基于参数约束的分支覆盖符号执行优化算法*

於家伟[1],李世明[1,2],毕雪洁[1],李秋月[1],高胜花[1]
(1.哈尔滨师范大学 计算机科学与信息工程学院,黑龙江 哈尔滨 150025;
2.上海市信息安全综合管理技术研究重点实验室,上海 200240)

摘　要:软件质量检测常用的方法是软件测试,符号执行作为主流的测试技术已被广泛应用于学术界与工业界中。但是随着程序规模的增大和函数调用的增加,因某些路径约束条件的特殊性,而难以生成正确的测试用例,从而导致符号执行不能对所有路径做到全覆盖。为了提高符号执行在特殊约束条件对路径的覆盖率等问题,本文提出了基于参数约束的符号执行优化算法。首先,该算法通过搜索收集程序代码中函数的特殊参数,然后利用这些特殊参数作为约束条件,最后将约束条件添加到路径的约束集中。该算法使符号执行生成的测试用例更加精确,从而实现覆盖特殊约束条件下的路径分支,以提高符号执行的精确性和路径覆盖率。在开源符号执行平台 CREST 中实验并验证上述优化算法,验证及测试结果表明本文提出的算法能够提高符号执行在特殊约束条件下对路径的覆盖率。

关键词:符号执行;参数约束;测试用例;软件测试
中图分类号:TP311　　**文献标识码**:A　　DOI:10.19358/j.issn.2096-5133.2020.01.003
引用格式:於家伟.基于参数约束的分支覆盖符号执行优化算法[J].信息技术与网络安全,2020,39(1):14-18.

Optimization of branch covering symbol execution based on constraints

Yu Jiawei[1],Li Shiming[1,2],Bi Xuejie[1],Li Qiuyue[1],Gao Shenghua[1]
(1.College of Computer Science and Information Engineering,Harbin Normal University,Harbin 150025,China;
2.Shanghai Key Laboratory of Information Security Management Technology Research,Shanghai 200240,China)

Abstract:A common method for software quality inspection is software testing. Symbol execution as a mainstream testing technology has been widely used in academia and industry. However,with the increase in program size and function calls,due to the special nature of certain path constraints,it is difficult to generate correct test cases,which results in symbolic execution not being able to cover all paths. In order to improve the problem of symbol execution on the path coverage under special constraints,this paper proposes a symbolic execution optimization algorithm based on parameter constraints. Firstly,the algorithm collects special parameters of functions in the program code by searching,then uses these special parameters as constraints,and finally adds constraints to the constraint set of the path. This algorithm makes the test cases generated by symbol execution more accurate,so as to achieve path branch coverage under special constraints,to improve the accuracy of symbol execution and path coverage. Experiment is carried out to verify the above optimization algorithm in the open source symbol execution platform CREST. The results of verification and testing show that the algorithm proposed in this paper can improve the path coverage of symbol execution under special constraints.

Key words: symbolic execution;parameter constraint;test case;software test

0　引言

软件漏洞是软件中潜藏的代码缺陷,通过提高检测代码的覆盖率可以提高漏洞的发现概率[1],而生成高覆盖率的测试用例进行检测漏洞时,若代码执行通过率高时,可认为该程序在一定程度上是可靠的[2]。

作为一种程序测试技术[3],符号执行在软件测试、程序缺陷挖掘和测试用例生成中得到广泛的研究和应用,其程序变量是以抽象符号形式来通过符号模拟程序运行并搜集路径上的约束条件。此外,根据程序的语义、遍历程序的路径空间也可用来检测程序是否满足一定的安全特性。

*基金项目:上海市信息安全管理技术研究重点实验室开放课题(AGK2015003)

输入约束[4]作为符号执行优化的方法之一,近年来业界已取得了一定成果。TRABISH D 等人[5]结合静态分析和符号切片技术使符号执行能够搜索到更重要的路径;GODEFROID P 等人[6]提出以调用该执行函数生成的摘要作为约束条件来减少代码的重复执行;RAMOS D A 等人[7]将约束条件引入到 KLEE 中,通过检查被测程序的单个功能而不是整个程序,提高了效率;WONG E 等人[8]提出了基于文档辅助的建模方法,通过自然语言处理和试探法生成文档并提取约束条件;郭曦等人[9]通过分析路径逻辑表达式和提取共享表达式来提高状态合并的效率;安靖等人[10]通过生成外部调用函数摘要来避免因多次测试外部调用而引起的路径爆炸问题。

上述关于符号执行的优化方法在一定程度上都能够减少路径分支并提高代码覆盖率。但是,上述各方法注重符号执行过程中的路径发现问题,而非测试用例精确覆盖路径问题,故对于特殊参数情况下的测试用例精确生成存在不足。同时,程序中因调用大量外部环境函数且自身参数存在特殊性,故符号执行生成的测试用例导致某些路径尽管被发现了却无法被执行,进而降低了符号执行的准确性和路径覆盖率。

综上所述,若将收集到的特殊参数转换成约束条件并添加到约束集中,则生成的测试用例质量会得到提高。本文基于上述思想,提出了基于参数约束的分支覆盖符号执行优化算法,可用于优化特殊测试用例以提高路径覆盖率。

1 传统符号执行

传统符号执行的优点之一是可以检测复杂路径,在收集路径上的约束条件后通过约束求解器生成满足条件的测试用例,从而检测了该条路径,示例代码如图 1 所示。

```
1. test(int p1, int p2, int p3)
2. {
3.       p1=0, p2=0, p3=0;
4.       if(x>0)
5.             {p1=-2;}
6.       if(y<5)
7.       {
8.            if(y+z>0)
9.                 {p2=1;}
10.                p3=2;
11.      }
12.      if(p1+p2+p3==3)
13.      {
14.         //some error
15.          exit(1);
16.      }
17.}
```

图 1 符号执行示例代码

首先,将函数 test 的参数 $p1$、$p2$、$p3$ 定义为符号变量 X、Y、Z,初始状态为 true。当符号执行时,从第一个 if 条件语句开始收集路径上的约束条件,直至所有路径约束条件搜索结束,示例代码的程序执行树如图 2 所示。

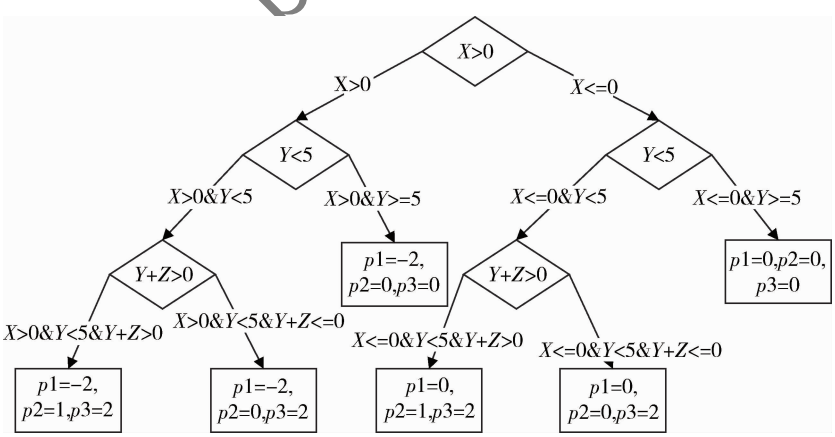

图 2 示例程序执行树

由图 2 可知符号执行分别搜索并收集全部六条路径上的所有约束条件,并利用约束求解器进行求解,求得六条路径对应的测试用例。当采用以 $<X<=0, Y<5, Y+Z>0>$ 为约束条件求得的测试用例时,其执行结果分别为 $p1=0, p2=1, p3=2$,将引发程序错误。

实际软件运行中,多数情况下利用各种函数调用实现软件内外环境交互,符号执行可以追踪到外部环境,导致更多的非原软件中的路径被执行;而外部环境的复杂性、特殊性与不可预知性,导致符

号执行无法收集特殊情况(如读写权限等)下的约束条件,降低了生成测试用例的精确性,故导致路径测试失败,间接降低了路径覆盖率。

针对上述问题,本文提出基于约束的分支覆盖符号执行优化算法,通过收集函数中的特殊参数作为约束条件,将收集到的约束条件添加到路径约束集中,减少不合格的测试用例生成,从而提升符号执行的路径覆盖率。

2 基于参数约束的符号执行优化算法

2.1 参数约束的优化算法框架

针对传统符号执行在含有特殊参数的路径中难以精确生成测试用例的问题,提出参数约束优化框架如图 3 所示。

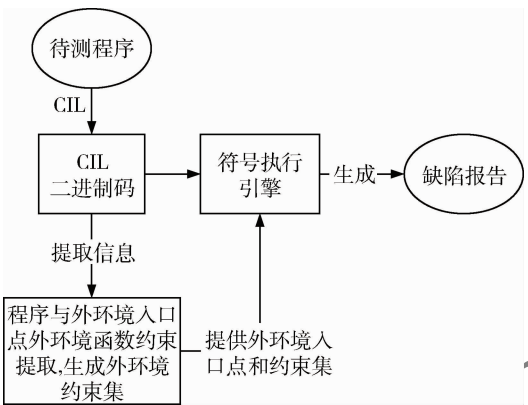

图 3 参数约束优化框架

本算法选用 CLI 编译框架将待测程序转换成二进制码并进行插桩,收集路径信息和程序执行状态。利用参数约束条件提取算法确定程序与外环境交互入口点并收集该外环境的相应约束,最后将收集到的约束条件集合合并到外环境约束集中;然后转换约束集中约束条件格式以满足测试需要。当符号执行运行到外环境交互入口点时,从外环境约束集中提取该条执行路径入口点相应约束,增加到该路径符号执行约束集中,最后由符号执行工具 CREST 生成该路径的测试用例和对应的缺陷报告。本算法使得路径约束集更加准确,生成的测试用例更加精确。

2.2 基于参数约束的符号执行优化算法

(1)参数约束条件提取

通过 CIL 插桩技术向被测程序中插入记录信息,跟踪程序的执行状态,确定外环境函数入口点和函数名称。通过函数名称去函数手册确定函数是否含有特殊参数,若没有结束;若含有特殊参数,进行下一步。对函数手册中的特殊参数进行收集,生成外环境约束集并进行格式转化,算法描述如下:

算法 1 参数约束条件提取

```
Input:  CIL Binarycode
Output:  {PC}                                  //外环境约束集
1. SET Deviationlong
2. while CIL Binarycode ≠ ∅  do
3.      if Function F have(include) Special parameters then
4.          {PC} = {PC} ∪ Special parameters
5.          Function F markFlag
6.      End if
7.      Binarycode = Binarycode - Deviationlong
8. End while
9. {PC} Format conversion
10. RETURN  {PC}
```

确定函数位置并收集特殊参数作为约束集,最后生成外环境约束集。该算法使得程序自身信息在生成测试用例时被更充分利用。

(2)约束优化算法

程序进行符号执行,当执行到函数时检查是否有标记,若有标记则为特殊函数,调用外环境约束集并提取出当前函数的约束,将其添加到当前路径中。若没有标记,则不是特殊函数,继续进行符号执行,直到符号执行结束。

算法 2 约束优化算法

```
Input:  {PC}
Output:{PC_N}
1. while Symbolic execution do
2.      if Function have Flag then
3.          O_PC = {PC ∈ Flag}
                                  //从外环境约束集中取出约束
4.          PC_ = pc ∪ O_PC
                                  //将取出的约束添加到当前路径约束
5.      End if
6. End while
7. RETURN PC_N
```

通过算法 1 做的标记,确定外环境函数的位置并提取特殊参数生成约束集,通过算法 2 将外环境约束集加入到路径约束集中。算法 1 和算法 2 使得路径约束更加精确,生成的测试用例更加有效。

3 实验与分析

3.1 实例分析

为验证算法可行性,实验前采用一段 C 语言源代码用于分析,如图 4 所示。该代码段中含有常见的部分系统函数。示例中 CREST_int(flag)表示将 flag 作为符号变量进行符号执行,当 flag 只有选择特定的参数时 open 函数才能正确返回,测试用例才能有效,else 路径才能被覆盖。

```
int main()
{
    CREST_int(flag);
    int fd = open(Ā myfileā , flag);
    if(fd < 0)
    {
       printf(Ā open err\nā );
       exit(1);
    }
    else{
          const char*msg=Ā hello open\nā ;
          write(fd,msg,strlen(msg));
          close(fd);
       }
    return 0;
}
```

图 4　示例代码

该实例中含有 TRUE 和 FALSE 两条路径,动态符号执行收集路径约束为 $<fd>=0>$ 和 $<fd<0>$。符号执行为每条路径生成对应的测试用例。当 flag ∈{RDINLY,RDWR,WRONLY,APPEND,…}时为有效输入(int 类型的宏定义)。约束条件为 $<fd<0>$ 的路径,随机生成测试用例 flag,导致 open 函数错误,则进入 TRUE 路径。约束条件为 $fd>=0$ 的路径,随机生成的正确测试用例 flag 机率极小,导致不能正确生成该条路径的测试用例,该条路径测试失败。虽然符号执行找到了该条路径,由于不能正确生成测试用例,导致不能覆盖该条路径。

向约束集添加约束的处理方法描述如下:

(1)在 CIL 插桩时收集图 4 路径约束条件,若约束条件为 open 函数返回值时,标记为外部函数入口点。

(2)通过函数手册收集 open 函数的参数 RDINLY、RDWR、WRONLY、APPEND 等信息。

(3)将 open 函数的参数信息生成外环境约束集 open_PC={flag:RDINLY | RDWR | WRONLY | APPEND | CREAT | EXCL | TRUNC | NONBLOCK},并进行格式转换。

(4)当符号执行到外部函数入口点 open 函数时将 open_PC 约束添加到 $<fd>=0>$ 和 $<fd<0>$ 约束集中,对于 $fd>=0$ 的路径,只要从 open_PC 中随机一个即可满足测试用例。对于 $<fd<0>$ 的路径,生成的测试用例只要不满足 open_PC 中的约束即可。

通过上文基于参数约束的分支覆盖符号执行优化算法可以生成更加精确的测试用例。而优化前的符号执行可以找到两条路径,可以覆盖 TRUE 路径,但是 FALSE 路径因不能正确生成测试用例而不能被覆盖。由此可见基于参数约束的优化算法比传统动态符号执行可以覆盖更多的分支路径,生成更加准确的测试用例。

3.2　实验与结果

为了验证本文提出的基于参数约束的分支覆盖符号执行优化算法,对 3 个路径规模分别为 20、60、110 的测试程序进行测试。目前含有大量特殊参数函数的测试程序较少,为了增加含有特殊参数的路径分支数,仿照实例分析的代码,构建了三个含有大量特殊参数函数(open、lseek、fcntl 等)待测程序:程序 A、程序 B、程序 C。所有实验运行在 VMware Workstation 12(RAM 2 GB)的 Ubuntu 18.04 上,宿主配置是 Intel Core i5-4210u,内存为 8 GB。

CREST 和本文算法针对构建的测试程序进行符号执行的路径覆盖数对比如图 5 所示。由图 5 可知,CREST 已经覆盖了大量的路径,但是含有特殊参数的路径,由于没有生成正确的测试用例,路径没有被覆盖。采用本文算法后 CREST 约束集更精确,生成了有效的测试用例,提高了符号执行的分支覆盖率,在一定程度上解决了因不能正确生成测试用例而不能覆盖该条路径的问题,证明了本文算法的有效性。

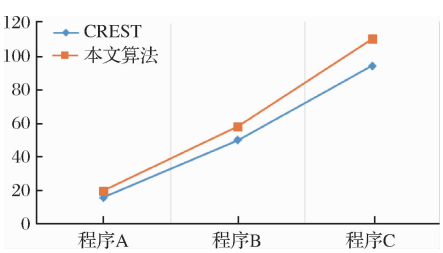

图 5　路径覆盖数对比

4　结论

符号执行路径优化的方向在于发现没有执行过的路径、程序深处的路径,但对于符号执行测试用例生成的优化较少。本文分析了由于测试用例

生成得不够精确,导致发现的路径不能被执行,降低了路径覆盖率。

本文提出了基于参数约束的分支覆盖符号执行优化算法,将具有特殊性质的参数进行收集并添加到约束集合中,使得生成的测试用例更加精确,可以执行到具有特殊参数的路径中,提高了路径覆盖率。目前只对函数的特殊参数进行了约束,下一步的研究工作是将实际程序中具有特殊性质的选项进行收集并利用,生成约束集。

参考文献

[1] 李舟军,张俊贤,廖湘科,等. 软件安全漏洞检测技术[J]. 计算机学报,2015,38(4):717-732.

[2] 谢肖飞,李晓红,陈翔,等. 基于符号执行与模糊测试的分支覆盖导向性混合测试方法[J/OL]. 软件学报:1-18[2019-11-18]. https://doi.org/10.13328/j.cnki.jos.005789.

[3] 叶志斌,严波. 符号执行研究综述[J]. 计算机科学,2018,45(S1):41-48.

[4] 汪孙律,林渝淇,杨秋松,等. 基于输入约束的符号执行优化[J]. 通信学报,2019,40(3):19-27.

[5] TRABISH D, MATTAVELLI A, RINETZKY N, et al. Chopped symbolic execution[C]. The 40th International Conference on Software Engineering. ACM,2018: 350-360.

[6] GODEFROID P. Compositional dynamic test generation[C]. Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages,2007,42:47-54.

[7] RAMOS D A, ENGLER D R. Under-constrained symbolic execution:correctness checking for real code[C]. The 24th Usenix Conference on Security Symposium. USENIX Association,2015: 49-64.

[8] WONG E,ZHANG L,WANG S,et al. DASE:document-assisted symbolic execution for improving automated software testing[C]. 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering (ICSE). ACM,2015: 620-631.

[9] 郭曦,王盼. 基于依赖条件重构的程序符号值分析方法[J]. 电子学报,2019,47(3):630-635.

[10] 安靖,钟金鑫,魏更宇,等. 函数摘要在 Concolic 测试方法中的应用[J]. 北京邮电大学学报,2012,35(1):24-27.

(收稿日期:2019-11-21)

作者简介:

於家伟(1994 -),男,硕士研究生,主要研究方向:网络与信息安全、漏洞挖掘。

李世明(1976 -),通信作者,男,硕士,副教授,主要研究方向:网络空间安全物联网技术、数据挖掘技术。E-mail:hsdlsm@163.com。

毕雪洁(1994 -),女,硕士研究生,主要研究方向:网络与信息安全、漏洞挖掘。

(上接第 13 页)

[13] Niu Xing, Sun Xinruo, Wang Haofen, et al. Zhishi.meweaving Chinese linking open data[C]. The Semantic Web-ISWC,2011:205-220.

[14] 齐斌,王宇,邹红霞,等. 基于信息熵和模糊集的网络安全知识图谱选择技术[J]. 小型微型计算机系统,2018,39(11):48-52.

[15] CHAWUTHAI R,TAKEDA H. RDF Graph Visualization by Interpreting Linked Data as Knowledge[M]. Semantic Technology,2015.

[16] HOKSZA D,JELINEK J. Using Neo4j for mining protein graphs:a case study[C]. International Workshop on Database & Expert Systems Applications,2015.

[17] WEBBER J. A programmatic introduction to Neo4j[C]. Conference on Systems,2012.

[18] DRAKOPOULOS G,BAROUTIADI A,MEGALOOIKONOMOU V. Higher order graph centrality measures for Neo4j[C]. International Conference on Information,2016.

(收稿日期:2019-10-21)

作者简介:

陶耀东(1981 -),男,博士,研究员,主要研究方向:工业互联网安全、数控技术。

贾新桐(1994 -),男,硕士,主要研究方向:工业互联网安全。

吴云坤(1975 -),男,硕士,主要研究方向:大数据安全。

版权声明