLGO	1	0.125	–	0.50
#69	VCov	6	1	4.00	–
	AFLGO	2	0.25	–	**0.78**
#77	VCov	7	1.875	1.875	–
	AFLGO	4	1	–	0.65
#78	VCov	8	5.125	1.05	–
	AFLGO	8	4.875	–	0.53

表 7 VCov 和 AFLGO 在 Oniguruma 错误复现测试集上 4 h 发现的缺陷路径数量统计结果

Issue	Fuzzer	Runs	*Avg*(*Num*)	Factor	A_{12}
#56	VCov	8	419	1.23	**0.88**
	AFLGO	8	342	–	–
#57	VCov	8	994	1.01	0.50
	AFLGO	8	981	–	–
#58	VCov	8	242	1.07	0.52
	AFLGO	8	226	–	–
#59	VCov	8	268	1.17	0.56
	AFLGO	8	229	–	–
#61	VCov	8	281	1.13	0.60
	AFLGO	8	248	–	–

表 5 显示了对 Binutils-2.26 中, cxxfilt 程序的 4 h 测试发现的目标缺陷路径数量的统计结果. 其中, 对

CVE-2016-4489 和 CVE-2016-4491, VCov 得到的平均缺陷路径数 *Avg*(*Num*)小于 AFLGO. 对于 CVE-2016-4489, 还出现了 VCov 发现的缺陷路径显著少于 AFLGO 的情况. 分析实验数据发现: DG 对这两个缺陷的切片结果明显少于其他缺陷, 导致 VCov 陷入了局部最优. 原因是这两个缺陷现场相关的隐式信息流较多, 另外, DG 切片工具自身有一个无限递归缺陷, 其对这两个测试目标的分析有较大影响. 在切片时, 我们无法指派较高层级的入口函数, 仅仅根据缺陷报告上的函数调用栈进行了分段切片, 这导致获得的有效覆盖信息存在不完整的情况.

表 6 中由于部分缺陷的复现路径数量过少, 采用了小数来表示 *Avg*(*Num*), A_{12} 用于统计两组路径数量数值差异程度, 加粗的 A_{12} 值表示 Mann-Whitney U 检验(阈值 0.05)得到的两组数据差异显著. 从这 3 个测试集的测试结果可以看出, VCov 仅在个别案例上提升了 AFLGO 对多样化执行路径的发现能力. 大部分缺陷复现中, VCov 发现的缺陷执行路径数并不显著地多于 AFLGO. 不过也注意到, VCov 并未显著降低 AFLGO 多样化触发目标执行状态的发现能力.

在 MJS 的 8×4 h 测试中, 因为首次触发时间的提升, 导致其在 8 次实验中的复现次数比 AFLGO 多. 而在其他缺陷复现中, VCov 在平均复现的路径数量上有微弱的优势. 在整理生成的崩溃输入时发现, VCov 发现的与目标缺陷无关的缺陷数量明显少于 AFLGO. 这说明, VCov 的能量调节确实使得定向测试更聚焦了.

根据对指定目标缺陷的发现时间的实验评估, VCov 在多数情况下相比于 AFLGO 都有平均意义下的性能提升. 这说明 VCov 借助有效覆盖增益统计和能量缩放有效促进了对目标相关执行状态的测试聚焦. 本方法在 cxxfilt 这种体量稍大的程序上优势更加明显. 根据表 1–表 4 对首次触发时间的评估, VCov 确实提升了依赖于复杂计算缺陷复现效率(RQ1). 根据对执行路径数量探索能力的评估, VCov 仅在个别案例上显著提升了 AFLGO 在给定时限内发现的缺陷路径数量. VCov 对目标代码的多样化执行路径搜索影响有限(RQ2).

4 相关工作

4.1 全局覆盖率导向的种子度量

通用模糊测试度量种子优劣的主要目的是提升代码覆盖率, 从而发现更多缺陷. 度量信息包含种子自身信息(例如文件大小)以及其触发的执行信息(例如运行时间、控制流覆盖情况、状态变量取值等)等.

Reber 等人[7]对 6 种不同的种子选择策略进行了评估, 其中包括随机选择、固定时间发现的缺陷数, 获得最大覆盖的最小子集、运行时间最小的 Top-*k*、文件大小最小的 Top-*k* 以及 Peach 中基于覆盖的筛选算法. 评估指标包含代码覆盖能力、缺陷发现能力. 在对多媒体文件格式处理程序上的测试显示, 获得最大覆盖的最小子集这种筛选策略是所有策略中最优的. 该文给出了种子集合约简能够显著提升测试性能的确定性结论.

Böhme 等人[11]发现在 AFL 测试过程中, 高频执行路径被过量测试的问题, 其用马尔科夫链模型来分析路径间的迁移关系, 通过调节能量来均衡不同执行路径的测试频次, 减少对高频路径的重复探测. 此后,许多基于随机交叉变异的模糊测试工作都将低频特征作为优质种子的筛选依据[6,12−14]. EcoFuzz[15]则从种子对新路径探索回报的角度对种子的优劣进行评估. 其引入多臂老虎机模型进行最大化回报的分析, 并给出了一种相对节省能量的调度方法.

对于覆盖率引导的模糊测试, 种子是执行路径或执行状态的反映, 也代表着探索新执行状态中间步骤. 细腻的甄别机制能够更准确地分辨不同的执行路径, 从而进行细粒度的状态搜索. CollAFL[16]为了减少 AFL 覆盖统计中的散列哈希冲突, 提高对执行路径的分辨率, 其提出了一种试探性确定基本块随机 ID 和控制流边的散列方法. 除了控制流信息外, 数据取值所表征的比较状态或运算状态也用于甄别有效的种子, 并按照最优化程度度量种子优劣. 例如, Angora[17]基于对分支比较的数据监控和污点分析, 用梯度法求解固定区间数据形成凸函数的比较条件; Steelix[18]对字符串的比较状态按匹配度进行了划分, 识别推进字符串比较进度的种子; GreyOne[19]则借助按位匹配度来确定推进一般分支条件比较进展; IJON[20]提供的 IJON_MAX 和 IJON_MIN 最优化原语对种子的优先级进行了划分, IJON 使模糊测试人员可以根据源代码中的变量定制细粒度的状态反馈.

上述提升的种子度量和分级划分策略是面向全程序覆盖设计, 这与固定代码导向的定向测试的应用场景有所差异. 因此, 本文与上述方法不同, 需要对代码关联的执行状态进行有偏的划分. 虽然 IJON 也可以实现这一点, 但是需要手工分析确定目标依赖的关键中间代码和中间状态. 程序切片恰好能实现对目标依赖代码的自动识别. 上述细粒度的种子划分和评级方法对于固定代码导向的定向测试依然有效, 但是需要结合有效覆盖分析才能对目标相关状态的针对性搜索, 否则, 这些方法仍然会产生大量的冗余种子, 导致目标无关执行状态的盲目搜索和测试失焦. 本文的研究着力于此点, 通过切片来识别有效覆盖代码区, 缓解冗余种子引发的副作用.

4.2 目标代码导向的种子度量

补丁/变更测试、缺陷验证和复现等应用中, 需要对代码位置确定的目标执行状态生成触发输入. 早期解决该类问题的方法主要采用基于约束的测试输入生成技术[21], 2017 年, Böhme 等人在解决该类问题时提出了一种基于灰盒模糊测试框架的方案 AFLGO[1], 并提出了定向灰盒模糊测试(directed grey-box fuzzing)的概念. 该方案的主要思想是: 度量种子触发路径与目标的远近, 为与目标更近的种子提供更多繁衍后代的机会. AFLGO 借助退火算法来实现渐进式地能量调节以完成这种聚焦. 文献[1]中的实验结果显示, 其效率超越了当时较为领先定向符号执行方案 KATCH[22]和 Bugredux[23].

AFLGO 之后出现了一些对度量种子效用的改进或创新工作, 例如, 陈泓旭等人在其工作 Hawkeye[2]中指出: 目标代码导向定向模糊测试应该搜索所有到达目标点的执行路径, 而不只是偏向于最短路径. 其对 AFLGO 的距离公式进行了改进, 以减少对最短路径的偏好. 为了避免图上最短路径的耗时的静态分析和计算, 张旖旎等人在他们的工作 SCDF[3,4]中提出了一种基于序列覆盖率度量种子与目标的逼近程度的方法. 该方法仅需要在关键语句或函数序列处进行插装, 然后运行时获得序列覆盖率, 避免了复杂的静态分析. 针对运行开销很大的程序的长时间定向测试场景, 宗珮媛等人在其工作 FuzzGuard[5]中, 利用已有的目标代码触发记录, 借助深度学习提取种子的内容特征来预测输入的目标可达性, 进而提前过滤掉随机变异生成的明显目标不可达种子, 提升定向测试效率. 为了避免退火算冷却时间设置不当引发的性能退化, 提高可用性, RDFuzz[6]分开度量种子逼近目标的程度和对多样化路径的搜索能力, 对种子综合打分, 确定能量. 其采用分支频率来度量种子对多样化执行路径的搜索能力.

上述方法评估并发挥种子对目标相关状态的搜索作用以促进测试聚焦. 但是全局覆盖统计缺乏对目标关联分析, 导致目标无关状态的搜索. 全局覆盖统计下, 种子除了有探索目标相关执行状态的能力外, 还有搜索目标无关执行状态的能力. 相比于度量和遴选对目标相关状态搜索能力强的种子, 抑制目标无关状态的搜索是提升定向灰盒模糊测试的专注度、促进测试聚焦的另一个研究维度. 另外, 逼近程度和状态触发难易等特征仅孤立地分析了种子个体的状态搜索效用, 而本文提出的有效覆盖增益特征则关心种子队列的整体效用. 无关状态搜索问题就是冗余种子对整体种子队列搜索效用的一种副作用. 本文提出使用有效覆盖统计来改善全局覆盖统计造成的这种副作用. 为了避免运行结果对程序切片的准确性和完整性过于敏感, 本文采用能量缩放的方式而不是直接删掉冗余种子.

Wustholz 等人[14]在对智能合约定向测试时, 采用非目标导向前缀(no-target-ahead prefix)来去除目标不可达的种子. 原理是, 非目标导向前缀之下的所有后缀执行路径都无法到达目标. 但是目标不可达种子也可能对目标状态计算的中间步骤探索有贡献. 该方法并未考虑这一点. 另外, 采用在线的静态分析(抽象解释)的方式识别和记录执行前缀的性能开销是可观的, 因此其仅适用于智能合约等逻辑较为简单程序的定向测试. 相比之下, 本文的有效覆盖统计方法能够重用对目标依赖的中间计算状态有新发现的种子. 采用离线分析和编译时插装, 运行速度更快, 尤其适合较大程序中仅有一小部分代码与目标有关的场景.

Peng 等人[24]根据二进制程序补丁寻找 1day 漏洞. 为了避免定向灰盒模糊测试阻塞于字符串比较等形式简单但不容易被随机测试发现的分支条件, 其识别和定位最近目标可达的主导分支条件, 并借助符号执行快速求解. 主导分支条件仅提供了目标控制依赖信息. 数据流与目标无关的代码仍然会被符号执行探索. 而且, 抵达阻塞时限前, 定向灰盒模糊测试本身仍存在冗余种子的问题. 本文使用的逆向程序切片技术能够同时识

别出目标的控制和数据依赖, 可同时减少数据依赖无关执行空间的搜索. 而且本文基于有效覆盖统计的能量调节保留了灰盒模糊测试可扩展性优势, 不会因静态分析技术的某些细节而受到较大的影响. VCov 也很容易与快速求解简单分支条件的方法[12,17−19,25,26]结合, 弥补随机交叉变异的不足.

4.3 缺陷特征导向的种子度量

面向特定类型漏洞挖掘的模糊测试工作依据漏洞的特征对种子触发漏洞的能力进行度量. 针对释放后重用缺陷 UFuzz[26], UAFuzz[27]等模糊测试方法构造包含缺陷状态的状态机, 根据种子触发的状态与目标状态相距的跳数度量种子优劣. 针对内存耗尽型的缺陷, Memlock[28]利用种子触发的程序执行的内存消耗特征度量种子优劣. 针对性能缺陷, 增加控制流执行频次、执行路径长度等 CPU 消耗特征度量种子的优劣[29]. 针对内存破坏型漏洞的研究, 则利用种子触发的内存访问代码信息[30,31]和一些高概率形成漏洞的 API[32]或具有较高的复杂性的代码[33]度量种子优劣.

这些方法仅仅孤立地度量种子自身, 缺乏对种子队列整体作用的分析. 而且相关的方法并未提供抵达目标关键操作前的中间状态引导策略, 引导策略粒度较粗. 而切片得到的有效覆盖代码区能够捕获更多的中间状态变化, 提供有效的补充. 在面向特定漏洞挖掘的应用场景下, 冗余种子带来的不相关状态搜索问题的严重性仅与缺陷状态依赖的代码占比有关. 当潜在缺陷代码位置点较多时, 大部分代码都与缺陷有关, 此时冗余种子的影响并不明显. 但是如果潜在缺陷点较少, 本文提出的降冗策略就能带来缺陷状态发现效率的明显提升.

5 结 论

本文指出: 在定向灰盒模糊测试中, 新种子引发的覆盖增益是否影响目标执行状态的计算, 是评判新种子对当前队列搜索效用增益的一个重要指标. 带来的覆盖增益与目标执行状态无关的冗余种子不仅无异于提升种子队列对目标执行状态的搜索能力, 反而会诱导定向模糊测试搜索目标无关的执行状态. 现有的方法在度量种子有效性时, 缺乏对该因素的考量, 使用全局覆盖统计. 这阻碍了定向测试的聚焦, 影响目标执行状态的发现效率. 为了解决该问题, 本文提出了一种根据目标依赖的代码覆盖变化来判定种子对于状态探索作用优劣的方法. 利用程序切片技术来确定影响目标状态计算的代码, 在覆盖统计时, 识别覆盖增益位于这些代码的种子, 并从能量分配的角度增加这些有效覆盖增益种子的后代生成数量, 减少其他冗余种子的后代生成数量. 实验结果表明: 我们的方法有效削弱了冗余种子的影响, 提升了目标执行状态的发现效率.

References:

[1] Böhme M, Pham VT, Nguyen MD, Roychoudhury A. Directed grey-box fuzzing. In: Thuraisingham B, ed. Proc. of the ACM SIGSAC Conf. on Computer and Communications Security (CCS 2017). New York: ACM, 2017. 2329−2344. [doi: 10.1145/3133956.3134020]

[2] Chen HX, Xue YX, Li YK, Chen BH, Xie XF, Wu XH, Liu Y. Hawkeye: Towards a desired directed grey-box fuzzer. In: Lie D, Mannan M, eds. Proc. of the ACM SIGSAC Conf. on Computer and Communications Security (CCS 2018). New York: ACM, 2018. 2095−2108. [doi: 10.1145/3243 734.3243849]

[3] Liang HL, Zhang YN, Yu Y, Xie ZS, Jiang L. Sequence coverage directed greybox fuzzing. In: Guerrero J, ed. Proc. of the IEEE/ACM 27th Int'l Conf. on Program Comprehension (ICPC 2019). New York: ACM, 2019. 249−259. [doi: 10.1109/ICPC.2019.00044]

[4] Zhang YN. Research and implementation of defects detection system based on directed fuzzing technology [MS. Thesis]. Beijing: Beijing University of Posts and Telecommunications, 2019 (in Chinese with English abstract).

[5] Zong PY, Lv T, Wang DV, Deng ZZ, Liang RG, Chen K. FuzzGuard: Filtering out unreachable inputs in directed grey-box fuzzing through deep learning. In: Capkun S, Roesner F, eds. Proc. of the 29th USENIX Security Symp. (USENIX Security 2020). Berkeley: USENIX Association, 2020. 2255−2269.

[6] Ye J, Li R, Zhang B. RDFuzz: Accelerating directed fuzzing with intertwined schedule and optimized mutation. In: Proc. of the Mathematical Problems in Engineering 2020. London: Hindawi, 2020. Article ID 7698916. [doi: 10.1155/2020/7698916]

[7] Rebert A, Cha SK, Avgerinos T, Foote J, Warren D, Grieco G, Brumley D. Optimizing seed selection for fuzzing. In: Fu K, Jung J, eds. Proc. of the 23rd USENIX Security Symp. (USENIX Security 2014). Berkeley: USENIX Association, 2014. 861−875. [doi: 10.5555/2671225]

[8] Chalupa M. Slicing of LLVM bitcode [MS. Thesis]. Brno: Masaryk University, 2016.

[9] Vargha A, Delaney HD. A critique and improvement of the “CL” common language effect size statistics of McGraw and Wong. Journal of Educational and Behavioral Statistics. 2000, 25(2): 101−132. [doi: 10.3102/10769986025002101]

[10] Mann HB, Whitney DR. On a test of whether one of two random variables is stochastically larger than the other. Annals of Mathematical Statistics, 1947, 18(1): 50−60. [doi: 10.1214/aoms/1177730491]

[11] Böhme M, Pham VT, Roychoudhury A. Coverage-based greybox fuzzing as Markov chain. In: Weippl E, Katzenbeisser S, eds. Proc. of the ACM SIGSAC Conf. on Computer and Communications. New York: ACM, 2016. 1032−1043. [doi: 10.1145/2976749.2978428]

[12] 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. San Diego: Internet Society, 2017. 1−14.

[13] Lemieux C, Sen K. Fairfuzz: A targeted mutation strategy for increasing greybox fuzz testing coverage. In: Huchard M, Kästner C, Fraser G, eds. Proc. of the 33rd ACM/IEEE Int’l Conf. on Automated Software Engineering. New York: ACM, 2018. 475−485. [doi: 10.1145/3238147.3238176]

[14] Wüstholz V, Christakis M. Targeted grey-box fuzzing with static lookahead analysis. In: Rothermel G, Bae DH, eds. Proc. of the 42nd ACM/IEEE Int’l Conf. on Software Engineering. New York: ACM, 2020. 789−800. [doi: 10.1145/337781 1.3380388]

[15] Yue T, Wang P, Tang Y, Yu B, Lu K, Zhou X . EcoFuzz: Adaptive energy-saving greybox fuzzing as a variant of the adversarial multi-armed bandit. In: Capkun S, Roesner F, eds. Proc. of the 29th USENIX Security Symp. (USENIX Security 2020). Berkeley: USENIX Association, 2020. 2307−2324.

[16] Gan S, Zhang C, Qin X, Tu XW, Pei ZY, Chen ZN. CollAFL: Path sensitive fuzzing. In: Li JH, ed. Proc. of the IEEE Symp. on Security and Privacy (S&P). Piscataway: IEEE, 2018. 679−696. [doi: 10.1109/SP.2018.00040]

[17] Chen P, Chen H. Angora: Efficient fuzzing by principled search. In: O’Conner L, ed. Proc. of the IEEE Symp. on Security and Privacy (S&P 2018). Piscataway: IEEE, 2018. 711−725. [doi: 10.1109/SP.2018.00046]

[18] Li YK, Chen B, Chandramohan M, Lin SW, Liu Y, Tiu A. Steelix: Program-state based binary fuzzing. In: Bodden E, Schäfer W, Deursen AV, Zisman A, eds. Proc. of the 11th Joint Meeting on Foundations of Software Engineering. New York: ACM, 2017. 627−637. [doi: 10.1145/3106237.3106295]

[19] Gan S, Zhang C, Chen P, Zhao BD, Qin XJ, Wu D, Chen ZN. GREYONE: Data flow sensitive fuzzing. In: Capkun S, Roesner F, eds. Proc. of the 29th USENIX Security Symp. (USENIX Security 2020). Berkeley: USENIX Association, 2020. 2577−2594.

[20] Aschermann C, Schumilo S, Abbasi A, Holz T. IJON: Exploring deep state spaces via fuzzing. In: Kellenberger P, ed. Proc. of the IEEE Symp. on Security and Privacy (S&P 2020). Piscataway: IEEE, 2020. 1597−1612. [doi: 10.1109/SP40000.2020.00117]

[21] Yang K, He YP, Ma HT, Wang XF. Precise execution reachability analysis. Ruan Jian Xue Bao/Journal of Software, 2018, 29(1): 1−22 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/5375.htm [doi: 10.13328/j.cnki.jos.005375]

[22] Marinescu PD, Cadar C. KATCH: High-coverage testing of software patches. In: Meyer B, Baresi L, Mezini M, eds. Proc. of the 9th Joint Meeting on Foundations of Software Engineering. New York: ACM, 2013. 235−245. [doi: 10.1145/2491411. 2491438]

[23] Jin W, Orso A. BugRedux: Reproducing field failures for in-house debugging. In: Glinz M, Murphy G, Pezzè M, eds. Proc. of the 34th Int’l Conf. on Software Engineering. Piscataway: IEEE, 2012. 474−484. [doi: 10.1109/ICSE.2012.6227168]

[24] Peng J, Li F, Liu BC, Xu LL, Liu BH, Chen K, Huo W. 1dVul: Discovering 1-day vulnerabilities through binary patches. In: O’Conner L, ed. Proc. of the 49th Annual IEEE/IFIP Int’l Conf. on Dependable Systems and Networks (DSN). Piscataway: IEEE, 2019. 605−616. [doi: 10.1109/DSN.2019. 00066]

[25] You W, Wang X, Ma S, Zhang XY, Wang XF. ProFuzzer: On-the-fly input type probing for better zero-day vulnerability discovery. In: Kellenberger P, ed. Proc. of the IEEE Symp. on Security and Privacy (S&P). Piscataway: IEEE, 2019. 769−786. [doi: 10.1109/SP.2019.00057]

[26] Wang H, Xie X, Li Y, Qin SB, Wen C, Li YK, Liu Y, Chen HX, Sui YL. Typestate-guided fuzzer for discovering use-after-free vulnerabilities. In: O'Conner L, ed. Proc. of the 42nd Int'l Conf. on Software Engineering. New York: ACM, 2020. 999−1010. [doi: 10.1145/3377811.3380386]

[27] Nguyen MD, Bardin S, Bonichon R, Groz R, Lemerre M. Binary-level directed fuzzing for use-after-free vulnerabilities. In: Proc. of the 23rd Int'l Symp. on Research in Attacks, Intrusions and Defenses (RAID 2020). Berkeley: USENIX Association, 2020. 47−62.

[28] Wen C, Wang H, Li YK, Qin SC, Liu Y, Xu ZW, Chen HX, Xie XF, Pu GG, Liu T. Memlock: Memory usage guided fuzzing. In: O'Conner L, ed. Proc. of the 42nd Int'l Conf. on Software Engineering. New York: ACM, 2020. 765−777. [doi: 10.1145/3377811.3380396]

[29] Lemieux C, Padhye R, Sen K, Song D. PerfFuzz: Automatically generating pathological inputs. In: Proc. of the 27th ACM SIGSOFT Int'l Symp. on Software Testing and Analysis. New York: ACM, 2018. 254−265. [doi: 10.1145/3213846.3213874]

[30] Coppik N, Schwahn O, Suri N. MemFuzz: Using memory accesses to guide fuzzing. In: Proc. of the 12th IEEE Conf. on Software Testing, Validation and Verification (ICST 2019). Piscataway: IEEE, 2019. 48−58. [doi: 10.1109/ICST.2019.00015]

[31] Haller I, Slowinska A, Neugschwandtner M, Bos H. Dowsing for overflows: A guided fuzzer to find buffer boundary violations. In: King S, ed. Proc. of the 22nd USENIX Security Symp. Berkeley: USENIX Association, 2013. 49−64.

[32] Wang YH , Jia XK , Liu YW, Zeng K, Bao T, Wu DH, Su PR. Not all coverage measurements are equal: fuzzing by coverage accounting for input prioritization. In: Proc. of the Network and Distributed System Security Symp. San Diego: Internet Society, 2020. [doi:10.14722/ndss.2020.24422]

[33] Du X, Chen B, Li Y, Guo J, Zhou Y, Liu Y, Jiang Y. Leopard: Identifying vulnerable code for vulnerability assessment through program metrics. In: Proc. of the 41st Int'l Conf. on Software Engineering. IEEE, 2019. 60−71. [doi: 10.1109/ICSE.2019.00024]

附中文参考文献:

[4] 张旖旋. 基于定向模糊测试技术的缺陷检测系统研究与实现 [硕士学位论文]. 北京: 北京邮电大学, 2019.

[21] 杨克, 贺也平, 马恒太, 王雪飞. 精准执行可达性分析: 理论与应用. 软件学报, 2018, 29(1): 1−22. http://www.jos.org.cn/1000-9825/5375.htm [doi: 10.13328/j.cnki.jos.005375]

杨克(1989－), 男, 博士, 主要研究领域为软件安全分析, 操作系统安全.

蔡春芳(1996－), 女, 硕士, 主要研究领域为软件安全分析.

贺也平(1962－), 男, 博士, 研究员, 博士生导师, 主要研究领域为系统安全, 隐私保护.

谢异(1995－), 男, 硕士, 主要研究领域为软件安全分析, 操作系统安全.

马恒太(1970－), 男, 博士, 副研究员, 主要研究领域为软件安全分析, 操作系统安全.

董柯(1996－), 男, 硕士, 主要研究领域为软件安全分析, 操作系统安全.