计算机系统应用 ISSN 1003-3254, CODEN CSAOBN
Computer Systems & Applications,2019,28(1):17−24 [doi: 10.15888/j.cnki.csa.006714]

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

基于覆盖频率的模糊测试改进方法①

傅　玉[1], 石东辉[2], 张　阳[1], 程　亮[1]

[1](中国科学院 软件研究所 可信计算与信息保障实验室, 北京 100190)
[2](深圳大学 深圳南特商学院, 深圳 518060)
通讯作者: 傅　玉, E-mail: fuyu_iscas@outlook.com

摘　要: 模糊测试是当前检测程序错误的最主流、最有效的手段之一. 模糊测试工具首先对种子文件进行变异, 生成大量新输入文件, 然后挑选新输入来执行目标程序, 以触发程序中潜在的漏洞. 当前对模糊测试的研究多着眼于改进变异算法, 提高生成的新文件对目标程序代码的覆盖, 忽略了备用种子文件的筛选策略对提高模糊测试覆盖率与测试效率的的贡献. 针对该问题, 我们提出了基于覆盖频率的种子文件筛选策略, 在每次执行目标程序时, 我们记录程序执行中覆盖过的路径边; 根据边被执行次数的多少, 我们将这些边分为低频边和高频边; 对于包含了更多低频边且执行效率高的种子文件, 我们给予更高的优先级. 我们在模糊测试工具 American Fuzzy Lop (AFL) 实现了对应的算法, 实验表明我们的算法有效提高了模糊测试的效率和代码覆盖率.

关键词: 自动测试; 模糊测试; 漏洞检测

引用格式: 傅玉,石东辉,张阳,程亮.基于覆盖频率的模糊测试改进方法.计算机系统应用,2019,28(1):17–24. http://www.c-s-a.org.cn/1003-3254/6714.html

Improved Fuzz Testing Approach Based on Coverage Frequency

FU Yu[1], SHI Dong-Hui[2], ZHANG Yang[1], CHENG Liang[1]

[1](Trusted Computing and Information Assurance Laboratory, Institute of Software, Chinese Academy of Sciences, Beijing 100190, China)
[2](Shenzhen Audencia Business School, Shenzhen University, Shenzhen 518060, China)

Abstract: In recent years, fuzz testing has become one of the most popular and efficient methods to detect program bugs and vulnerabilities. By mutating seed files, fuzzing tools generate a large volume of test inputs, and feed them to the program under test in order to expose security weakness. Current researches mostly focus on improving the mutation algorithm to make newly generated files cover more target program codes. However, little attention had been paid to elaborately optimizing the policies to sort seed files to be fuzzed, which prioritizes the seed files with higher probability to cover new program spaces in the fuzzing process, and consequently improves the efficiency of fuzzing. Therefore, we proposed a coverage frequency based selection approach to guide the fuzzer to execute promising seed files first. To do so, we first kept tracking how many times each edge between two basic blocks has been executed by the target program in the fuzz testing. Based on the number that each edge has been executed, we then categorized them into high-frequency edges and low-frequency edges. Only seeds containing more low-frequency edges, as well as being executed very fast by the target program, were assigned with high priority. We implemented our method on American Fuzzy Lop (AFL), one of the most popular fuzzers and applied the modified AFL version to 5 real world programs. The result shows that our approach can improve both the efficiency of AFL and the code coverage explored in the target program.

Key words: automated testing; fuzz testing; vulnerability detection

① 基金项目: 国家自然科学基金 (61471344)
Foundation item: National Natural Science Foundation of China (61471344)
收稿时间: 2018-06-20; 修改时间: 2018-07-12; 采用时间: 2018-07-25; csa 在线出版时间: 2018-12-07

1 引言

自 20 世纪 90 年代被提出以来[1], 模糊测试逐渐成为了检测程序错误最有效的方法之一, 被微软、谷歌等主流软件厂商广泛应用于软件开发流程中, 以保障软件产品的安全性和可用性, 提高软件质量. 模糊测试是一种通过生成或变异, 自动构造海量随机输入来运行并检测目标程序性质的自动化检测技术. 目标程序接受构造的数据作为输入并在受监控的环境下执行, 若出现程序崩溃或断言失败等异常状况, 再由测试人员定位错误位置并分析错误原因, 从而达到排除错误, 提高软件质量的效果. 相对于模型验证、数据流分析等其他检测技术, 模糊测试拥有误报率低、部署简单等优点.

根据不同的角度, 当前的模糊测试工具有两种分类法. 首先, 按照构造新输入的方式不同, 模糊测试可分为基于生成 (generation-based) 和基于变异 (mutation-based) 两种. 基于生成的模糊测试[2–4]要求提供目标程序输入文件的格式信息或相关语法, 测试工具根据这些知识自动生成相应的输入文件来执行程序. 该方法的优势是生成的新文件符合目标程序对输入文件格式的基本要求, 能够通过一些复杂的文件格式检测, 在运行时能有效的覆盖目标程序的主要代码路径片段. 但这种方法可复用性较低, 对不同类别的程序需要编写不同的代码来描述输入文件的格式与规则. 因此常用于检测一些处理高度结构化文件的程序, 如 HTML、Java Script 这类文件的解释器. 基于变异的模糊测试无需了解输入文件的语法格式, 测试工具通过比特位翻转、特殊值替换、增减文件片段等手段来随机轻微改变已有的种子文件生成新输入. 基于变异的方法适合于构建结构简单的文件 (如图片文件), 部署简单, 可扩展性高, 在实际应用中被广泛采用.

其次, 根据模糊测试过程中对程序内部结构的利用程度, 可分为黑盒模糊测试、白盒模糊测试和灰盒模糊测试. 黑盒模糊测试[5]不对目标程序的代码和运行状态做任何分析和检测, 根据预设脚本和程序输出构造下一个输入, 速度快但是构造出的文件质量不如后两者. 白盒模糊测试[6–9]在使用静态分析技术获取目标程序的信息, 通过污点分析、符号执行等技术指导输入的构造, 使每次执行尽可能的到达危险或未经探索的代码片段. 白盒测试构造的输入质量优秀, 但由于路径爆炸和约束求解效率等问题, 构造输入的效率不高, 可扩展性较差. 灰盒模糊测试[10–13]采取了折中的手段, 对程序进行轻量级的分析以获取程序运行状态信息, 根据这些信息对每次执行的结果进行评估, 制定后续的测试计划. 相对于黑盒测试, 灰盒测试提高了输入的质量; 只使用轻量级的分析, 保证了模糊测试的执行速度, 是目前运用最为广泛的方法.

本文着重于对基于变异的灰盒模糊测试进行研究, 其中基于覆盖率的方法最为成功. 基于覆盖率的方法记录执行目标程序过程中, 不同输入对应的不同程序路径. 当一个输入覆盖了一段新的程序代码时, 测试工具认为这是一个有价值的输入, 将其放入种子列表用于后续测试; 若输入未能覆盖新代码, 测试工具认为它无价值并舍弃. 这种近似遗传算法的手段, 使得以这些种子为母版的新输入更容易发现新的程序路径. American Fuzzy Lop (AFL)[10]为此类模糊测试的代表, 也因此成为当今最为流行的模糊测试工具之一.

尽管这类方法取得了很好的效果, 但是仍存在可改进的地方. 根据我们的观察, 目前的模糊测试工具存在冗余测试的问题, 部分程序路径被大量重复测试, 而一些罕见路径则缺乏关注. 这是由于测试工具对种子文件缺少筛选导致的, 部分种子文件可能经过更多罕见边, 却以相同的频率被测试, 导致这些罕见边被执行的次数很少, 难以触发其中的程序错误. 一些相关研究已开始着手解决这类问题. 如 AFLFast[13]使用马尔可夫链模型来预估测试一个种子文件所需要的次数以减少不必要额外测试.

针对上面的问题, 我们提出了基于覆盖频率的种子文件筛选策略. 在每次执行目标程序时, 我们记录哪些程序边被覆盖, 并统计它们在所有执行中被覆盖的次数. 当完成了一个种子文件的测试后, 准备选择下一个目标时, 我们将当前所有被覆盖的边按照出现的频率分为高频边和低频边. 对于包含更多低频边且执行速度快的种子文件, 我们给其更高的执行优先级以及更多的测试次数. 我们在 AFL 上实现了我们的算法, 完成了原型工具 MuFuzzer, 并在 Linux 环境下对多个常用实际程序进行了实验. 实验结果表明相对于 AFL, 我们的工具 MuFuzzer 能够更快的提升对程序代码的覆盖, 在执行 24 小时之后, 对目标程序的覆盖率有平均 8.55% 的提升, 证明了我们的方法是有效的.

2 背景

本节首先介绍模糊测试的基本工作流程; 由于我们的方法是在 AFL 上开发的, 第二部分我们将介绍

AFL 的工作原理等相关背景.

2.1 基于覆盖率的模糊测试

基于覆盖率的模糊测试工具的基本流程如图 1 所示. 左侧外围的是待测试的目标程序与对应的初始输入, 中间的模糊测试工具接受两者并进行长时间的循环测试, 最终在右侧输出发现的漏洞信息.

在模糊测试开始前, 首先需要对目标程序进行插桩, 以获取程序在运行时的信息. 目标程序可能是以二进制可执行文件或源文件的格式输入, 这需要模糊测试工具有在二进制上或在编译时插桩的能力.

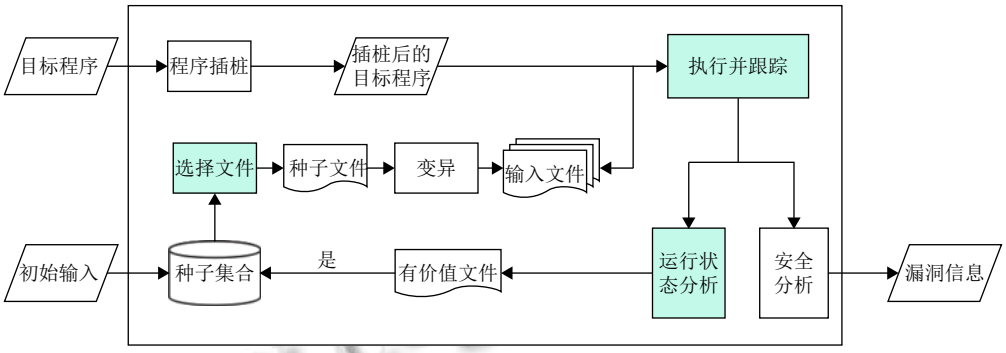

图 1 基于覆盖率的模糊测试工具的基本框架

右侧提供的初始输入应当为目标程序能处理的文件类型, 例如一个专职处理 png 文件的程序, 应选择 png 类型文件作为初始输入, 而不是 pdf 之类的文件. 好的初始输入能覆盖目标程序的大部分代码, 节约了模糊测试工具探索程序的时间. 若两个文件能覆盖程序的不同功能代码, 将两个文件都加入初始输入能提高测试的效率.

模糊测试开始时, 测试工具的种子序列中只有初始输入. 测试工具按照一定的规则, 从种子序列中选出一个用于变异的种子文件. 被选出的种子文件再按照给定的变异算法进行变异, 生成大量新的输入文件. 模糊测试工具执行目标程序, 并跟踪目标程序的运行状态. 当程序崩溃或断言失败时, 模糊测试工具记录这个错误, 输出相关包括该输入文件在内的漏洞信息. 若程序正常退出, 模糊工具通过执行时记录的状态信息, 例如覆盖率、执行时间等, 来判断当前输入是否有价值. 如果有价值, 则放入种子队列, 否则舍弃. 当完成对一个种子文件的测试之后, 再从种子序列中选出下一个文件进行测试. 模糊测试工具按照这个流程循环执行, 直到规定的条件达到或用户选择终止.

我们的工作主要在选择文件、执行与跟踪、以及执行状态分析这三个功能模块上 (图 1 中带阴影的三个模块) 对现有工具 AFL 进行了改进.

2.2 AFL 介绍

在基于覆盖率的灰盒模糊测试工具中, AFL 是最为优秀的代表. 它的出色点主要在于首先它使用了一系列系统底层的编程技巧, 极大的提高了程序执行效率, 单线程环境下每秒测试目标程序可达到惊人的数千次, 大大超过了同类工具. 其次, AFL 以基本块的转移作为覆盖信息的基本单位, 相对于以基本块作为基本单位, AFL 的记录了更为细致的信息, 更容易发现新覆盖并触发漏洞. 最后, AFL 在高效、有效的前提下, 简单的配置和使用方法也促进了 AFL 的传播和发展, 数百个软件漏洞经由该工具发现[14].

AFL 在编译目标程序时, 给每个基本块添加一个 (0, 65 535) 之间的随机数作为该基本块的标识值. 当程序从上一个基本块执行到当前基本块时, 利用这两个基本块的标识值计算出一个新的哈希值, 用这个值来代表这两个基本块的之间的边, 具体使用方法如下:

```
cur_location = <COMPILE_TIME_RANDOM>;
shared_mem[cur_location ^ prev_location]++;
prev_location = cur_location >> 1;
```

其中 cur_location 和 prev_location 分别表示当前基本块和上一个基本块的标识值, 它们俩异或的结果就是当前边的哈希值. AFL 在计算当前边的哈希值之前, 会对前一个基本块的标识值做右移一位的处理, 如此便能区别两个基本块的执行顺序. shared_mem 是一个 64 KB 长度的 8 位数数组, 由测试工具和目标程序共享, 用来记录一次执行中每条边的执行次数. AFL 只需要将 shared_mem 和之前的记录做对比, 就能知道这次执

行是否有覆盖新的边; 若有新的覆盖, 该输入即为一个种子文件.

在每次发现新的种子文件后, AFL 会尝试在不改变覆盖范围的前提下, 尽可能的减少种子文件的长度, 加快执行的速度. 越短的文件意味着越少的测试次数, 同时执行速度越快意味着完成对该种子文件测试的时间越短. 当加入新种子文件后, AFL 会在保持总的覆盖率不变的情况下, 剔除执行速度慢的老的种子文件.

AFL 不断从种子序列中按照概率选出文件进行测试, 其中未被测试过的种子文件优先级最高. 当输入触发了程序崩溃后, AFL 保存该输入, 并按照覆盖信息判断是否这是否是一个新的崩溃.

3 基于覆盖频率的筛选策略

本章介绍我们提出的覆盖频率的种子文件筛选策略. 首先我们将介绍测试的整体流程, 然后我们将详细讲解我们获取覆盖频率信息的方法, 最后介绍具体的筛选策略.

3.1 总体流程

AFL 在选择下一个作为变异母版的种子文件时, 会优先选取未测试过的文件, 这类文件比已测试过的种子文件更容易发现新的程序覆盖. 但这种的方法忽视了不同文件间的区别. 覆盖了更多低频路径片段的文件与其他文件以相同的频率被选取, 导致这些低频路径片段在实际测试过程中更少被执行到. 被执行的次数少, 意味着这些路径片段上的可能存在的错误更不易被触发, 更难以被发现.

针对 AFL 这类模糊测试工具存在的这一问题, 我们提出了基于覆盖频率的排序算法. 当 AFL 从队列中选取新文件时, 我们将种子文件包含的低频边数量纳入参考, 修改后的 AFL 算法框架如算法 1.

算法 1. 基于频率的模糊测试流程

```
1   S = Ø;
2   load S with the user provided test input;
3   SU = Ø;
4   repeat until terminated:
5     s = pick_next(S);
6     move s from S to SU
7     repeat n times:
8       s′ = mutate(s);
9       execute(s′);
10      update_frequency(s′);
11      if has_new_coverage(s′) then
12        S = S U s′
13        update_coverage();
14    score_frequency(S);
15    if S == Ø then
16      score_frequency(SU);
17      replace S with SU
18      SU = Ø and goto line 4
```

算法 1 的流程中, 高亮部分是我们对 AFL 修改的部分.

队列 S 为种子序列, 初始输入由用户提供 (1、2 行). SU 为备用队列, 用于储存已测试过的种子文件, 初始值为空 (第 3 行). 模糊测试一个不间断的过程, 在用户选择停止之前, 模糊测试将一直进行 (第 4 行). AFL 从队列 S 中取出用于变异的种子文件 s, 将其标记为已测试以防重复测试 (5、6 行). 再通过变异算法, 重复生成多个新的输入文件 (第 7、8 行). 每生成一个新的文件 s′, AFL 以 s′为输入并执行目标程序 (8、9 行). 若 s'覆盖了新的程序路径, 则将 s′加入队列种子序列 S, 并更新覆盖率信息; 否则就舍弃该文件 (11~13 行). 当完成了对种子序列中所有文件测试后 (第 15 行), AFL 用队列 SU 替换 S 继续对队列中的所有文件进行测试.

我们在 AFL 的基础上添加了更新覆盖频率信息 (第 10 行) 和依据信息判定种子文件价值的功能 (第 14、16 行), 并修改了 AFL 选择下一个种子文件 (第 5 行) 的功能, 使得覆盖更多低频边的执行速度快的种子文件能更优先被测试, 实现了我们的算法.

3.2 获取覆盖频率信息

在依照覆盖频率信息筛选种子文件前, 我们首先需要获取覆盖频率信息. 在每次执行程序时, 我们以一定的概率记录这次执行的覆盖信息; 在完成对一个种子文件的测试、准备选择下一个目标前, 我们依据记录的覆盖频率, 划分高频、低频边为选择下个文件提供信息.

在 2.2 节, 我们提到了 AFL 使用 shared_mem 数组来存储当次执行时的信息, 并依照该信息判断输入是否覆盖了新的边. 在 AFL 中, 有一个 64 KB 大小的全局数组 virgin_bits 来储存在整个测试过程中所有执行曾经覆盖过的边的信息. virgin_bits 上的某个成员为非空, 既表示这个元素对应的边曾经被执行到过; 反之为空标志对应边还未被探索. AFL 只需对照 shared_mem 和 virgin_bits, 若 shared_mem 某个位置的成员为非空, 而 virgin_bits 上对应位置的成员为空, 既表示这是一

个该次执行覆盖了一个新的边.

为了获取所有边的覆盖频率信息, 区分高频边和低频边, 我们需要知道整个模糊测试过程中不同边的覆盖次数. 因此我们的工具额外维护了一个包含 65 536 个成员的 32 位无符号整数数组 hit_counts 来记录每条边的覆盖次数 (算法 1 第 10 行). 当一条边在某次测试中被覆盖时, 既 shared_mem 在对应位置不为 0, 则 hit_counts 的对应成员的值增加 1; 反之则不变. 我们按照 hit_counts 中数值的大小, 来区分是否为低频边, 若低频边选取过多, 会将很多常见边囊括进来; 过少则会使很多不常被执行到的边被排除在外, 这些都会影响筛选算法的最终效果. 我们通过分析实验结果, 发现覆盖次数最少的前 5% 的边相对于其他被执行到的次数要少很多, 且大多数被测程序约有 5000~12 000 条边被覆盖到, 故我们在实验中选取覆盖次数最少的前 512 条边作低频边. 若测试程序更为复杂, 覆盖的边更多, 也可增加低频边数量来更好的对种子文件进行筛选.

当完成了对某一个种子文件的测试, 或者完成了对所有种子的一轮测试之后, 我们根据 hit_counts 的状态对当前被执行过的边进行重新划分, 分出低频边和高频边, 并统计每个种子文件中的低频边的数量 (算法 1 第 14、16 行).

3.3 根据频率信息选取种子文件

在获取了每个种子文件的覆盖频率信息后, 我们利用该信息选择下一个被测试的种子文件 (算法 1 第 5 行). 在实验时, 我们发现存在部分种子文件执行速度慢, 导致覆盖率提升缓慢的问题. 为缓解这一问题, 我们将执行时间加入了选取文件的考虑范围.

AFL 的每个种子文件至少有一条边, 在这条边上该种子文件是执行得最快的. 若新加入的种子文件覆盖了原种子文件的这条边, 且执行速度比原种子文件快, 而原种子文件又没有其他边是执行最快的时, AFL 将剔除掉这个旧的文件. 我们在自己的工具中利用了这个信息, 假设当前种子文件的在 n 条边上执行最快, 且覆盖了 m 条低频边, 我们给出它的优先值为 p:

$$p=n*(m+1)$$

n 的值越大, 该种子在越多的边上执行速度最快, 表明该种子的实际速度一般也越快. m 的值越大, 该种子包含的低频边越多, 表明该种子越有测试的价值. m 可能为零, 因为种子确实可能不包含低频边. 我们在计算优先值 p 时给 m 加 1, 给那些不包含低频边, 但是测试效率高的种子提供一定的优先级.

当一个测试用例的模糊测试完成后 (7~12 行), 或整个测试进行完一轮, 即测试用例队列 S 中的输入全部使用过一遍时 (14 行), 我们根据 hit_counts 对高频和低频边进行划分 (13、15 行). 将边按 hit_counts 排序, 根据程序规模正, 取频率最低的前 n 个边作为低频边. 我们按照优先值 p 对该轮未被测试的种子文件进行排序, 选取优先值最高的种子进行测试.

3.4 性能开销分析与优化

我们算法在更新 hit_counts 时, 实际上是对 shared_mem 数组的一次遍历: 若 shared_mem 数组中元素的值为非零, hit_counts 对应位置元素的值增加 1; 反之若 share_mem 中的元素的值为零, 不改变 hit_counts 对应位置元素的值. 更新 hit_counts 的时间复杂度为 O(n), 其中 n 为 hit_counts 的规模. AFL 平均每秒能执行上千次测试, 若每测试一个生成的文件都执行一次更新操作, 则也会执行上千次, 这是一笔不可忽视的时间开销. 在实验中, 我们发现高频边和低频边的出现频率相差非常大. 这和模糊测试的特性是符合的, 既绝大多数的测试都是重复、无意义的, 只有少数能发现新的程序边或触发漏洞. 为了降低记录频率信息对模糊测试效率的影响, 除了发现新的程序边时会肯定更新 hit_counts, 其他时候我们的工具只以极低的概率更新覆盖频率信息.

划分低频边和依据频率信息选取下一个种子文件这两个功能通常只在完成对一个种子文件的完整测试后才会被调用. 种子文件需要按照文件长度的不同, 一般需要几十万、上百万次才能完成测试, 因此上两个功能的执行次数相对于更新 hit_counts 要少多了. 划分低频边等价于对 hit_counts 进行一次排序, 时间复杂度为 O(n*$\log_2 n$), 这里的 n 为 hit_counts 的规模. 选取下一个种子文件需要遍历整个种子队列, 时间复杂度为 O(m), 这里的 m 为队列长度.

4 实验评估

我们在 AFL 的基础上实现了模糊测试工具 MuFuzzer, 并实现了本文中描述的算法. 我们将本工具与原版的 AFL 进行了对比实验. 实验在一台使用 3.4 GHz 主频英特尔 4 核 8 线程 i7 处理器, 内存为 16 GB 的主机上进行, 系统为 64 位 Ubuntu 16.04. 我们参考 AFL 网站, 选取了 objdump, nm, mutool, tcpdump,

xmllint 这 5 个程序作为实验对象, 并对比了它们在原版 AFL 下和 MuFuzzer 下的实验效果.

这五个程序的信息, 以及对应的实验结果如表 1 所示. 第一行显示了 5 个目标程序的名称, 第二行是目标程序来自的软件包的名称及版本信息, 第三行给出了测试目标程序所用的具体指令. 第 4 行和第 5 行分别给出了我们的工具 MuFuzzer 和 AFL 对这 5 个程序测试运行 24 小时之后覆盖率上的结果.

表 1 实验程序的基本信息及 MuFuzzer 和 AFL 的实验结果

程序		objdump	nm	mutool	tcpdump	xmllint
软件版本		binutils-2.30	binutils-2.30	mupdf-1.13.0	tcpdump-4.9.2	libxml2-2.9.8
测试指令		objdump -d input	nm input	mutool draw input	tcpdump -nr input	xmllint -o /dev/null input
覆盖率 (%)	MuFuzzer	12.96	7.62	7.31	18.97	8.75
	AFL	11.65	7.49	6.92	16.97	8.20

总体来说, MuFuzzer 在这 5 个程序上, 最终的覆盖率均大于 AFL. 其中在 objdump 上提升了 1.31% 的覆盖率, 相对提升为 11.24%; nm 上提升了 0.13% 的覆盖率, 相对提升为 1.74%; mutool 上提升了 0.39%, 相对提升为 5.64%; tcpdump 上提升为 2.0%, 相对提升为 11.79%; xmllint 提升为 0.55%, 相对提升 6.71%. 对于全部 5 个测试程序, MuFuzzer 相对于 AFL 有 8.55% 的相对覆盖率提升. 上述实验结果表明我们的方法能够确实的提高基于覆盖率的灰盒模糊测试的最终覆盖率.

图 2 描述了 MuFuzzer 和 AFL 在对 5 个目标程序进行的为期 24 小时的实验中覆盖率随时间的状态变化. 纵坐标为被覆盖的边的数量, 横坐标是时间. 值得关注的有两点, 首先 MuFuzzer 在 objdump, nm, mutool 这 3 个目标程序中, 一直保持着覆盖率对 AFL 的领先态势直到测试结束, 而在 tcpdump 和 xmllint 中, MuFuzzer 和 AFL 一度交替领先. 我们通过分析发现, 在后两个程序的种子序列中, 存在一些体积较大的文件. 体积大导致每次执行所花时间较长, 需要测试的次数变多, 总的时间变多. 而这些文件覆盖了不少低频边, 且其他种子文件不经过这些边, 所以它们仍然是覆盖这些边的执行速度最快的种子. MuFuzzer 在计算优先度时将它们排在前面, 导致了 MuFuzzer 提前在上面花了不少时间测试, 因而覆盖率提升变慢.

其次, 在最后的几个小时, 所有程序的覆盖率曲线接近平滑, 基本不再提升, 这是因为模糊测试工具完成了对所有种子文件的测试, 开始重复测试旧的文件所导致的. MuFuzzer 的曲线平滑后的值都要高于 AFL, 假设种子序列相同, 所测的文件都一样, 出现这种结果是不符合逻辑的. 我们通过分析推测原因如下, MuFuzzer 会优先测试低频边多的种子文件, 而一些边需要满足一系列的前置条件才能达到. 例如: 假设边 x 需要同时经过边 a 和边 b 才能满足条件, 且在最初的测试中经过低频边 ab 的种子文件被发现了. MuFuzzer 通过频率筛选会优先选择测试这个种子文件, 因而很可能发现过 x 的新种子文件. AFL 则不会优先测试这个文件, 在后续的测试中发现了两个高效新种子分别过 a 和 b 并淘汰了原种子文件. 因为 ab 都是低频边, AFL 在对新文件测试的过程中, 很难生成同时通过 ab 的文件, 进而更难以发现边 x. MuFuzzer 和 AFL 在实际中都会以效率为标准剔除一些速度慢的种子, 因而最终结果不一样. MuFuzzer 的方法能提升最终的程序代码覆盖率, 证明了我们方法的价值.

5 结束语

模糊测试是作为一种行之有效的漏洞检测方法, 自诞生起一直是计算机安全领域的重要研究对象之一. 一些现有研究尝试提高生成输入的质量来提高模糊测试的效果. Hybrid-Fuzz[15]利用符号执行的方法, 收集关注的程序路径上的约束条件, 使用约束求解来生成能实际覆盖该路径的测试输入, 以此达到检测目标程序特定代码点的功能. Dowser[16]利用污点分析, 减少符号执行中约束求解的复杂, 提高生成输入的速度. Driller[17]考虑了符号执行的高计算量复杂度的问题, 只在测试工具长时间无法发现目标程序的新代码时调用约束求解.

与上述研究不同, 本文选择了改进种子文件筛选策略来提高模糊测试的效果. 本文提出了一种基于覆盖频率的种子文件排序算法, 通过记录边的总覆盖次数来区分低频边与高频边, 优先选择包含更多低频边的、执行速度快的种子文件进行测试. 我们在 AFL 上实现了该算法, 实验表明我们的算法能够提高新代码片段的发现速度, 以及最终的代码覆盖率, 对于实际的运用有着积极的意义.

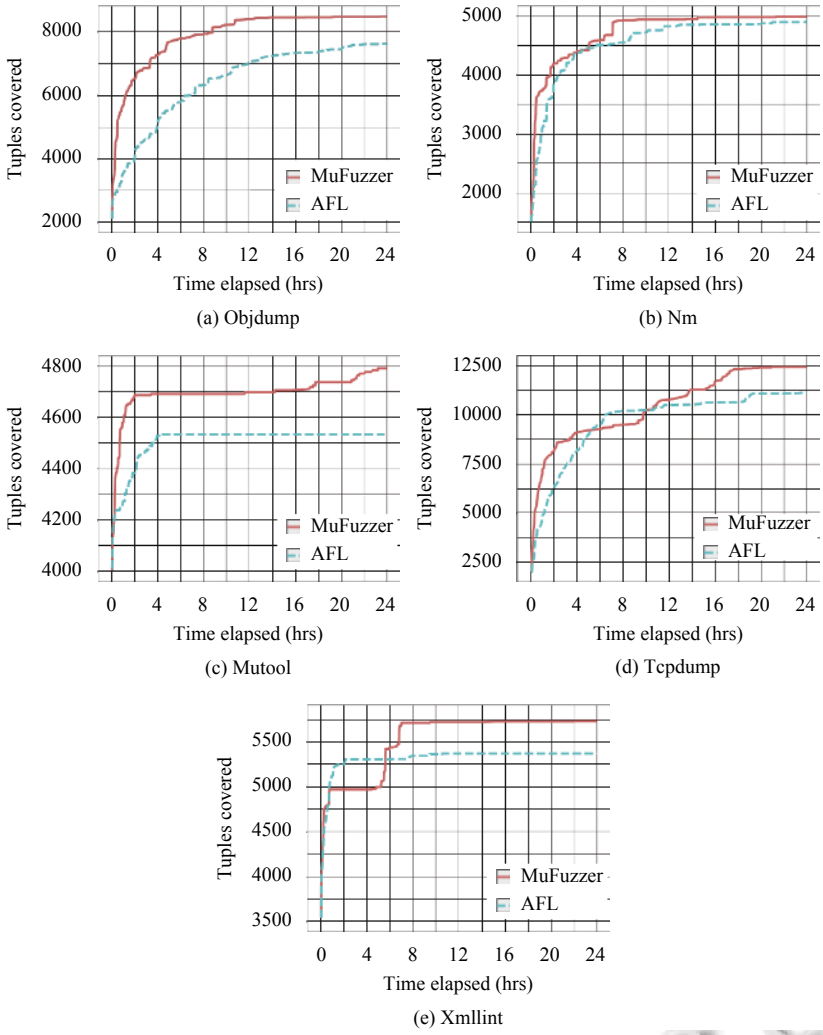

图 2 程序的覆盖率提升状态图

参考文献

1 Miller BP, Fredriksen L, So B. An empirical study of the reliability of UNIX utilities. Communications of the ACM, 1990, 33(12): 32–44. [doi: 10.1145/96267.96279]

2 Pham VT, Böhme M, Roychoudhury A. Model-based whitebox fuzzing for program binaries. Proceedings of the 2016 31st IEEE/ACM International Conference on Automated Software Engineering. Singapore. 2016. 543–553.

3 PEACHTECH. Peach fuzzer platform. https://www.peach.tech/products/peach-fuzzer/peach-platform/. [2018].

4 Yang XJ, Chen Y, Eide E, *et al*. Finding and understanding bugs in C compilers. ACM SIGPLAN Notices, 2011, 46(6): 283–294. [doi: 10.1145/1993316.1993532]

5 Woo M, Cha SK, Gottlieb S, *et al*. Scheduling black-box mutational fuzzing. Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security. Berlin, Germany. 2013. 511–522.

6 Ganesh V, Leek T, Rinard M. Taint-based directed whitebox fuzzing. Proceedings of the 2009 IEEE 31st International Conference on Software Engineering. Vancouver, BC, Canada. 2009. 474–484.

7 Godefroid P, Kiezun A, Levin MY. Grammar-based whitebox fuzzing. ACM SIGPLAN Notices, 2008, 43(6): 206–215. [doi: 10.1145/1379022.1375607]

8 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, CA, USA. 2008. 209–224.

9 Cha SK, Avgerinos T, Rebert A, *et al*. Unleashing mayhem on binary code. Proceedings of 2012 IEEE Symposium on

Security and Privacy. San Francisco, CA, USA. 2012. 380–394.

10 Michał Z. American fuzzy lop. http://lcamtuf.coredump.cx/afl/. [2018].

11 Li YK, Chen BH, Chandramohan M, *et al*. Steelix: Program-state based binary fuzzing. Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering. Paderborn, Germany. 2017. 627–637.

12 Böhme M, Pham VT, Roychoudhury A. Coverage-based greybox fuzzing as Markov chain. Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security. Vienna, Austria. 2016. 1032–1043.

13 Böhme M, Pham VT, Nguyen MD, *et al*. Directed greybox fuzzing. Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security. Dallas, Texas, USA. 2017. 2329–2344.

14 RASH M. afl-cve. https://github.com/mrash/afl-cve. [2017].

15 Pak BS. Hybrid fuzz testing: Discovering software bugs via fuzzing and symbolic execution[Master's Thesis]. Pittsburgh: Carnegie Mellon University, 2012.

16 Haller I, Slowinska A, Neugschwandtner M, *et al*. Dowsing for overflows: A guided fuzzer to find buffer boundary violations. Proceedings of the 22nd USENIX Conference on Security. Washington, DC, USA. 2013. 49–64.

17 Stephens N, Grosen J, Salls C, *et al*. Driller: Augmenting fuzzing through selective symbolic execution. Proceedings of Network and Distributed System Security Symposium. San Diego, CA, USA. 2016. 1–16.