软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software, [doi: 10.13328/j.cnki.jos.006225]

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

SmartFuSE: 基于深度学习的符号执行与模糊测试的混合测试方法*

高凤娟[1], 王 豫[1], 司徒凌云[2], 王林章[1]

[1](计算机软件新技术国家重点实验室(南京大学), 江苏 南京 210023)

[1](南京大学 计算机科学与技术系, 江苏 南京 210023)

[2](南京大学 信息管理学院, 江苏 南京 210023)

通讯作者: 王林章, E-mail: lzwang@nju.edu.cn

摘 要: 随着软件技术的快速发展,面向领域的软件系统在广泛使用的同时带来了研究与应用上的新挑战.由于领域应用对安全性、可靠性有着很高的要求,而符号执行和模糊测试等技术在保障软件系统的安全性、可靠性方面已经发展了数十年.许多研究和被发现的缺陷表明了它们的有效性.但是由于两者的优劣不同,将这两者的结合仍是近期热门研究话题.目前的结合方法在于两者相互协助,例如模糊测试不可达的区域交给符号执行求解.但是这些方法只能在模糊测试(或符号执行)运行时判定是否应该借助符号执行(或模糊测试),无法同时利用这两者的优势,从而导致性能不足.基于此,我们提出基于深度学习,将基于符号执行的测试与模糊测试相结合的混合测试方法.该方法旨在测试开始之前就判断适合模糊测试(或符号执行)的路径集,从而制导模糊测试(或符号执行)到达适合它们的区域.同时,我们还提出混合机制实现两者之间的交互,从而进一步提升整体的覆盖率.基于 LAVA-M 中程序的实验表明,我们的方法相对于单独符号执行或模糊测试,能够提升20%多的分支覆盖率,增加约1～13倍的路径数目,多检测到929个缺陷.

关键词: 软件测试;深度学习;路径表示;符号执行;模糊测试;混合测试

中图法分类号: TP311

中文引用格式: 高凤娟,王豫,司徒凌云,王林章.SmartFuSE: 基于深度学习的符号执行与模糊测试的混合测试方法.软件学报. http://www.jos.org.cn/1000-9825/6225.htm

英文引用格式: Gao FJ, Wang Y, Situ LY, Wang LZ. SmartFuSE: deep learning-based hybrid testing using symbolic execution and fuzzing. Ruan Jian Xue Bao/Journal of Software, 2021 (in Chinese). http://www.jos.org.cn/1000-9825/6225.htm

SmartFuSE: deep learning-based hybrid testing using symbolic execution and fuzzing

GAO Feng-Juan, WANG Yu, SITU Ling-Yun, WANG Lin-Zhang

[1](School of Computer Science and Technology, Nanjing University, Nanjing 210023, China)

[1](State Key Laboratory for Novel Software Technology (Nanjing University), Nanjing 210023, China)

[2] (Nanjing University, School of Information Management, Nanjing 210023, China)

Abstract: With the rapid development of software techniques, domain-driven softwares raise new challenges in software security and robustness. Symbolic execution and fuzzing have been rapidly developed in recent decades years, demonstrating their ability in detecting software bugs. Enormous detected and fixed bugs demonstrate their feasibility. However, it is still a challenging task to combine the two methods due to their corresponding weakness. State-of-the-art techniques focus on incorporating the two methods such as using symbolic execution to solve paths when fuzzing gets stuck in complex paths. Unfortunately, such methods are inefficient because they have to

* 基金项目: 国家自然科学基金(No.62032010); 江苏省研究生科研与实践创新计划项目

Foundation item: The National Natural Science Foundation of China (No.62032010); Postgraduate Research &Practice Innovation Program of Jiangsu Province

收稿时间: 2020-09-13; 修改时间: 2020-10-26; 采用时间: 2020-12-19; jos 在线出版时间:2021-01-22

switch to fuzzing (resp. symbolic execution) when conducting symbolic execution (resp. fuzzing). In this paper, we present a new deep learning-based hybrid testing method using symbolic execution and fuzzing. This method tries to predict paths that are suitable for fuzzing (resp. symbolic execution) and guide the fuzzing (resp. symbolic execution) to reach the paths. To further enhance the effectiveness, we propose a hybrid mechanism to make them interact with each other. We evaluate our approach on the programs in LAVA-M and the result shows compared to using symbolic execution or fuzzing independently, using our method helps more than 20% increase of branch coverage, 1 to 13 times increase of the number of paths, and uncover 929 bugs.

Key words: software testing; deep learning; path representation; symbolic execution; fuzzing; hybrid testing

软件是推动新一代信息技术发展的驱动力,随着物联网、云计算、人工智能等技术的快速发展,面向这些领域的软件不断发展.但是与之而来的是软件质量保障、软件安全性等问题.领域软件面临的挑战不仅是软件数目的增加,更主要的是由于领域应用有着自身特点,例如工控、制造等领域往往具备动态组织单元、网络泛在、数字孪生等特征,对生产效率和安全性也有着更高的要求.软件缺陷,作为从软件诞生就存在的问题,至今依旧威胁着领域软件的鲁棒性和安全性.

软件缺陷(software defect)产生于开发人员的编码过程,软件开发过程不合理或开发人员的经验不足,均有可能产生软件缺陷.而含有缺陷的软件在运行时可能会产生意料之外的后果,严重的时候会给企业造成巨大的经济损失,甚至会威胁到人们的生命财产安全.因此,越早发现软件缺陷,越能降低修复缺陷的代价.

模糊测试和基于符号执行的测试方法是当前主流、有效的缺陷检测方法.模糊测试(Fuzzing)是通过向被测程序不断提供非预期的输入并监视该程序是否出现异常,从而探测软件缺陷.它是一种基于缺陷注入的自动软件测试技术.它使用大量自动生成的测试用例作为应用程序的输入,来发现应用程序中可能存在的安全缺陷,比如缓冲区溢出.符号执行(Symbolic Execution)是一种程序分析技术.其可以通过分析程序来得到执行特定程序路径的输入.使用符号执行分析一个程序时,该程序会使用符号值作为输入,而非一般执行程序时使用的具体值.在达到目标代码时,分析器可以得到相应的路径约束,然后通过约束求解器来得到可以触发目标代码的具体输入.这两种方法都有着一定的局限性.模糊测试方法由于无法意识到路径条件,所以如果遇到窄约束的情况(如两个字符串是否相等),则无法高效的覆盖此路径[1].相比之下,基于符号执行的测试方法则由于路径爆炸以及约束求解等问题,虽然能处理窄约束,但是无法全面快速的覆盖程序空间[2](比如较深的程序位置).

目前有一些针对基于符号执行的测试和模糊测试相结合的方法[1][2][3][4][5][6][7],这些方法主要都是在其中一个方法遇到瓶颈的时候借助另一个方法来克服.例如 Driller[1]是在模糊测试时,在遇到特殊边界条件(比如满足特定输入的窄路径)无法继续探索时,则借助符号执行技术求解该路径再将求解出的结果返回模糊测试.但是这些方法都存在一个效率问题,即只能在其中一个方法遇到困难的时候才去调用另一个方法,会降低效率.

为了解决该效率问题,我们提出名为 SmartFuSE(SMART FUzzing and Symbolic Execution)的基于深度学习,将基于符号执行的测试与模糊测试相结合的混合测试方法.给定一个程序,构造该程序的路径的图表示,通过神经网络模型预测路径是适合模糊测试还是适合符号执行.对于适合符号执行的路径集,则制导符号执行优先执行该路径集.对于适合模糊测试的路径集,同样通过制导模糊测试工具尝试优先执行该路径集.同时,我们还提出混合机制完成基于符号执行的测试与模糊测试的交互,从而进一步提升整体的覆盖率.我们针对 LAVA-M 中的 4 个程序进行测试,实验结果表明我们的方法相对于单独通过模糊测试或符号执行的方法能够提升 20%多的分支覆盖率,增加约 1～13 倍的路径数目,多检测到 929 个缺陷.

总而言之,本文的主要贡献如下:

- 提出基于深度学习的符号执行与模糊测试的路径预测方法;
- 提出了制导的符号执行和制导的模糊测试方法,制导符号执行(或模糊测试)优先探索模型预测出的适合符号执行(或模糊测试)的路径集;
- 提出了通过结合基于符号执行的测试和模糊测试的混合测试方法;
- 开发了原型工具 SmartFuSE,通过实验体现 SmartFuSE 的有效性.

本文其余部分结构如下:第 1 章介绍相关背景,包括符号执行、模糊测试技术和图神经网络.第 2 章介绍

SmartFuSE 的方法框架.第 3 章介绍如何进行数据表示和数据收集来训练图神经网络模型.第 4 章介绍如何基于图神经网络模型制导符号执行和模糊测试的混合测试过程.第 5 章介绍工具实现并通过实验评估了 SmartFuSE 的有效性和性能.第 6 章介绍了相关工作.第 7 章对本文进行了总结.

```
1.  int foo (int x, char* y) {
2.      int len, t=0, r=0;
3.      len = strlen(s);
4.      if (x*len == 30) {
5.          t = x;
6.      }
7.      else {
8.          if (strcmp(s,"abc") == 0) {
9.              t = x-3;
10.         }
11.     }
12.     for (int i=0;i<y;i++){
13.         r += i;
14.     }
15.     if (r > t)
16.         return r;
17.     else
18.         return t;
19. }
20.
21. int main(int argc, char** argv) {
22.     int a, b;
23.     char c[8];
24.     FILE *fp = fopen(argv[1], "r"); // read mode
25.     fscanf(fp, "%d %d %s", &a, &b, c);
26.     fclose(fp);
27.     return foo(a, b, c);
28. }
```

Fig. 1 Example Program

图 1 示例程序

1 相关背景

1.1 符号执行

符号执行技术是一种常用的软件分析技术,最早是由 James C. King 等人于 1975 年左右提出的[9].目前 KLEE[10]是应用最为广泛的符号执行引擎之一.符号执行的关键思想就是,用符号值代替具体值作为输入,并用符号表达式来表示与符号值相关的程序变量的值.在遇到程序分支指令时,程序的执行也相应地搜索每个分支,分支条件被加入到符号执行保存的程序状态的 π 中,表示当前路径的约束条件.在收集了路径约束条件 π 之后,使用约束求解器来验证约束的可满足性,以确定该路径是否可达.若该路径约束可解,则说明该路径是可达的,将会继续探索该路径;反之,则说明该路径不可达,将会结束对该路径的继续探索.

对于图 1 中的示例,通过对 main 函数入口参数进行符号化,符号执行将符号地执行程序.当执行到第 4 行的 *if* 语句时,符号执行将为两个分支各分化出一个状态,并在 true 分支的状态中增加路径约束 x*len == 30,在 false 分支的状态中增加路径约束 x*len != 30.这样随着执行的路径越深,符号执行状态中的路径约束也会越来越复杂,约束求解也会更加耗时.尤其是在执行 12 行的循环语句时,还会导致状态爆炸问题,可能会导致符号执行无法很好地继续深入探索其他路径空间.

1.2 模糊测试

模糊测试(fuzz testing, fuzzing)是一种常用的软件测试技术,最早是由 Millor 等人于 1988 年提出[19][20].其核心思想是将自动或半自动生成的随机数据输入到一个程序中,并监视程序异常,如崩溃、断言(assertion)失败,以发现可能的程序错误,比如内存泄漏.模糊测试常常用于检测软件或计算机系统的安全缺陷.模糊测试是一个自动或半自动的过程, 这个过程包括反复操纵目标软件并为其提供处理数据.模糊测试中的关键是模糊测试用例的生成方法, 用于生成模糊数据的工具可称为模糊器.

AFL(American Fuzzy Lop)[11]是由安全研究员 Michał Zalewski 开发的一款基于覆盖引导(Coverage-guided)的模糊测试工具,是目前应用最为广泛的模糊测试工具之一.AFL 通过在编译时对代码进行插桩,以记录代码覆盖率.然后对输入队列中的测试用例按照特定的策略进行变异,比如位翻转、字节拷贝、字节删除等操作.如果变异后的输入可以增加代码覆盖,则保留该输入到输入队列中.上述过程一直重复进行,直到执行触发到 crash,将报告并记录相应的测试用例.

对于图 1 中的示例,AFL 通过模糊测试将能够在数秒内生成能够覆盖到第 5 行代码的输入,即 if (x*len == 30)的 true 分支.但是对于第 10 行代码,即 if (strcmp(s,"abc") == 0)的 true 分支,AFL 在 10 个小时内都无法生成能够覆盖到该代码的测试输入.

1.3 图神经网络

我们借助图神经网络的变种,即门控图神经网络(Gated graph neural network,缩写为 GGNN).GGNN 是一种基于图神经网络的模型,它采用与门控递归单元(GRU---LSTM 的变种,可以解决远距离依赖)类似的门控机制扩展了这些架构.这允许通过反向传播过程来学习网络.

我们先介绍 GNN.我们定义图 G=(V, E, M),V 为节点,E 为边,M 是向量的集合,即(μ_{v1} , ..., μ_{vm}),其中 μ_{v1} 是实数,用于记录节点 v 在图中的表示.GNN 通过如下传播规则更新每个节点的表示：

$$\mu_v^{(l+1)} = h(\{\mu_u^{(l)}\}_{u\in N^k(v),k\in\{1,2,\ldots,K\}}) \tag{1}$$

具体而言,节点v的新的表示是通过整合计算相邻节点的表示(向量形式)得到的.$N^k(v)$是与 v相连接的,并且边的类型为 k的所有相邻节点.即$N^k(v)=\{u|(u,v,k)\in\mathcal{E}\}\cup\{u|(v,u,k)\in\mathcal{E}\}$.随后,对$\forall v\in\mathcal{V}$,重复该过程 L 次,以更新μ_v为$\mu_v^{(L)}$.多数 GNN 通过为不同类型的边计算单独的表示,再将这些表示合并得到最终的节点表示[12].

$$\mu_v^{(l+1),k} = \sigma_1(\sum_{u\in N^k(v)} \mathbf{W}_2\mu_u^{(l)}), \forall k\in\{1,2,\ldots,K\} \tag{2}$$

$$\mu_v^{(l+1)} = \sigma_2(\mathbf{W}_3[\mu_u^{(l),1},\mu_u^{(l),2},\ldots,\mu_u^{(l),K}]) \tag{3}$$

$$\mu_v^{(0)} = \mathbf{W}_1\mu_v \tag{4}$$

公式 4 表示初始的情况,即 l 为 0 并且μ_v是初始的向量.*W1*,*W2* 和 *W3* 这三个矩阵是需要被学习的变量,σ_1 和 σ_2 是非线性激活函数.为了进一步提高模型的能力,Li 等人[12]提出了门控图神经网络(GGNN),主要的差别在于使用了 Gated Recurrent Units (GRN)[13]作为公式 1 中 h 的实例化.如下两个公式解释了 GGNN 是如何工作的

$$\tilde{m}_v^l = \sum_{u\in N(v)} f(\mu_u^{(l)}) \tag{5}$$

$$\mu_v^{(l+1)} = GRU(\tilde{m}_v^l,\mu_v^{(l)}) \tag{6}$$

为了更新节点v的表示,如公式 5,将会对节点v的所有相邻节点$N(v)$的表示,使用$f(\cdot)$函数(例如线性函数)并求和计算得到消息$\tilde{m}_v^l$.然后,如公式 6,*GRU* 基于$\tilde{m}_v^l$和当前的节点v的表示$\mu_v^{(l)}$来计算下一步新的表示$\mu_v^{(l+1)}$.类似

的,这个传播过程会重复一个特定次数.最后我们使用最后一步得到的节点表示作为这个节点的最终表示.

2 SmartFuSE 方法框架

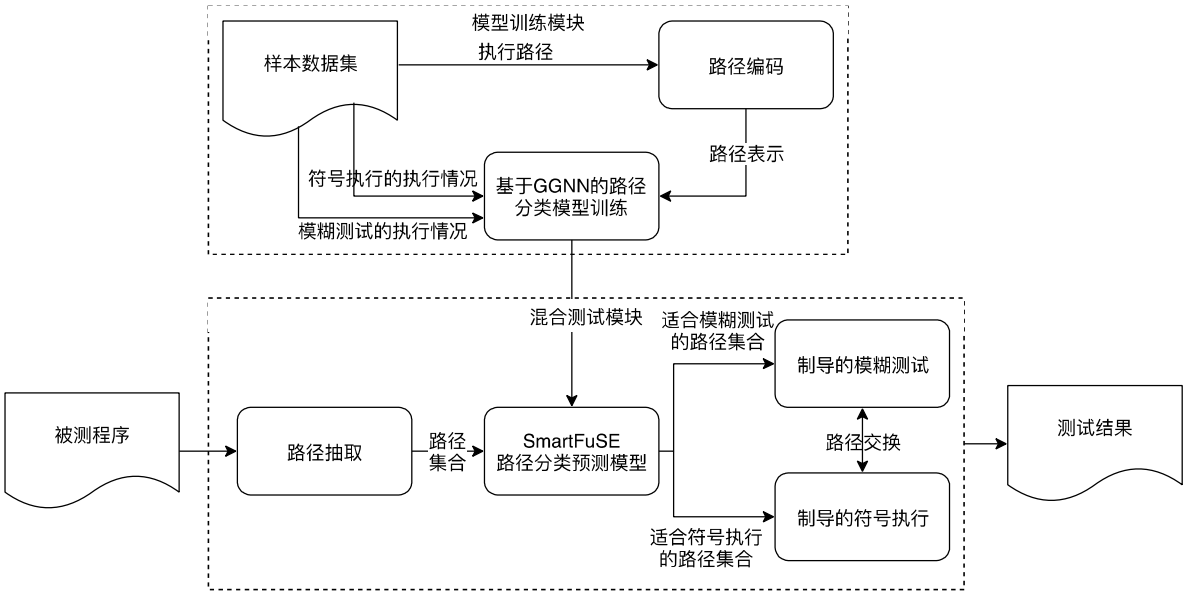

Fig. 2 The overall framework of our method

图 2 方法框架图

SmartFuSE 框架如图 2 所示,主要分为两个部分,模型训练模块和混合测试模块.在模型训练模块,我们先收集一定的样本,样本是路径的集合和到达每条路径时符号执行和模糊测试分别所需的时间(无法到达时时间为无限大).借助已经收集的样本信息,SmartFuSE 先依据路径信息对路径进行编码,用图的方式表示该路径.最后,借助图神经网络模型针对样本进行训练.该模块的输出是一个已经训练完毕的模型.

在混合测试模块,该模块的输入是某个被测程序,针对该被测程序,由于路径数目庞大,我们先只抽取一定深度的路径,然后在更深处的路径中只选取部分路径,以控制需要预测的路径数目.在抽取了路径之后,针对分析出的路径,SmartFuSE 借助模型训练模块得到的模型预测每条路径是符合模糊测试还是符号执行.针对所有适合模糊测试的路径,SmartFuSE 制导模糊测试朝着这些路径执行.如果适合符号执行,那么 SmartFuSE 会制导符号执行优先以这些路径作为目标.

由于预测的不准确性,会导致某些适合模糊测试(符号执行)的方法被传递给符号执行(模糊测试).所以为了进一步提升性能,SmartFuSE 将预测出的适合模糊测试(符号执行)的路径集中,而模糊测试(符号执行)无法覆盖的路径传递给符号执行(模糊测试),即路径交换,以尝试遍历该路径,以此进一步提升覆盖率.

3 模型训练

我们将依次介绍样本数据集收集的方式、路径编码和 GGNN 神经网络.

3.1 数据集

在机器学习中,数据集占据着十分重要的地位,如何有效的生成数据集是一个难点.一般都会使用已有的数据集,既可以提供基本的数据,也可以作为评估实验效果的基准.但是,当前没有发现针对符号执行或模糊测试的数据集.

因此,为了收集用于训练模型的数据,需要在已有的项目上收集、分析数据.我们将收集的数据表示为一个

三元组：<*Path, SE_Time, Fuzz_Time*>,其中 *Path*=($l_1,l_2,...,l_n$)表示程序中的一条路径,$l_i(i \in \{1,...,n\})$表示该路径覆盖的代码行,信息包括代码行所在的文件名以及具体行号.*SE_Time* 是符号执行首次覆盖路径 *Path* 所需要的时间.类似的,*Fuzz_Time* 是模糊测试首次覆盖路径 *Path* 所需要的时间.如果在指定的时间内某个方法没有覆盖,则对应的覆盖时间为正无穷.在无循环的情况下,一个 *Path* 表示的路径能够与其他 *Path* 不相关.但是在有循环的情况下,循环体执行的次数会生成多个 *Path*,这些路径由于只有循环体重复次数不同,具有相关性,并且会大幅度的增加数据集的数据量,从而影响其他数据的比例.因此,为了解决这个问题,我们将会通过一个配置项来限制具有相关性的 *Path* 的数目,默认为 3.

3.2 程序执行路径表示

本文基于抽象语法树(AST),在控制流和数据流之上构造程序路径的表示,并将到达的路径染色,以区分未被覆盖的路径.本文使用图 G=<V, E>表示一条程序路径 *Path* 的路径表示图.每一个节点 V 对应一个 AST 节点,节点的类型包括 FunctionDecl、UnaryOperator、BinaryOperator、IfStmt、ForStmt、WhileStmt 等.每一条边 E 表示节点之间的关系,所有边的类型包括：AST 边、控制流边、数据依赖边和路径标识边.控制流边即控制流图(CFG)中的边.针对控制流边的表示,SmartFuSE 依据执行路径,抽取该路径下的函数调用关系.再依据该调用关系分析每个函数的控制流图.依据函数调用关系连接这些控制流图会得到一个整体的控制流图,该控制流图称为函数间控制流图(ICFG).随后,SmartFuSE 进一步在函数间控制流图上分析数据依赖关系.针对其中的每一个变量,SmartFuSE 都会增加与该变量的数据依赖边.这些依赖关系在函数间控制流图上是通过额外的边表达出来的.除此之外,我们引入路径标识边,用于表示该路径具体执行了哪些语句.如果路径标识边和控制流边同时存在,则只保留路径标识边,用于表示执行路径的染色信息.

程序路径的图表示构造过程如算法 1 所示.首先,基于 AST 构造函数间控制流图 ICFG(行 1),然后在 ICFG 上分析数据依赖关系[64],并加入数据依赖边,构成新的图 DF-ICFG(行 2).之后,除 *Path* 中的 l_1 之外,针对 *Path* 中的每一个行号 *l*,抽取对应的控制流节点 *bb*,其前一行 *prevLine* 的控制流节点为 *prevbb*,将在 DF-ICFG 中增加从 *prevbb* 到 *bb* 的路径标识边(行 4-12).最后,通过 trimGraph 只保留包含路径标识边的 CFG,否则,在 ICFG 中删除对应的 CFG,得到最终的程序路径 *Path* 的图表示 *PathGraph*(行 13).

算法 1： 程序路径的图表示构造算法

```
函数：buildPathGraph(Path, AST)
输入：Path, AST
输出：PathGraph
①   ICFG = buildInterProcCFG(AST);
②   DF-ICFG = addDataFlowEdges(ICFG);
③   prevbb = ∅
④   for each l in Path do
⑤       bb = getCFGNode(l);
⑥       if l is Path.l1 then
⑦           prevbb = bb;
⑧           continue;
⑨       done
⑩       replaceCFGEdgeByPathEdge(DF-ICFG, prevbb, bb);
⑪       prevbb = bb;
⑫   done
⑬   PathGraph = trimGraph(DF-ICFG);
```

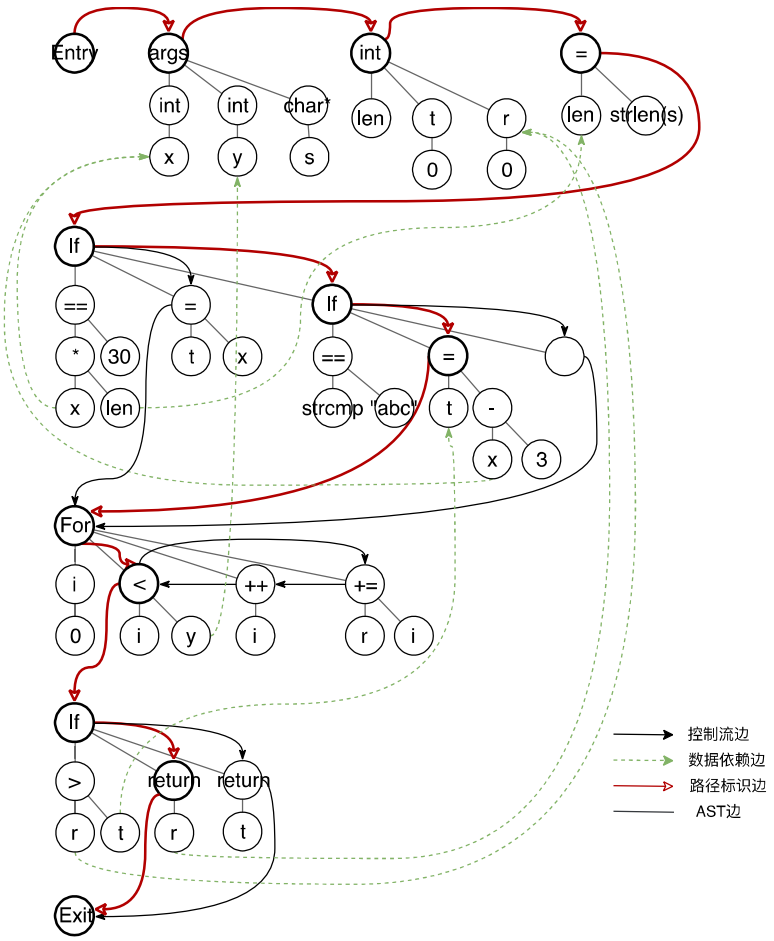

Fig. 3 Example of path representation

图 3 路径表示示例

如图 3 为图 1 中示例程序中的函数 foo 中某一条执行路径(覆盖的代码行为<4,8,9,12,15,16>)的路径表示,其中的节点对应 AST 节点,不带箭头的边为 AST 中的边,实线箭头的边表示控制流边,红色实线空心箭头的边表示路径标识边,绿色虚线箭头的边表示数据依赖边.

4 混合测试

混合测试模块分为三部分：路径抽取和路径分类,制导的符号执行以及制导的模糊测试.

4.1 路径抽取和路径分类

由于程序路径数目往往极大,无法穷尽所有路径.所以为了效率起见,SmartFuSE 选择抽取程序的部分路径.如算法 2 所示,SmartFuSE 依据预先设定的路径深度 *depth*,使用 extractPathsByStmtCov 策略遍历所有到达该深度的路径(行 6-7).extractPathsByStmtCov 表示通过随机选择语句的方式抽取能够达到语句覆盖的路径集合.但是,某个特定深度路径下的路径,它更深的路径信息也会有重要的参考价值.所以,针对每一条到达特定深度的路径,SmartFuSE 分析更深路径下的路径,但会降低抽取路径的要求,以减少抽取的路径数目.

这里采用 extractPathsByBranchCov 策略,即通过随机选择分支的方式抽取能够达到分支覆盖的路径集合(行 9).

通过路径抽取,得到路径集合.然后将按照算法 1,对集合中的每一条路径构造图表示.针对抽取出的路径的

图表示,SmartFuSE 将用训练完毕的模型预测这些图,预测的路径分数以二元组(X,Y)表示,X 表示适合模糊测试的分数,而 Y 表示适合符号执行的分数,两者之和等于 1.我们定义三种路径分类结果,适合模糊测试(X>Y)、适合符号执行(X<Y)和不确定(X≈Y).不确定是指 X 和 Y 的值相差很小(如少于 0.05).如果适合某种方法,则将该路径信息传递给这种方法.如果是不确定,则同时传递给两种方法.我们将分类为适合符号执行的路径构成路径集 SEPathSet,将分类为适合模糊测试的路径构成路径集 FuzzPathSet.

算法 2: 路径抽取算法
函数: extractPaths(AST, depth)
输入: AST, depth
输出: PS(Path set)

```
①  CG = buildCallGraph(AST);
②  topoOrder = sortByTopologicalOrder (CG);
③  pathMap = {}
④  for each fun in topoOrder do
⑤      CFG = buildCFG(fun);
⑥      if CGDepth(fun) <= depth then
⑦          Paths = extractPathsByBranchCov(CFG);
⑧      else
⑨          Paths = extractPathsByStmtCov(CFG);
⑩      done;
⑪      pathMap[fun]=Paths;
⑫  done
⑬  PS = combinePathsByTopoOrder(pathMap);
⑭  saveInFile(PS)
```

4.2 制导的符号执行

在预测出适合符号执行的路径集 SEPathSet 后,SmartFuSE 将制导符号执行优先探索该路径集.其核心思路是在传统的符号执行基础之上,增加路径集作为输入,为与这些目标路径相关的执行状态赋予更高的优先级,以此达到优先探索分析目标路径的目的.

如算法 3 所示,制导的符号执行将路径抽取和路径分类预测得到的适合符号执行的路径集 SEPathSet 作为输入.每次符号执行从状态池中选择一个优先级最高的状态进行执行,如果状态池中的所有状态优先级都相同,则按照符号执行引擎中的搜索策略(比如 DFS、BFS、random path、random state 等)选择状态(GuidedSymbolicExecution⑥). 在执行到分支语句时(ExecuteInstruction①), 拷贝状态赋给两个分支(ExecuteInstruction②-⑤),并更新两个状态的优先级(ExecuteInstruction⑥-⑦).即对于每一个状态,判断其基本块的第一条语句是否在路径集 SEPathSet 中,如果该分支在路径集中,则提升其对应的执行状态的优先级(UpdatePrioriy①②).

算法 3: 符号执行制导算法
函数: GuidedSymbolicExecution()
输入: SEPathSet

```
GuidedSymbolicExecution()
①  ExecutionStatePool = □;
②  AddedStateSet = □;
③  RemoveStateSet = □;
④  ExecutionStatePool.add(initialState);
⑤  while ExecutionStatePool.size > 0 && !TIMEOUT do
⑥      state = selectState(ExecutionStatePool);
⑦      ExecuteInstruction(state);
⑧      UpdateState(ExecutionStatePool, AddedStateSet, RemoveStateSet);
```

```
⑨  done

ExecuteInstruction(ExecutionState state)
①  if state.pc.instructionType == FORK then
②      state2= fork(state);
③      state.pc = state.pc.trueBranchStmt;
④      state2.pc = state.pc.falseBranchStmt;
⑤      AddedStateSet.add(state2);
⑥      UpdatePrioriy(state);
⑦      UpdatePrioriy(state2);
⑧      Other operations in traditional symbolic execution...
⑨  done

UpdatePrioriy (ExecutionState state)
①  if SEPathSet.contains(state.pc.stmtLabel) then
②      state.priority++;
③  done
```

4.3 制导的模糊测试

同样,在预测出适合模糊测试的路径集 FuzzPathSet 后,SmartFuSE 将制导模糊测试优先探索该路径集.其核心思想是,程序中 FuzzPathSet 中的路径经过次数越多的节点,拥有更高的权重;通过对新产生的测试用例经过的路径节点计算总权重,总权重越高,优先级则越高.模糊测试每次选择优先级最高的测试用例作为输入进行下一轮的模糊测试.

具体制导过程如算法 4 所示,制导的模糊测试以目标路径的集合 FuzzPathSet 作为输入.首先,SmartFuSE 将会为程序中每一个节点计算权重(GuidedFuzzing①),将每一个节点的权重初始化为 0,然后对于每一个节点 V,若 FuzzPathSet 经过该节点的路径数目为 N,则节点 V 的权重为 N(initWeight①-⑧).然后为测试输入列表 Seeds 中的每一个种子 seed 计算其优先级(GuidedFuzzing③-④),首先获取原模糊测试工具为该种子计算的优先级,并把该值标准化到[0,1]区间的值(记为 v)作为 FuzzPathSet 中 seed 的初始优先级(computePriority①);然后在 v 的基础上,增加该种子经过路径上的节点的权重之和,作为该种子的最终优先级(computePriority②-⑤).这样可以制导模糊测试优先探索 FuzzPathSet 的路径,当 FuzzPathSet 中的路径探索完之后就会按照原模糊测试工具计算的优先级选择种子进行后续的模糊测试.在计算得到测试输入列表 Seeds 中的每一个种子的优先级后,选取优先级最高的种子 selectedSeed(GuidedFuzzing⑥),对 selectedSeed 进行变异操作得到新的测试输入 newSeed(GuidedFuzzing⑦),并将 selectedSeed 从 Seeds 中移除(GuidedFuzzing⑧).随后执行新的测试输入 newSeed,如果 newSeed 可以覆盖新的代码分支,则将 newSeed 加入到测试输入列表 Seeds 中(GuidedFuzzing 9～11).同时,如果 newSeed 经过的路径在 FuzzPathSet 中,则将该路径上的所有节点的权重减一(GuidedFuzzing).这样可以不再重复制导模糊测试去探索 FuzzPathSet 中已经覆盖过的路径.当程序中所有节点的权重都减少到 0 时,意味着已经完成了 FuzzPathSet 中所有路径的覆盖;此后,根据 computePriority 中的算法,将使用原模糊测试工具计算的优先级选择种子进行后续的模糊测试,即尝试覆盖 SmartFuSE 尚未覆盖到的新的代码和分支.

算法 4：模糊测试制导算法

```
函数：GuidedFuzzing
输入：P, FuzzPathSet, Seeds
①  initWeight (FuzzPathSet, P);
②  while !TIMEOUT or Seeds != □ do
③      for each seed in Seeds do
④          priority[seed] = computePriority(seed);
⑤      done
⑥      selectedSeed = selectHighestPriority(Seeds);
⑦      newSeed = mutate(selectedSeed);
```

```
⑧        Seeds.remove(selectedSeed);
⑨        status = Execute(newSeed);
⑩        if status == coverNewBranch then
⑪            Seeds.add(newSeed);
⑫            path = getCoveredPath(newSeed);
⑬            if path in FuzzPathSet then
⑭                for each node in path do
⑮                    weight[node]--;
⑯                done
⑰            done
⑱        done
⑲    done
initWeight (FuzzPathSet, P)
①    for each node in P do
②        weight[node] = 0;
③    done
④    for each path in FuzzPathSet do
⑤        for each node in path do
⑥            weight[node]++;
⑦        done
⑧    done
computePriority(seed)
①    priority = AFLPiority[seed]; //由原来的模糊测试工具给出,并标准化到[0,1]
②    path = getCoveredPath(seed);
③    for each node in P do
④        priority += weight[node];
⑤    done
⑥    return priority
```

4.4 混合机制

一方面,模型的预测准确率一般都不会是100%的准确率,为了容忍路径分类预测错误的情况,SmartFuSE会在符号执行或者模糊测试无法产生新的分支覆盖的时间超过指定的时间的时候,将没有覆盖的路径从自身的制导路径集中(SEPathSet/FuzzPathSet)移除并传递给另一方,再进行一次制导,以此进一步提升覆盖率.

另一方面,如果仅仅通过路径预测获得适合符号执行(或模糊测试)并制导符号执行(或模糊测试),该制导策略更多地是能够快速提升符号执行(或模糊测试)对适合符号执行(或模糊测试)的代码的路径覆盖.这种制导策略无法应对符号执行和模糊测试都很难覆盖到的路径.因此,我们将符号执行生成的所有测试用例也传递给模糊测试,让模糊测试能够在这些输入的基础上,变异出新的输入,以此尝试覆盖之前符号执行和模糊测试都很难覆盖到的路径.

同时,对于模糊测试已经覆盖到的路径,也被传递给符号执行,在符号执行中再次探索到这些路径时,可以选择不再进行耗时的约束求解,将更多的时间用于探索其他路径.

5 实现与实验

5.1 实现与实验设置

本文实现了混合测试工具SmartFuSE,整体框架主要通过Python实现,其中使用Tensorflow 1.4建立GGNN的模型,在KLEE和AFL的基础之上修改成制导的符号执行和模糊测试方法.

本文选择了模糊测试常用测试对象LAVA-M[15]项目中的程序,即base64、md5sum、uniq和who,这些程序中被插入了大量难以检测的缺陷.我们在LAVA-M程序上测试了工具SmartFuSE的有效性,通过实验尝试回答以下问题:

RQ1: SmartFuSE 中通过路径分类预测模型预测的路径,制导符号执行、模糊测试的效果如何?

RQ2: SmartFuSE 中的路径分类制导和混合机制对模糊测试的优化效果如何?

RQ3: SmartFuSE 与其他混合测试、模糊测试工具相比,检测缺陷的效果如何?

实验中使用的机器信息为：CPU Intel i7-8700K,32G 内存,操作系统为 64 位 Ubuntu 16.04.训练模型时的数据采集自 Coreutils-6.11 中的 date、cat、dd、df、cp、echo 等程序.程序的代码覆盖信息均使用修改后的 afl-cov[18]工具收集.为了展示在训练数据采集中,这些项目的时间设为 KLEE、AFL 各自 24 小时的运行时间.在 LAVA-M 项目中 SmartFuSE、KLEE、AFL 的运行时间也都为 24 个小时.由于已有的混合测试工具 Afleer[2] PANGOLIN[16]等工具不开源,Driller[1]无法获取某些依赖,因此我们在实验中使用 Driller 官方代码库提供的 Python 接口 shellphuzz[17]来代替 Driller 进行实验.

5.2 实验结果

为了验证 SmartFuSE 路径分类预测模型的预测的准确性,我们对收集了 bzip2-1.0.6 在 KLEE、AFL 上分别运行两小时执行的路径,并记录了产生覆盖该路径的测试用例所需的时间,然后与 SmartFuSE 路径分类预测模型预测的路径分类结果进行了比对,发现 SmartFuSE 路径分类预测模型预测的准确率为 84.7%.

5.2.1 LAVA-M 程序上通过路径分类预测模型预测的路径,制导符号执行、模糊测试的代码覆盖能力的比较(RQ1)

如表 1 所示,我们分别统计了在 LAVA-M 四个程序上,KLEE、和制导的 KLEE 的语句覆盖率、分支覆盖率和路径数目.从语句覆盖率来看,制导的 KLEE 相比 KLEE 多覆盖了 1.5%的代码.从分支覆盖率来看,制导的 KLEE 比 KLEE 多覆盖了 2.7%的分支.而在路径覆盖方面,制导的 KLEE 相比 KLEE 增加了 5.5 倍.表中 KLEE 的路径数目我们只统计其生成的测试用例能够覆盖的路径数目,可以看出 KLEE 由于约束求解等操作非常耗时,无法像 AFL 那样仅通过变异就能产生大量测试用例.通过 SmartFuSE 路径分类预测模型预测的适合 KLEE 的路径来制导 KLEE,可以有效增加 KLEE 的代码覆盖率.

如表 2 所示,我们分别统计了在 LAVA-M 四个程序上,AFL、和制导的 AFL 的语句覆盖率、分支覆盖率和路径数目.从语句覆盖率来看,制导的 AFL 相比 AFL 没有显著增加.从分支覆盖率来看,制导的 AFL 比 AFL 多覆盖了 3%的分支.而在路径覆盖方面,制导的 AFL 相比 AFL 增加了 0.4 倍.

5.2.2 SmartFuSE 中的路径分类制导和混合机制对模糊测试的优化效果(RQ2)

如表 3 所示,我们分别统计了在 LAVA-M 四个程序上,KLEE、AFL 和 SmartFuSE 的语句覆盖率、分支覆盖率和路径数目.从语句覆盖率来看,SmartFuSE 相比单独的 KLEE、AFL 增长不是很明显,总体上,SmartFuSE 比 KLEE 多覆盖了 3.4%的代码,比 AFL 多覆盖了 2.8%的代码.从分支覆盖率来看,SmartFuSE 比 KLEE 多覆盖了 26.9%的分支,比 AFL 多覆盖了 20.7%的分支.而在路径覆盖方面,SmartFuSE 则明显比 KLEE、AFL 能够覆盖更多的路径,路径覆盖数目相比 KLEE 增加了 13.5 倍,相比 AFL 增加了 0.9 倍.

Table 1 Coverage Information of KLEE and Guided KLEE on LAVA-M

表 1 LAVA-M 上 KLEE 和制导的 KLEE 的覆盖统计

程序	LOC	语句覆盖率 LCOV(%)		分支覆盖率 BCOV(%)		路径数目	
		KLEE	制导的 KLEE	KLEE	制导的 KLEE	KLEE	制导的 KLEE
base64	113	40.7	53.1	31.7	44.2	27	52
md5sum	416	28.8	37.3	20.8	25.4	1	3
uniq	260	77.7	84.2	64.5	69.2	66	62
who	4073	97.4	97.4	27.2	29.2	31	691
Total	4862	89.2	90.7	33.8	36.5	125	808

Table 2 Coverage Information of AFL and Guided AFL on LAVA-M

表 2 LAVA-M 上 AFL 和制导的 AFL 的覆盖统计

程序	LOC	语句覆盖率 LCOV(%)	分支覆盖率 BCOV(%)	路径数目

		AFL	制导的 AFL	AFL	制导的 AFL	AFL	制导的 AFL
base64	113	39.8	40.7	33.7	34.6	339	439
md5sum	416	68.5	68.5	51.7	51.7	309	368
uniq	260	47.3	47.3	26.5	26.5	106	150
who	4073	96	96.1	29.3	30.2	181	377
Total	4862	89.77	89.82	40.0	40.3	935	1334

Table 3 Coverage Information on LAVA-M

表 3 LAVA-M 上的覆盖统计

程序	LOC	语句覆盖率 LCOV(%)			分支覆盖率 BCOV(%)			路径数目		
		KLEE	AFL	SmartFuSE	KLEE	AFL	SmartFuSE	KLEE	AFL	SmartFuSE
base64	113	40.7	39.8	40.7	31.7	33.7	35.6	27	339	413
md5sum	416	28.8	68.5	69.0	20.8	51.7	30.1	1	309	285
uniq	260	77.7	47.3	77.7	64.5	26.5	64.5	66	106	151
who	4073	97.4	96	97.4	27.2	29.3	61.8	31	181	963
Total	4862	89.2	89.8	92.6	33.8	40.0	60.7	125	935	1812

接下来,进一步分析 SmartFuSE 的缺陷检测能力.如图 4 所示,LAVA-M 的每个程序中,除了程序中已有的缺陷外,还被人工植入了若干缺陷,其中 base64 中植入 44 个、md5sum 中植入 57 个,uniq 中植入 28 个以及 who 中植入 2136 个缺陷.通过对实验结果进行分析可以发现,SmartFuSE 在 LAVA-M 程序集上的缺陷检测能力相比 KLEE 和 AFL 有了很大的提升.单独的 KLEE 执行 24 小时,未能在 LAVA-M 任一程序中发现缺陷.单独的 AFL 执行 24 小时仅能检测 1～2 个缺陷.我们分析这应该是因为：KLEE 虽然有较好的语句覆盖率,但是其仅覆盖了很少数目的路径,所以没有能够检测到缺陷;AFL 由于比较依赖于好的测试输入,实验中仅为 AFL 提供了 LAVA-M 中给出的测试输入,AFL 基于较少的种子难以变异出丰富的测试用例.而通过简单的 AFL+KLEE,即 AFL 和 KLEE 同时执行 24 小时的过程中,将 KLEE 的测试用例传递给 AFL,可以在 who 程序中检测到 74 个缺陷.这也说明了 KLEE 能够覆盖 AFL 不同的代码空间,将 KLEE 的测试用例传递给 AFL,可以帮助 AFL 探索新的代码空间.但是 AFL+KLEE 的检测缺陷的能力还是相对较差.SmartFuSE(unguided)通过进一步在 AFL+KLEE 的基础上引入混合机制进行优化,让 AFL 优先变异 KLEE 传递过来具有新的分支覆盖的种子,让 KLEE 对 AFL 已经覆盖的路径不再约束求解产生重复的测试用例.从图 4 可以看出 SmartFuSE(unguided)相比简单的 AFL+KLEE 可以多检测到五百多个缺陷.这也显示了本文的混合机制可以优化 AFL 和 KLEE 的执行效果.进一步地,SmartFuSE 在 SmartFuSE(unguided)的基础上,增加了通过路径分类预测模型预测的路径,制导符号执行、模糊测试,同时针对预测错误的情况的处理等.从图 4 可以发现 SmartFuSE 执行 24 小时,在 base64 中不仅发现了人工植入的所有 44 个缺陷,还另外检测到了 11 个缺陷;在 uniq 中发现了 18 个缺陷;在 who 中发现了 856 个缺陷,总共发现了 929 个缺陷.md5sum 程序中,SmartFuSE 未能检测到任何缺陷,我们调查发现这是由于该程序中的哈希运算导致的[2].总之,通过图 4 可以看出本文提出的路径分类制导和混合机制可以帮助 KLEE 和 AFL 发现更过的程序缺陷.

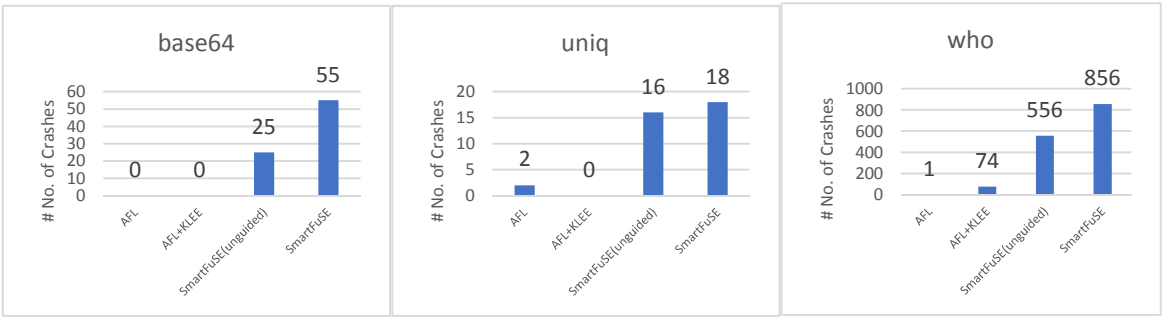

Fig. 4 The Number of Crashes Detected by SmartFuSE, AFL and KLEE on LAVA-M

图 4 SmartFuSE, AFL and KLEE 在 LAVA-M 程序集上检测到的缺陷数目结果

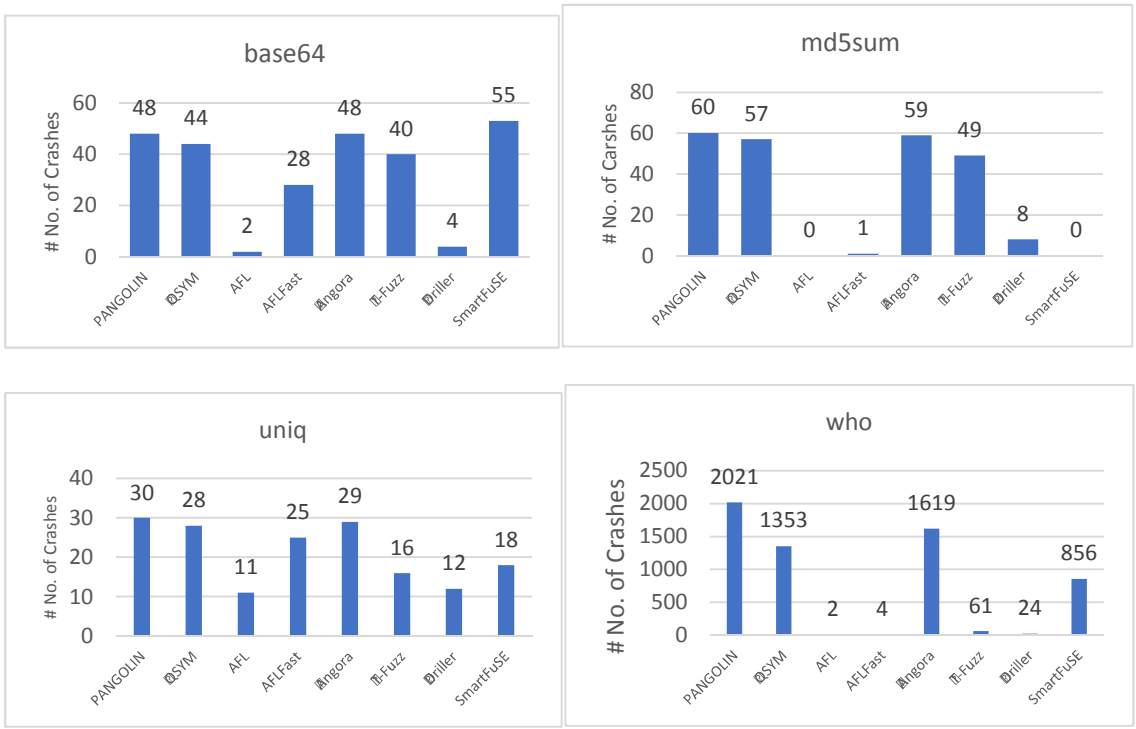

Fig. 5 The Number of Crashes Detected by Several Fuzzing and Hybrid Tools on LAVA-M

图 5 几个模糊测试和混合测试工具在 LAVA-M 程序集上检测到的缺陷数目结果

5.2.3 SmartFuSE 与其他混合测试、模糊测试工具的比较(RQ3)

由于我们在使用 Driller(shellphuzz)过程中出现了部分执行错误,可能会影响其测试结果,因此,我们还参考 PANGOLIN[16]工作(S&P 2020 录用)中的实验结果,对比了 SmartFuSE 和已有的一些混合测试和模糊测试工具的效果.此处 AFL 的实验结果使用两个 AFL 进程完成,而上文中的 AFL 实验结果使用一个 AFL 进程完成.如图 5 所示,展示了混合模糊测试工具 PANGOLIN、QSYM[5]、Driller 和 SmartFuSE,以及模糊测试工具 AFL、AFLFast[54]、Angora[59]和 T-Fuzz[60]在 24 小时内在 LAVA-M 程序集上检测到的缺陷数目.从图 5 中可以看出,在 base64 程序中 SmartFuSE 发现的缺陷数目最多,而在另外三个程序中 SmartFuSE 检测到的缺陷数目处在中等水平.在四个项目中,SmartFuSE 都能够比 AFL、Driller 检测到更多的缺陷.我们将在相关工作中讨论这几个工具的特性.

5.3 实验讨论

通过对实验结果进行分析可以发现,SmartFuSE 在 LAVA-M 程序集上的缺陷检测能力相比 KLEE、AFL 有了很大的提升.我们进一步对 SmartFuSE 模糊测试中产生的能够触发缺陷的测试用例进行了分析,发现有很多测试用例正是在 SmartFuSE 符号执行模块产生的测试用例的基础上不断进行变异操作后得到的新的测试用例.同时,SmartFuSE 通过制导符号执行,可以使其更快地探索适合符号执行的路径,并生成更多的测试用例;符号执行阶段为模糊测试阶段提供更多测试用例,帮助模糊测试突破其不太容易覆盖的路径,生成具有更高代码覆盖的测试用例,进而检测到更多的程序缺陷.这也证明了我们的基于符号执行的测试和模糊测试的混合机制确实可以帮助 SmartFuSE 更快地覆盖到符号执行和模糊测试都很难覆盖到的程序路径,使得 SmartFuSE 工具的

缺陷检测能力相比单独的符号执行或者模糊测试都更强、更有效.

可能影响工具效果的因素：(1) 数据集的丰富性,将会影响 SmartFuSE 路径分类预测模型的准确性,也将影响工具的实用性.训练神经网络需要大量的数据集,而目前没有相关的已有的数据集能够直接供我们使用.所以在实验中为了降低对数据集数目的要求,我们选取中小规模的程序去生成数据集.因为这些程序已经被 KLEE 广泛使用,KLEE 容易生成测试用例.相比之下,大规模程序中,KLEE 执行这些程序并求解约束生成测试用例所需的时间和内存开销极大,短时间内无法收集足够的数据集,这也是我们后续工作中需要解决的问题.另外,预测模型可能在训练集程序上有较好的预测准确性,但在其他程序上没有满意的预测效果.(2)SmartFuSE 路径分类预测模型的准确性,将会影响 SmartFuSE 混合测试的执行效率.模型的预测准确率一般都不会是 100%的准确率.路径分类预测错误,将会导致 SmartFuSE 制导符号执行(模糊测试)模块去探索并不适合该模块的路径.为了降低错误的路径分类预测带来的影响,我们通过设置制导的时间阈值,当一定时间无法制导符号执行(模糊测试)覆盖该路径时,将尝试让模糊测试(符号执行)来覆盖该路径.(3)时间阈值的设置会影响执行的效果.过大会导致错误预测的尝试时间过长;过小会导致正确的预测(如预测为适合符号执行)被不适合的方法(如模糊测试)尝试,从而无法覆盖对应路径.

6 相关工作

我们主要从符号执行、模糊测试以及符号执行与模糊测试结合的混合测试技术三个方面介绍相关工作.

6.1 符号执行

符号执行技术是一种常用的软件分析技术,但是传统的符号执行方法面临着路径爆炸、环境交互、系统调用和约束求解等问题[21],因此有许多针对符号执行的优化技术被提出.

为了应对环境交互和系统调用等问题,KLEE[10]、AEG[22]都为系统调用建立了抽象模型,以支持符号化文件、套接字等作为符号执行的输入.而选择性符号执行(Selective Symbolic Execution)比如 S2E[23]基于 KLEE 和 QEMU 虚拟机,通过将具体值的单路径执行和符号值的多路径执行同时进行,以支持在程序中目标代码进行符号执行,在调用系统内核函数时进行具体执行,以实现符号执行过程与具体执行过程之间的无缝切换.

为了应对路径爆炸问题,许多研究工作提出了不同的解决方法.一类工作提出了路径选择的优化算法[10]错误！未找到引用源。[24][25][26][27],即符号执行搜索策略.例如,KLEE[10]中就提供了多种路径搜索策略,包括深度优先搜索策略(DFS)、广度优先搜索策略(BFS)、随机状态搜索策略、随机路径搜索策略等.一类工作提出使用函数摘要技术[28][29][30]和循环摘要技术[31][32],通过复用已经探索过的代码的执行信息,避免对同一代码的重复执行,来提高符号执行效率.还有一类工作提出通过状态合并[33][34],即将几条路径合并成一个状态,来有效的减少需要遍历的路径.但是状态合并会导致路径约束更加复杂,增大约束求解的难度.还有一些工作通过与其他程序分析技术相结合[35][36][37][38][39],比如程序切片、污点分析、类型检查和编译优化等技术,修剪掉不感兴趣的路径,使得符号执行着重分析更容易发现程序错误和缺陷的代码空间.

对于约束求解问题,Z3[40]作为当前主流的 SMT 约束求解器,在符号执行引擎中,比如 KLEE[10]、Mayhem[36]、Angr[41],得到了广泛应用.同时 EXE[42]、KLEE[10]、AEG 错误！未找到引用源。还使用了 STP[43]约束求解器.同时符号执行引擎中也针对约束求解进行了优化设计.比如 EXE、KLEE、S2E 中通过对收集到的约束进行化简来降低约束的复杂度.EXE、KLEE、Memoise[44]、GreenTrie[45]等符号执行引擎中还会对约束求解的结果进行缓存从而减少对求解器的调用.

我们的方法通过将基于符号执行的测试与模糊测试结合,也可以帮助应对符号执行面临的环境交互和约束求解等问题.

6.2 模糊测试

模糊测试技术有基于白盒、基于黑盒和基于灰盒之分[46].黑盒模糊测试不会分析程序本身,而是基于预设的规则来变异和生成新的种子,比如基于遗传算法的模糊测试通过位翻转、字节拷贝、字节删除等操作变异出

新的输入;基于语法的模糊测试则依据语法规则自动生成新的种子[47].相比之下,白盒模糊测试会分析程序本身的信息,包括控制流,数据流等.白盒模糊测试最早是由 Godefroid 等人提出来的[48][49],该方法提出的工具 SAGE 通过在实际执行具体输入时收集路径约束,然后基于代码覆盖最大化准则,将约束系统地取反后求解,生成新的测试输入,继续进行模糊测试.灰盒模糊测试介于黑盒模糊测试和白盒模糊测试之间,该技术常常用到代码插桩技术[50][51]、污点分析技术[52]等.

为了提升模糊测试的代码覆盖能力和执行效率,许多研究工作提出了改进方法.BuzzFuzz[53]使用动态污点分析识别输入中与目标缺陷点相关的部分,并变异生成新的测试输入.AFLFast[54]通过构建马尔可夫链模型评估种子,为能够覆盖到低频路径的测试输入赋予更高的优先级和更长的变异时间.为了使得模糊测试更快覆盖到给定的目标位置,AFLgo[14]提出制导的模糊测试,通过计算种子到目标点的距离,然后使用马尔可夫链蒙特卡罗(MCMC)优化技术选取离目标点更近的种子赋予更高的优先级和更长的变异时间.Vuzzer[55]通过动态污点分析定位出输入中的魔法字节(magic bytes),然后对魔法字节进行变异来生成新的输入.Steelix[56]则提出使用轻量级的静态分析和二进制插桩来定位输入中的魔法字节.Neuzz[57]使用神经网络模型学习出程序中分支行为的平滑近似,并基于梯度制导策略对测试用例进行变异,从而平滑模糊测试搜索过程.Skyfire[58]提出了一种数据驱动的种子生成方法,该方法根据已有的大量测试输入学习出输入的语法和语义信息,据此可以为具有高结构化输入的被测程序生成模糊测试的输入.为了不使用符号执行就能对路径约束进行求解,Angora[59]提出了字节级污点分析,并使用梯度下降方法来快速找到满足复杂路径约束的解,从而为模糊测试得到新的输入.T-Fuzz[60]通过对 TTCN-3 模型进行建模,提出针对通信协议的模糊测试工具.

6.3 基于符号执行的测试与模糊测试结合的混合测试

Driller[1]针对二进制代码,在模糊测试(AFL)的基础上结合了混合执行引擎 Angr,在模糊测试遇到一些特殊边界条件(比如满足特定输入的窄路径)无法继续探索时,将会启用二进制混合测试模块(Angr).该模块通过约束求解生成能够突破这些限制的新输入,并将输入返回给模糊测试作为种子,使得模糊测试可以继续执行.DigFuzz[3]提出使用蒙托卡罗概率模型对模糊测试中执行到的路径计算优先级,将更加困难的路径交给混合执行来求解.Pak 等人[4]提出的混合模糊测试方法是首先使用符号执行探索程序的空间,然后将得到的测试输入中,路径关键部分的字节保持不变,其他部分的字节进行随机变异来生成新的输入,再交给模糊测试进行探索.与之类似地,QSYM[5]通过对混合测试中的部分路径约束进行求解生成测试用例作为基础种子,然后在这些种子的基础上进行变异来生成满足需求的测试输入.SYMFUZZ[6]首先通过符号执行,对黑盒模糊测试的两个输入种子的符号执行轨迹进行分析,得到输入位之间的依赖关系,然后基于该依赖关系,计算种子的最佳突变率,然后将新生成的种子作为模糊测试新的输入.Munch[7]提出了较为粗粒度的混合模糊测试方法,即为了提高函数覆盖率,首先使用模糊测试运行程序,然后使用制导符号执行对模糊测试未覆盖到的函数探索,产生新的测试输入继续模糊测试.Afleer[2]提出了更细粒度的混合模糊测试方法,通过对源码插桩生成插桩后的 LLVM 中间码和插桩后的可执行文件,然后同时运行模糊测试和符号执行.模糊测试不断的生成新的测试用例,并更新覆盖信息;符号执行则系统地探索程序状态空间,为未被覆盖的分支生成测试用例,作为模糊测试的种子.该方法中符号执行不可避免地会去探索本身并不适合该工具的路径.相比之下,SmartFuSE 还通过对路径进行分类预测,进而对符号执行和模糊测试过程进行制导,让它们各自优先探索适合它们的路径.这样可以使符号执行在前期分配更多的时间探索和求解适合其执行的路径.同时对预测出的以及一定时间执行后发现不适合符号执行的路径,也会制导模糊测试优先探索,从而,进一步提高 SmartFuSE 所能覆盖的路径.PANGOLIN[16]提出增量式模糊测试方法,通过对未覆盖的路径使用多面体进行路径抽象,可以制导种子变异和混合测试过程,在多项式时间内生成大量测试输入.SeededFuzz[61]使用静态分析、动态监测和符号执行为模糊测试生成和选择合适的种子.Berry[62]中提出的序列制导模糊测试(SDHF),在给定一个程序的语句序列时,该方法利用序列制导策略和混合执行来增强模糊测试的能力.ILF[63]提出了一个基于模仿学习的模糊测试方法(Imitation Learning based Fuzzer),通过神经网络模型对基于覆盖的符号执行产生的高质量的测试用例进行学习,从而学习得到模糊测试生成种子的策略.该方法只是将符号执行作为学习对象来提高模糊测试种子生成的质量,而我们的方法是通过

深度学习为符号执行和模糊测试分配各自更适合执行路径来完成混合测试.目前这些已有的符号执行与模糊测试结合的混合测试方法都没有使用深度学习来为两者分配合适的路径并以此制导符号执行与模糊测试来进行混合测试.

7 总结

本文提出了一个基于深度学习将基于符号执行的测试与模糊测试相结合的混合测试方法,并基于符号执行工具 KLEE 和模糊测试工具 AFL,实现了一个基于符号执行的测试与模糊测试的混合测试工具 SmartFuSE.我们通过对 LAVA-M 程序集中的几个程序进行测试,结果表明我们的工具 SmartFuSE 相对于单独通过模糊测试或符号执行的方法,不仅能够提升对代码的覆盖率,并且对缺陷的检测能力也得到了显著提升.

尽管我们通过深度学习的方法探索出了基于符号执行的测试与模糊测试对代码空间的探索具有不同的偏好,但是并没有提供出符号执行与模糊测试到底各自更适合的代码空间具有怎样的特征.我们考虑可以将符号执行与模糊测试到底各自更适合的代码特征总结出来,可以为未来的基于符号执行的测试、模糊测试以及它们的混合测试方法的优化提供参考.

References:

[1] Stephens N, Grosen J, Salls C, et al. Driller: Augmenting Fuzzing Through Selective Symbolic Execution. In: Proc. of the Network and Distributed System Security Symp. (NDSS). 2016.

[2] Xie X, Li X, etc. Branch Coverage-Guided Hybrid Testing Based on Symbolic Execution and Fuzzing. Journal of Software, 2019, 30(10):3071-3089 (in Chinese with English abstract).

[3] Zhao L, Duan Y, Yin H, Xuan J. Send Hardest Problems My Way: Probabilistic Path Prioritization for Hybrid Fuzzing. In: Proc. of the Network and Distributed System Security Symp. (NDSS). 2019.

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

[5] 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). 2018. 745-761.

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

[7] 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. 2018. 1475-1482.

[8] Xujie Si, Hanjun Dai, Mukund Raghothaman, Mayur Naik, and Le Song. Learning loop invariants for program verification. In: Proc. of Advances in Neural Information Processing Systems (NeurIPS). 2018. 7751–7762.

[9] King JC. Symbolic execution and program testing. Communications of the ACM. 1976 Jul 1;19(7):385-94.

[10] Cadar C, Dunbar D, Engler DR. Klee: unassisted and automatic generation of high-coverage tests for complex systems programs. In: Proc. of USENIX Symp. on Operating Systems Design and Implementations (OSDI). 2008. 209-224.

[11] American fuzzy lop. Online. 2020.09. http://lcamtuf.coredump.cx/afl/.

[12] Yujia Li, Daniel Tarlow, Marc Brockschmidt, and Richard Zemel. Gated graph sequence neural networks. arXiv preprint arXiv:1511.05493. 2015.

[13] Kyunghyun Cho, Bart Van Merriënboer, Dzmitry Bahdanau, and Yoshua Bengio. On the properties of neural machine translation: Encoder-decoder approaches. arXiv preprint arXiv:1409.1259. 2014.

[14] Böhme M, Pham VT, Nguyen MD, Roychoudhury A. Directed greybox fuzzing. In: Proc. of the ACM SIGSAC Conf. on Computer and Communications Security (CCS). 2017. 2329-2344.

[15] Dolan-Gavitt B, Hulin P, Kirda E, 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 (SP). IEEE 2016. 110-121.

[16] Huang H, Yao P, Wu R, Shi Q, Zhang C. PANGOLIN: Incremental Hybrid Fuzzing with Polyhedral Path Abstraction. In IEEE Symposium on Security and Privacy (SP). 2020 (Accepted). Online. https://rainoftime.github.io/files/HybridFuzz-Preprint.pdf

[17] Shellphuzz. Online. 2020.09. https://github.com/shellphish/fuzzer.

[18] AFL Cov. Online. 2020.09. http://cipherdyne.com/afl-cov/.

[19] Miller BP, Fredriksen L, So B. An empirical study of the reliability of UNIX utilities. Communications of the ACM. 1990 Dec 1;33(12):32-44.

[20] McNally R, Yiu K, Grove D, Gerhardy D. Fuzzing: the state of the art. DEFENCE SCIENCE AND TECHNOLOGY ORGANISATION EDINBURGH (AUSTRALIA); 2012 Feb 1.

[21] Baldoni R, Coppa E, D'elia DC, Demetrescu C, Finocchi I. A survey of symbolic execution techniques. ACM Computing Surveys (CSUR). 2018 May 23;51(3):1-39.

[22] Avgerinos T, Cha SK, Rebert A, Schwartz EJ, Woo M, Brumley D. Automatic exploit generation. Communications of the ACM. 2014 Feb 1;57(2):74-84.

[23] Chipounov V, Kuznetsov V, Candea G. The S2E platform: Design, implementation, and applications. ACM Trans. on Computer Systems (TOCS). 2012 Feb 1;30(1):1-49.

[24] Li Y, Su Z, Wang L, Li X. Steering symbolic execution to less traveled paths. ACM SigPlan Notices. 2013 Oct 29;48(10):19-32.

[25] Ma KK, Phang KY, Foster JS, Hicks M. Directed symbolic execution. In: Proc. of the Int'l Static Analysis Symp. Springer 2011. 95-111.

[26] Zhang Y, Chen Z, Wang J, Dong W, Liu Z. Regular property guided dynamic symbolic execution. In: Proc. of the IEEE/ACM 37th IEEE Int'l Conf. on Software Engineering (ICSE). IEEE 2015. 643-653.

[27] Xie T, Tillmann N, De Halleux J, Schulte W. Fitness-guided path exploration in dynamic symbolic execution. In: Proc. of the IEEE/IFIP Int'l Conf. on Dependable Systems & Networks (DSN). IEEE 2009. 359-368.

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

[29] Anand S, Godefroid P, Tillmann N. Demand-driven compositional symbolic execution. In: Proc. of Int'l Conf. on Tools and Algorithms for the Construction and Analysis of Systems. Springer 2008. 367-381.

[30] Boonstoppel P, Cadar C, Engler D. RWset: Attacking path explosion in constraint-based test generation. In: Proc. of Int'l Conf. on Tools and Algorithms for the Construction and Analysis of Systems. Springer 2008. 351-366.

[31] Godefroid P, Luchaup D. Automatic partial loop summarization in dynamic test generation. In: Proc. of the Int'l Symp. on Software Testing and Analysis (ISSTA). 2011. 23-33.

[32] Xie X, Chen B, Liu Y, Le W, Li X. Proteus: Computing disjunctive loop summary via path dependency analysis. In: Proc. of the 24th ACM SIGSOFT Int'l Symp. on Foundations of Software Engineering (FSE). 2016. 61-72.

[33] Hansen T, Schachte P, Søndergaard H. State joining and splitting for the symbolic execution of binaries. In: Proc. of Int'l Workshop on Runtime Verification. Springer 2009. 76-92.

[34] Kuznetsov V, Kinder J, Bucur S, Candea G. Efficient state merging in symbolic execution. Acm Sigplan Notices. 2012 Jun 11;47(6):193-204.

[35] Shoshitaishvili Y, Wang R, Hauser C, Kruegel C, Vigna G. Firmalice-automatic detection of authentication bypass vulnerabilities in binary firmware. In: Proc. of the Network and Distributed System Security Symp. (NDSS). 2015.

[36] Cha SK, Avgerinos T, Rebert A, Brumley D. Unleashing mayhem on binary code. In: Proc. of the IEEE Symp. on Security and Privacy. IEEE 2012. 380-394.

[37] Khoo YP, Chang BY, Foster JS. Mixing type checking and symbolic execution. ACM Sigplan Notices. 2010 Jun 5;45(6):436-47.

[38] Gao F, Wang L, Li X. BovInspector: automatic inspection and repair of buffer overflow vulnerabilities. In: Proc. of the 31st IEEE/ACM Int'l Conf. on Automated Software Engineering (ASE). IEEE 2016. 786-791.

[39] Wang G, Chattopadhyay S, Biswas AK, Mitra T, Roychoudhury A. Kleespectre: Detecting information leakage through speculative cache attacks via symbolic execution. ACM Trans. on Software Engineering and Methodology (TOSEM). 2020 Jun 1;29(3):1-31.

[40] 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 2008. 337-340.

[41] Shoshitaishvili Y, Wang R, Salls C, Stephens N, Polino M, Dutcher A, Grosen J, Feng S, Hauser C, Kruegel C, Vigna G. Sok:(state of) the art of war: Offensive techniques in binary analysis. In: Proc. of the IEEE Symp. on Security and Privacy (SP). IEEE 2016. 138-157.

[42] Cadar C, Ganesh V, Pawlowski PM, Dill DL, Engler DR. EXE: Automatically generating inputs of death. In: Proc. of the 13th ACM Conf. on Computer and Communications Security (CCS). 2006.

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

[44] Yang G, Păsăreanu CS, Khurshid S. Memoized symbolic execution. In: Proc. of the Int'l Symp. on Software Testing and Analysis (ISSTA). 2012. 144-154.

[45] Jia X, Ghezzi C, Ying S. Enhancing reuse of constraint solutions to improve symbolic execution. In: Proc. of the Int'l Symp. on Software Testing and Analysis (ISSTA). 2015. 177-187.

[46] Liang H, Pei X, Jia X, Shen W, Zhang J. Fuzzing: State of the art. IEEE Trans. on Reliability. 2018 Jun 4;67(3):1199-218.

[47] Kim SY, Cha S, Bae DH. Automatic and lightweight grammar generation for fuzz testing. Computers & Security. 2013 Jul 1; 36:1-1.

[48] Godefroid P, Levin MY, Molnar DA. Automated Whitebox Fuzz Testing. In: Proc. of the Network and Distributed System Security Symp. (NDSS). 2008. 151-166.

[49] Godefroid P, Levin MY, Molnar D. SAGE: whitebox fuzzing for security testing. Queue. 2012 Jan 11;10(1):20-7.

[50] Luk CK, Cohn R, Muth R, Patil H, Klauser A, Lowney G, Wallace S, Reddi VJ, Hazelwood K. Pin: building customized program analysis tools with dynamic instrumentation. Acm sigplan notices. 2005 Jun 12;40(6):190-200.

[51] Nethercote N, Seward J. Valgrind: a framework for heavyweight dynamic binary instrumentation. ACM Sigplan notices. 2007 Jun 10;42(6):89-100.

[52] Bekrar S, Bekrar C, Groz R, Mounier L. Finding software vulnerabilities by smart fuzzing. In: Proc. of the Fourth IEEE Int'l Conf. on Software Testing, Verification and Validation. IEEE 2011. 427-430.

[53] Fayaz SK, Yu T, Tobioka Y, Chaki S, Sekar V. {BUZZ}: Testing Context-Dependent Policies in Stateful Networks. In: Proc. of the 13th {USENIX} Symp. on Networked Systems Design and Implementation ({NSDI} 16). 2016. 275-289.

[54] Böhme M, Pham VT, Roychoudhury A. Coverage-based greybox fuzzing as markov chain. IEEE Trans. on Software Engineering. 2017 Dec 21;45(5):489-506.

[55] 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. 1-14.

[56] Li Y, Chen B, 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. 2017. 627-637.

[57] She D, Pei K, Epstein D, Yang J, Ray B, Jana S. NEUZZ: Efficient fuzzing with neural program smoothing. In: Proc. of the IEEE Symp. on Security and Privacy (SP). IEEE 2019. 803-817.

[58] Wang J, Chen B, Wei L, Liu Y. Skyfire: Data-driven seed generation for fuzzing. In: Proc. of the IEEE Symp. on Security and Privacy (SP). IEEE 2017. 579-594.

[59] Chen P, Chen H. Angora: Efficient fuzzing by principled search. In: Proc. of the IEEE Symp. on Security and Privacy (SP). IEEE 2018. 711-725.

[60] Johansson W, Svensson M, Larson UE, Almgren M, Gulisano V. T-Fuzz: Model-based fuzzing for robustness testing of telecommunication protocols. In: Proc. of the IEEE Seventh Int'l Conf. on Software Testing, Verification and Validation. IEEE 2014. 323-332.

[61] Wang W, Sun H, Zeng Q. SeededFuzz: Selecting and Generating Seeds for Directed Fuzzing. In: Proc. of the Int'l Symp. on Theoretical Aspects of Software Engineering. IEEE, 2016.

[62] Liang H, Jiang L, Ai L, Wei J. Sequence Directed Hybrid Fuzzing. In: Proc. of the IEEE 27th Int'l Conf. on Software Analysis, Evolution and Reengineering (SANER). IEEE 2020. 127-137.

[63] He J, Balunović M, Ambroladze N, Tsankov P, Vechev M. Learning to fuzz from symbolic execution with application to smart contracts. In: Proc. of the ACM SIGSAC Conf. on Computer and Communications Security (CCS). 2019. 531-548.

[64] Ferrante, J., Ottenstein, K. J., Warren, J. D. The program dependence graph and its use in optimization. ACM Trans. on Programming Languages and Systems (TOPLAS). ACM 1987. 319-349.

附中文参考文献:

[2] 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.