DOI:10.19651/j.cnki.emt.2209276

基于控制流分析的导向性灰盒模糊测试方法*

黎君玉 罗 琴 刘 智
(西南石油大学计算机科学学院 成都 610500)

摘 要: 模糊测试(Fuzzing)是软件漏洞挖掘的主要技术,它能随机生成测试用例并动态执行程序,可以覆盖较深的分支。但模糊测试技术中变异存在一定的盲目性,并且随机变异样本执行相同路径的频率很高,导致变异样本冗余,从而降低测试效率。本文提出并实现了一种基于控制流分析的导向性灰盒模糊测试方法 CTM。CTM 首先对目标二进制程序进行静态分析获取程序控制流图,再根据程序控制流分析程序路径执行稀有度,接着识别执行路径上敏感函数来计算程序执行路径比重,并且求解生成测试用例;其次在模糊测试过程中,对非格式关键信息位置进行变异;最后根据支路覆盖反馈信息,利用启发式规则对执行路径约束信息进行求解,来生成新测试用例样本。CTM 通过引导性的测试用例和定位变异方法,提高模糊测试生成满足复杂分支条件测试用例的概率,从而提高代码覆盖率和减少变异样本冗余。为了验证本方法有效性,本文选择 readelf、gif2png 等真实应用程序进行测试,并与业界主流 Fuzzing 软件 Driller 和 AFL 进行对比测试,测试结果表明,CTM 发现 crash 和探索新路径的能力都有所提高。

关键词: 符号执行;模糊测试;控制流图;约束求解;测试用例
中图分类号: TP311 **文献标识码:** A **国家标准学科分类代码:** 520.1060

Grey box test case generation method based on control flow analysis

Li Junyu Luo Qin Liu Zhi
(School of Computer Science, Southwest Petroleum University, Chengdu 610500, China)

Abstract: Fuzzing is the main technology of software vulnerability mining. It can randomly generate test cases and dynamically execute programs that can cover deeper branches. However, there is a certain blindness in mutation in fuzzing technology, and the frequency of random mutation samples executing the same path is very high, resulting in redundancy of mutation samples, thus reducing the test efficiency. This paper proposes and implements a guided grey-box fuzzing method CTM based on control flow analysis. CTM first statically analyzes the target binary program to obtain the program control flow graph, then analyzes the execution rarity of the program path according to the program control flow, then identifies the sensitive functions on the execution path to calculate the program execution path proportion, and solves and generates test cases; The position of non-format key information is mutated in the testing process; Finally, according to the feedback information of branch coverage, the execution path constraint information is solved by heuristic rules to generate new test case samples. CTM improves the probability of fuzzing to generate test cases that satisfy complex branch conditions through guided test cases and locating mutation methods, thereby improving code coverage and reducing mutation sample redundancy. In order to verify the effectiveness of this method, this paper selects real applications such as readelf and gif2png for testing, and compares it with the mainstream Fuzzing software Driller and AFL in the industry. The test results show that CTM's ability to detect crashes and explore new paths has been improved.

Keywords: symbolic execution;fuzzing;control flow graph;constraint solver;test case

0 引 言

随着计算机网络技术的飞速发展,计算机软件在医疗、交通、餐饮、金融等领域已成为不可或缺的一部分。随着软件规模扩大和软件复杂度的提高[1-2],软件安全问题也日益突出。软件中的漏洞为不法者提供可乘之机,近年来不法

收稿日期:2022-03-15
* 基金项目:国家自然科学基金(61902328)项目资助

者利用软件漏洞进行勒索的违法行为屡见不鲜。2021 年 Lo4j 漏洞危及数以百万计的应用；2021 年 5 月 Colonial Pipeline 受 DarkSide 攻击中断了数百万加仑燃料的运输，造成不可估量的危害。因此高效、自动化的安全测试方法对软件安全测试至关重要[3]，这也成为计算机科学和网络安全重要的研究方向。

目前在安全测试领域方面常见漏洞挖掘技术有静态分析、污点分析、符号执行、模糊测试等方法。模糊测试因原理简单，部署方便，自动化程度高而备受关注。

模糊测试(Fuzzing)是一种动态的程序测试方法，它通过向程序输入畸形的数据来测试程序存在的漏洞，以此有效地发现软件漏洞[4-5]。模糊测试根据对程序源代码的依赖程度和程序分析程度可分为白盒、黑盒和灰盒测试。白盒测试是对源代码进行分析，然后进行模糊测试。它能有效定位程序漏洞，但其消耗大量人力、物力。黑盒测试是在没有任何目标程序内部知识的情况下直接进行模糊测试，测试过程随机性太大。灰盒测试无需源代码，通过对二进制程序分析获取目标程序的内部信息后再进行模糊测试，能提高程序测试效率。因此灰盒测试更受人们关注。在大多情况下，由于商业利益和知识产权保护等原因，获取程序源代码难度大。并且相比源代码级别，二进制程序的模糊测试难度更大，更具挑战性。

AFL(american fuzzy loop)[6]是最具代表性的灰盒模糊测试工具。它利用遗传算法并以代码覆盖信息作为反馈用于种子筛选。相对于传统的模糊测试方法，AFL 能够有效提高模糊测试的效率，但该方法存在一定的局限性。如在模糊测试变异过程中会产生大量相同路径的无效测试用例，且在面对二进制程序中复杂分支检查时，随机变异难以生成满足该分支约束的测试样本，其变异存在一定盲目性，导致难以探索更深的路径。基于 AFL 的衍生版本[7-8]能够利用支路覆盖信息作为反馈，计算边能量生成测试用例。AFLFast[7]利用马尔科夫链对可达到路径进行优先级排序，通过统计路径出现频率，在高频率的路径上进行更深入地挖掘。AFLGo[8]结合程序内控制流图计算出每一个基本模块距离目标点的距离，在测试时根据此变量生成达到目标位置的输入。CollAFL[9]通过获取明确覆盖率信息减少路径碰撞。Steelix[10]则是在输入测试的过程中定位到 magic byte，然后根据突变特定的输入去高效适配 magic byte 来达到更深、更广的覆盖。这些衍生版本以测试用例选择和调度策略以及支路覆盖信息提高模糊测试效率。但利用启发式规则生成测试用例存在一定随机性，有可能会非常的高效，但也有可能会无法发现新的路径。并且基于反馈的方法受限于特定软件中分支的覆盖范围。

结合特定符号执行框架[11-13]能有效辅助模糊测试生成测试用例[14-16]。Driller[14]是符号执行和模糊测试结合的混合执行方法。它通过路径约束求解方式探索新路径以此覆盖更广的代码。但太依赖求解器能力，面临多重分支条件易出现路径爆炸问题。QSYM[15]、Munch[16]均利用混合执行方式提高模糊测试效率，Driller 侧重分支覆盖率、QSYM 侧重执行效率、Munch 侧重函数覆盖率，但符号执行技术不可避免存在路径爆炸问题。

针对上述问题，本文提出一种基于控制流分析的导向性灰盒模糊测试方法 CTM(control flow test case generate and mutation)，该方法首先通过分析二进制程序执行路径信息生成特定测试用例，其次根据生成用例覆盖路径信息对种子特定位置进行变异，最后检测边覆盖反馈信息和其更新状态，当无新状态时约束求解生成新测试用例。相比 AFL，该方法能有效地探索更深的路径并减少执行高频路径的测试用例次数，使覆盖率得到很大的提高。同时，该方法与 Driller 相比还进一步地缓解了路径爆炸问题，提高了测试效率。本文选 readelf、gif2png 等程序作为测试程序，与主流工具 Driller 和 AFL 进行对比，结果表明本方法在发现 crash 和探索新路径的能力上都有所提高。尤其在 bmp2tiff 和 gif2swf 上，本方法分别生成 unique crash 2 个和 47 个，而 AFL 和 Driller 在此上均无发现。

1　导向性模糊测试方法

1.1　核心处理流程

为了提高模糊测试对二进制程序测试效率，本文通过结合静态分析和符号执行，提出并实现一种基于控制流分析的导向性灰盒模糊测试方法 CTM(contorl flow test case generate and mutation)。核心处理流程如图 1 所示。

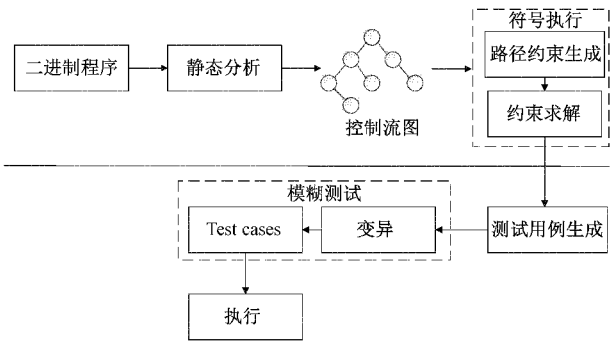

图 1　核心处理流程

1)对目标二进制程序进行静态分析，提取程序控制流图 CFG(control flow graph)，并根据控制流信息分析程序执行路径和常见敏感函数信息，通过比较程序执行路径上可识别敏感函数危险程度，把有效路径优先选取保留下来；

2)根据获取到的执行路径信息，利用符号执行将其求解得到初始测试用例作为初始种子；

3)在模糊测试中选择种子变异的阶段，对种子文件中已生成魔法字节和关键信息进行保留，对其他格式特征进行破坏。

4)在模糊测试过程中一直监视程序测试用例生成，当生成变异种子很久没有探索新路径时，根据收集测试用例，

针对种子执行路径,选取其中一份种子进行 trace,将其约束格式信息获取,利用一些启发式规则取反路径约束获取新的测试用例。

1.2 测试用例生成

控制流图是程序分析的基础,通过程序控制流图能够获取遍历程序执行过程中所有路径。程序控制流图每一个节点代表一个基本块(basic block)。为了提高程序的稳定性,大部分程序都对外来输入进行校验,一般通过条件分支语句设置魔法字节对文件输入格式进行校验。本文在二进制程序反汇编生成控制流图过程中,能够获取到程序检验的比较指令或者比较函数,其中常见的比较指令有strncmp、cmp 等指令。这类型的指令的执行都会影响程序执行路径,从而影响程序后续代码覆盖。本文特别关注这些路径信息,并且根据这些指令和路径信息进一步收集程序执行路径约束条件。根据控制流信息获取到程序执行过程中遍历路径信息后,还进一步分析程序执行过程中对"有趣"路径优先选择。当程序执行过程中所经过路径所包含敏感函数更多,所占权重能量越大,这条路径优先被选中进行执行,生成种子优先变异。在对程序静态分析过程中,本文还需对表 1 中敏感函数进行识别挑选。C 语言中存在一些较为敏感的函数。对于这些函数,程序没有对这些函数进行异常处理或者限制,直接使用很容易留下隐患。例如 gets 函数,用户无法指定其一次最多读入多少字节的内容,这使得 gets 变得十分的危险。因此对于函数的危险性,本文给予它进行分级赋予一定的权重能量,当执行路径中包含当前函数时,根据权重能量大小选择最优路径。

表 1 敏感函数表

函数	严重性	权重
gets	高危	3
strcpy	高危	3
sprintf	高危	3
scanf	高危	3
sscanf	高危	3
fscanf	高危	3
getchar	中危	2
0fgetc	中危	2
getc	中危	2
read	中危	2
memcpy	低危	1

为了能够提高模糊测试中测试用例的质量,减少无效样本干扰。本文设计基于控制流图的敏感执行路径的导向性测试用例生成方法。算法整体流程如下:

算法 1 描述具体符号执行过程。首先初始化栈 S 和崩溃样本 seeds(第 1 行),为二进制程序 P 构建控制流图 CFG(第 2 行),如果栈非空,则从栈中弹出一个节点 N_i(第 4~5 行)。对于 N_i 的每个未访问的相邻节点 N_j,按照添加权重的深度优先算法选择一个孩子节点 N_j(第 6 行),识别判断函数并对于特定敏感函数进行优先标记并选择(第 7 行),检查 N_j 是否有孩子节点(第 8 行)。如果没有子节点就把执行路径 Path 放入求解模块,然后生成一个特定的测试用例 ans(第 10 行)。然后 fuzzer 将 ans 作为初始输入进行变异生成崩溃样本,然后将崩溃样本 s 添加到 seeds(11~13 行)。如果 N_j 至少有一个子节点,它将被推入堆栈 S(第 15~16 行)。最后,崩溃的输入 seeds 作为最终的输出返回。

算法 1:测试用例生成方法

Input: A binary program p

Output: Crashing inputs seeds

1. Initial a stack S←Ø, seeds←Ø;
2. Construct a CFG G for p;
3. S←N_0;
4. while S=Ø do
5. N_i=S.pop();
6. weight_children_node=next_node(N_i)
7. unsecured_Function=unsecured();
8. if weight_children_node has no children then
9. path=N_0,…, N_i,N_j;
10. ans=concolic_execution_solver(p, path);
11. for Each crashing input s produced by fuzzer (ans) do
12. seeds←seeds∪s;
13. end for
14. else
15. new_node=weight_node();
16. S.push(new_node);
17. end if
18. end while
19. return seeds;

1.3 变异策略

为了能有效引导初代测试用例进行变产生有趣样本,本文在 AFL 变异策略基础上做了改进。首先针对执行路径信息频率高的样本进行划分,然后保留其特定魔法字节和其他关键信息不变对特定位置进行破坏。

对于实际应用程序来说外来输入格式都是十分复杂,程序中约束路径同样十分复杂,例如图片格式或者 ELF(executable and linking format)文件格式。与这些相比,现实世界中用例格式比表 2 中邮箱账号输入格式样本复杂许多。因此根据路径保留其中格式信息对于程序测试十分重要。本文目标是针对具有相同执行路径的测试用例。把用例划分关键信息和位置两个部分,其中关键信息保留的是执行路径上测试用例子串,位置是其子串所在位置。

表 2 中所出现的 4 个测试用例,基于最长公共子串搜

索算法推断出所有子串，选择硬编码作为最高优先级的子字符串，其位置在一组的所有测试用例中都是固定的。然后根据硬编码将集合分成几个子集。对于每个子集中的字符串，选择最长和最左边的子字符串作为第二高优先级子字符串。然后按顺序排列存储，并且记录子串各自所在的位置信息。其中位置为－1，则子串位于测试用例末尾（例如所有测试用例以“.com”结尾），它在测试用例的位置是固定的（例如“mail:”是所有字符串中的前 4 个字符）。位置为－2，这表示其在测试用例中的位置是可变的（例如“@”在字符串中的位置）。在 AFL 对测试用例变异过程中，在实现确定性策略（如算术）时，对位置不在固定位置数组中的字符进行变异，并利用该格式来调节不确定性策略（如 havoc）生成的测试用例。通过对关键位置格式信息进行保留，对无格式信息位置利用不确定变异策略对其进行突变、破坏，提高发现程序 crash 的能力。

表 2　测试用例划分变异

测试用例
mail:binary@qq.com
mail:program@163.com
mail:mutation@139.com
mail:strategy@126.com
关键信息
“mail:”,“a”,“@”,“.com”
位置
0,－2,－2,－1

1.4　混合符号执行

大部分情况下，模糊测试技术能够利用随机值和翻转以及突变策略技术能够探索较多的路径，但面对复杂分支条件难以生成特定输入探索更深的路径（例如图 2 中示例代码所示，需要特定输入才能通过第 7 和第 9 行检查语句）。本文通过跟踪有趣的种子输入，程序执行路径分支被当作一个符号求解程序，通过约束求解生成样本来满足达到通过该条件检查的输入。

本文通过检测模糊测试过程中探索新路径状态。当长时间无法发现新的边路时，选择并跟踪当前种子执行路径。通过预约束方式确保跟踪的种子执行结果与当前执行结果一致。在预约束执行过程中，输入的每个字节都被约束为与种子的实际字节相匹配。当执行到控制流中分支跳转条件时，通过反转约束条件是否发现新的状态分支。若发现新的状态转换，则短暂地取消预约束，求解出满足变换的输入。例如，如图 2 所示，它先约束输入中的所有字节，然后对跟踪的输入进行匹配，当跟踪样本执行到第 7 行代码，此时存在控制流分支跳转中魔法字节校验。取消后续约束输入，通过反转约束条件判断是否存在一个之前未识别的状态转换。若存在，则删除开始添加的预约束，获取这个位置的路径约束（$x=0x4D42$），生成一个在特定位置包含了魔术字节的输入。当求解生成新的状态转换的测试用例后，将其加入种子集，并将新加入的种子优先选择继续进行模糊测试。若无法发现新的状态，则恢复当前跟踪路径约束继续探索。最后重复整个导向式模糊测试的循环。

```
1  if(size > 512 || size < 0)
2  {
3      close(fd);
4      return -1;
5  }
6  read(fp,buf,size)
7  if(strncmp(&buf[0],"0x4D42",5) == 0)
8  {
9      if(strncmp(&buf[10],"89A",3) == 0)
10     {
11         bug();
12         close(fp);
13         return 0;
14     }else
15     {
16         ---bug()--;
17     }
18 }
```

图 2　示例代码

2　实验与分析

为了验证所提方法的有效性，本文基于 CTM 方法构建了一个模糊测试工具 CTMFuzz，如图 3 所示。CTMFuzz 是基于二进制分析平台 Angr 和模糊测试工具 AFL 开发的。本文首先通过静态分析了解程序内部知识，然后求解生成测试用例；其次在 AFL 中修改、添加变异策略引导种子变异方向；最后并行调度符号执行引擎求解新边路测试用例。本文通过 CTM 方法以此绕过条件分支检查，从而探索更深更广的路径。本文把 CTMFuzz 与主流模糊测试工具 AFL 和 Driller 做对比实验。针对示例代码和实际程序 readelf、gif2png、bmp2tiff、gif2swf 等二进制程序进行测试评估。实验环境 Ubuntu 16.04 64 位操作系统，AMD 3600X 处理器和 32 G 的内存。

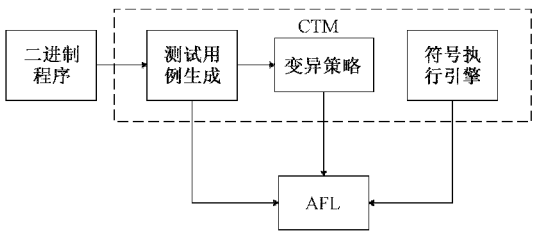

图 3　CTMFuzz 总体框架

2.1　示例代码测试

本文对表 3 中 3 个工具做对比实验。针对图 3 示例代码设置两个触发点测试，检验触发所需时间。初始样本均设置为“fuzz”。在模糊测试过程中变异存在一定随机性，通过 3 次实验，取 3 次实验结果的平均值。如表 3 所示，表格中展示 3 个工具对示例代码测试结果。

本文在示例程序中设置 2 个触发点，从表格中可以看出，CTMFuzz 很快就可以达到两个触发点，分别是 7 s 和

表 3 示例程序结果对比 s

对比项	第 1 个触发耗时	第 2 个触发耗时
AFL	358	658
Driller	338	624
CTMFuzz	7	13

13 s。反观 AFL 和 Driller 它们需要超过 10 min 才能触发 2 个点。本文对 AFL 和 Driller 都测试 3 次以上,AFL 最快触发第 1 点需要 124 s,最慢要 491 s,触发第 2 个点要 607 s。很明显看出基于随机变异策略在面对一些较复杂分支时,要触发 bug 所在处需要花费大量时间,且随机变异中随机性大,导致效率不稳定。Driller 触发 1 个点平均时间 338 s,第 2 个点需要 624 s 左右。Driller 十分依赖求解器求解能力,在面对大部分程序难以求解出合适测试用例情况下,其效率趋于 AFL。因此看出本工具面对二进制程序测试上极大提高程序测试效率。

2.2 实际程序测试

本文选取 readelf、gif2png、bmp2tiff、gif2swf 等二进制程序作为评估对象,表 4 呈现程序的相关信息,选取 AFL 中 testcase 的样本作为初始输入。

表 4 测试程序介绍

程序	版本	介绍
readelf	2.3	用于查看 ELF 格式的文件信息
gif2png	2.5.8	图片格式转化 gif 转 png
bmp2tiff	4.0.6	图片格式转化 bmp 转 tiff
gif2swf	0.9.2	图片格式转化 gif 转 swf
tiff2pdf	4.0.6	tiff 文件转 pdf 文件格式

为了进一步评估本方法的有效性,本文将会从 crash 数量、执行路径数量、新分支覆盖数量 3 个方面对本系统进行评估。本文选取表 4 中的程序作为对 CTMFuzz 在实际应用中漏洞挖掘能力与路径探索能力评估。并与主流模糊测试 AFL 和 Driller 做对比实验。在同等测试环境下,使用相同测试样本情况下进行 6 h 的模糊测试。模糊测试过程中存在一定随机性,通过实验 3 次取平均值,分别记录 3 个工具 crash 样本以及路径数量和新分支覆盖数量。实验结果如表 5~7 所示。

表 5 crash 数量对比结果

对比项	readelf	gif2png	bmp2tiff	gif2swf	tiff2pdf
AFL	0	16	0	0	0
Driller	0	24	0	0	0
CTMFuzz	0	57	2	47	0

crash 和执行路径的数量是体现模糊测试有效性的重要指标。由表 5 实验结果中看出,与 AFL 和 Driller 相比,

表 6 执行路径对比结果

对比项	readelf	gif2png	bmp2tiff	gif2swf	tiff2pdf
AFL	1 531	429	72	83	336
Driller	1 555	464	191	97	313
CTMFuzz	2 935	779	239	390	426

表 7 新分支覆盖对比结果

对比项	readelf	gif2png	bmp2tiff	gif2swf	tiff2pdf
AFL	874	99	64	15	134
Driller	877	108	64	15	112
CTMFuzz	1 110	117	67	47	256

CTMFuzz 能够发现更多的 crash,尤其在 gif2png 和 gif2swf 应用程序上的差异更为明显。在 bmp2tiff 和 gif2swf 上,AFL 和 Driller 在测试过程中都没有能够产生触发实际应用程序中 crash 的样本,而 CTMFuzz 在 bmp2tiff 和 gif2swf 分别触发 2 个和 47 个 unique crash。在 gif2ng 程序上,CTMFuzz 触发 57 个 unique crash,比 AFL 多 41 个,Driller 多 33 个。图片文件格式数据段格式各不相同。例如 bmp 图片格式由文件头、位图信息头、调色板、位图数据组成;tiff 图片文件头文件、图像文件目录、目录项组成。程序对输入图片解析格式方式也不同。并且图片格式转化程序中对于文件输入有校验机制,导致在随机变异过程中难以达到较深的路径。只有单一样本情况下,通过随机变异和路径反馈的方法能探索程序新路径的概率很低。

由表 6~7 实验结果中看出,本工具在执行路径总数上和新分支路径覆盖数量都优于 AFL 和 Driller。特别在 readelf、gif2swf 和 tiff2pdf 应用程序上的差异尤为明显。在 readelf 程序上,CTMFuzz 在总路径数效果上比 AFL 和 Driller 提升近 89%,在新分支覆盖上提升 26%。在 gif2swf 程序上,CTMFuzz 在总路径数效果上比 AFL 和 Driller 提升近 400%,在新分支覆盖上提升超过 300%。由于 readelf 应用程序对文件处理方式较为复杂,且输入文件格式结构难以获取。不同程序的处理数据方式不同,Driller、AFL 通过变异难以覆盖更多的路径。程序对输入的复杂处理流程和输入文件校验机制,导致能够求解的输入相对较少。Driller 符号执行对复杂的输入,面对多重复杂的分支难以求解出合适测试用例,致使无法探索更多的路径。本系统工具通过分析程序内部结构对输入处理流程和执行路径信息,能够有效指导测试用例生成,能够覆盖到更多的分支。进一步通过对非关键格式部分变异能够生成恶劣样本测试程序的存在漏洞。因而能够有效提高发现程序 crash 和探索程序新路径的能力。

图 4~5 是 readelf 程序测试过程路径执行变化检测图。图 6~7 是 gif2swf 程序测试过程路径执行变化检测图。total paths 是程序执行总路径,current path 当前路

径，pending paths 待测路径，pending favs 待测有意思的路径，cycles done 循环次数。

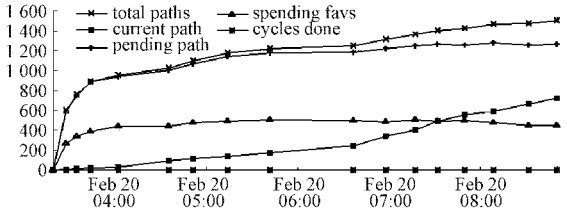

图 4　readelf：AFL 分支路径探索时间状态图

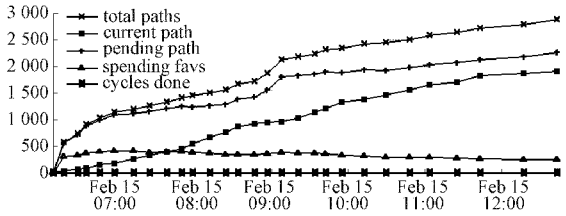

图 5　readelf：CTMFuzz 分支路径探索时间状态图

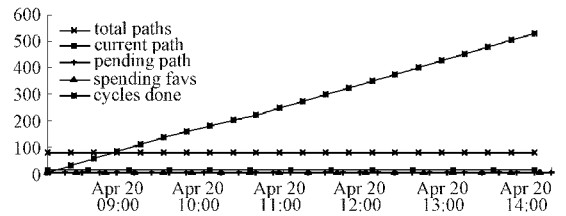

图 6　gif2swf：AFL 分支路径探索时间状态图

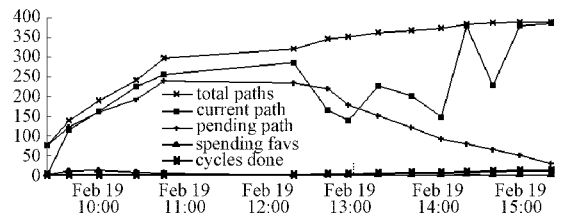

图 7　gif2swf：CTMFuzz 分支路径探索时间状态图

从实验结果中曲线状态看出，本方法在 readelf 程序上对于路径探索一直呈上升趋势，而 AFL 在 2 h 后曲线趋向平缓，readelf 程序内部存在复杂分支，初始样本不足，导致 AFL 通过随机变异策略生成测试用例无法覆盖更多的分支。在 gif2swf 上，从实验结果表示本方法优势更为突出，探索执行路径在开始就有极大突破，并且能够触发程序的崩溃状态，而 AFL 对于此程序无法有新的进展。AFL 不断对样本进行变异，cycles done 循环次数一直上涨，但却无法变异出探索新路径的样本。Driller 在面对这些实际程序时，由于太过于依赖求解器能力，导致无法产生新的样本，所以其状态与 AFL 近乎一致。

综合上面的对比数据可以看出，通过基于 CFG 具有导向性的模糊测试方法优化的 CTMFuzz 模糊测试工具，较之于 AFL 和 Drillc 在测试效率和覆盖率上有极大提高。从而表明本文提出的方法有效地提高了 Fuzzing 发现 crash 和探索新路径的能力，同时减少了时间开销。

3　结　　论

针对现有模糊测试技术代码覆盖率不足，变异存在盲目性的问题，本文提出一种基于控制流分析的导向性灰盒模糊测试方法，并实现原型系统 CTMFuzz。该方法首先从测试用例生成角度进行引导，通过分析二进制程序控制流，获取程序执行过程中执行路径信息和函数调用信息生成特定测试用例；其次从变异策略角度针对种子信息进一步对变异进行引导；最后在模糊测试过程中并发调用符号执行，对种子执行分支反馈信息方向进行引导，利用启发式规则对种子执行路径上约束求解生成新的测试用例。与其他符号执行和模糊测试结合的测试工具相比，本文采用轻量级的静态分析方法和符号执行技术，通用性高，减少了时间开销。通过实验结果表明 CTMFuzz 在发现 crash 能力和探索新路径能力上都优于 AFL 和 Driller。但是由于 Angr 是符号执行框架，其不可避免地存在路径爆炸问题。并且面对逻辑复杂的程序，如果没有合适预约束输入，也会导致求解难度提高。

本文未来将针对大规模程序进行测试，同时结合文件格式输入，希望能够通过二进制程序进一步分析文件输入格式中语法和语义信息生成测试用例，并希望能将其应用于协议测试领域。

参考文献

[1]　王叶阑，庹先国，石睿，等. 核素识别和辐射报警的安卓应用软件设计[J]. 自动化仪表，2020，41(4)：37-40.

[2]　郭莎，孟丽囡，唐源，等. 功率变换器虚拟测量系统的研究与实现[J]. 国外电子测量技术，2020，39(6)：27-31.

[3]　贺晋宁，杜伟伟，高静. 软件自动化测试的探索实践[J]. 国外电子测量技术，2016，35(7)：1-4.

[4]　BOEHME M，CADAR C，ROYCHOUDHURY A. Fuzzing：challenges and reflections[J]. IEEE Softw，2021，38(3)：79-86.

[5]　SLOWINSKA A，STANCESCU T，BOS H. Howard：A dynamic excavator for reverse engineering data structures[C]. Network & Distributed System Security Symposium DBLP，2011.

[6]　MICHAL Z. AFL technical details[EB/OL]. (2017-07-15)[2022-04-10]. http://lcamtuf. coredump. cx/afl/technical_details. txt.

[7]　BÖHME M，PHAM V T，ROYCHOUDHURY A. Coverage-based greybox fuzzing as markov chain[J]. IEEE Transactions on Software Engineering，2017，45(5)：489-506.

[8]　BÖHME M，PHAM V T，NGUYEN M D，et al.

Directed greybox fuzzing[C]. Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, 2017: 2329-2344.

[9] GAN S, ZHANG C, QIN X, et al. Collafl: Path sensitive fuzzing [C]. 2018 IEEE Symposium on Security and Privacy(SP), IEEE, 2018: 679-696.

[10] LI Y, CHEN B, CHANDRAMOHAN M, et al. Steelix: Program-state based binary fuzzing [C]. Proceedings of hc 2017 11th Joint Meeting on Foundations ofSoftware Engineering, 2017: 627-637.

[11] 任蒲军,付敬奇.一种 Modbus TCP 模糊测试中畸形数据过滤方法[J].电子测量技术,2019,42(7):7-12.

[12] SHOSHITAISHVILI Y, WANG R, SALLS C, et al. Sok:(state of) the art of war: Offensive techniques in binary analysis [C]. 2016 IEEE Symposium on Security and Privacy(SP), IEEE, 2016: 138-157.

[13] CHIPOUNOV V, KUZNETSOV V, CANDEA G. S2E: A platform for in-vivo multi-path analysis of software systems[J]. Acm Sigplan Notices, 2011, 46(3): 265-278.

[14] STEPHENS N, GROSEN J, SALLS C, et al. Driller: Augmenting fuzzing through selective symbolic execution[C]. NDSS, 2016, 16(2016): 1-16.

[15] YUN I, LEE S, XU M, et al. QSYM: A practical concolic execution engine tailored for hybrid fuzzing[C]. USENIX Security Symposium. USENIX Association, 2018.

[16] OGNAWALA S, HUTZELMANN T, PSALLIDA E, et al. Improving function coverage with munch: A hybrid fuzzing and directed symbolic execution approach [C]. The 33rd Annual ACM Symp on Applied Computing, SAC, 2018: 1475-1482.

作者简介

黎君玉,硕士研究生,主要研究方向为网络空间安全,主要研究二进制安全漏洞挖掘方向。

E-mail:645540367@qq.com

罗琴,博士,副研究员,主要研究方向为网络空间安全。

E-mail:dorothy_lq@163.com

刘智,博士,讲师,主要研究方向为网络空间安全与机器学习。

E-mail:zhi.liu@swpu.edu.cn