软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2019,30(11):3259−3280 [doi: 10.13328/j.cnki.jos.005562]

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

一种基于程序功能标签切片的制导符号执行分析方法*

甘水滔[1], 王林章[2], 谢向辉[1], 秦晓军[1], 周 林[1], 陈左宁[1]

[1](数学工程与先进计算国家重点实验室,江苏 无锡 214083)

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

通讯作者: 甘水滔, E-mail: ganshuitao@gmail.com

摘 要: 提出了一种基于程序功能标签切片的制导符号执行分析方法 OPT-SSE.该方法从程序功能文档提取功能标签,利用程序控制流分析,建立各功能标签和程序基本块的映射关系,并根据功能标签在程序执行中的顺序关系生成功能标签执行流.针对给定的代码目标点,提取与之相关的功能执行流切片,根据预定义好的功能标签流制导规则进行符号执行分析,在路径分析过程中,及时裁剪无关的功能分支路径以提升制导效率.通过对不同的功能标签流进行分离制导符号执行分析,可避免一直执行某复杂循环体的情形,从而提高对目标程序的整体分支覆盖率和指令覆盖率.实验结果表明,通过对 binutils、gzip、coreutils 等 10 个不同软件中的 20 个应用工具上的分析,OPT-SSE 与 KLEE 提供的主流搜索策略相比,代码目标制导速度平均提升到 4.238 倍,代码目标制导成功率平均提升了 31%,程序指令覆盖率平均提升了 8.95%,程序分支覆盖率平均提升了 8.28%.

关键词: 制导符号执行;分支覆盖率;指令覆盖率;搜索策略;程序切片

中图法分类号: TP311

中文引用格式: 甘水滔,王林章,谢向辉,秦晓军,周林,陈左宁.一种基于程序功能标签切片的制导符号执行分析方法.软件学报,2019,30(11):3259−3280. http://www.jos.org.cn/1000-9825/5562.htm

英文引用格式: Gan ST, Wang LZ, Xie XH, Qin XJ, Zhou L, Chen ZN. Guiding symbolic execution analysis by leveraging program function label slice. Ruan Jian Xue Bao/Journal of Software, 2019,30(11):3259−3280 (in Chinese). http://www.jos.org.cn/1000-9825/5562.htm

Guiding Symbolic Execution Analysis by Leveraging Program Function Label Slice

GAN Shui-Tao[1], WANG Lin-Zhang[2], XIE Xiang-Hui[1], QIN Xiao-Jun[1], ZHOU Lin[1], CHEN Zuo-Ning[1]

[1](State Key Laboratory of Mathematical Engineering and Advanced Computing, Wuxi 214083, China)

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

Abstract: This paper introduces a novel approach called OPT-SSE to speed up and scale symbolic execution—guided symbolic execution method based on program function label slice. The OPT-SSE is constituted by two parts. One part is static analysis, which decomposes the whole program execution paths by different program function label slices through establishing the mapping between function tags and program CFG. The other part is specified function tags guided rule symbolic execution, which cuts unrelated branch path timely according to specified function tags flow when taking symbolic execution analysis. Through analyzing specified function label slice or different function label slices concurrently, OPT-SSE could avoid stuck in complex cycle, which is beneficial for speeding up goal-guided process and scaling the symbolic execution. OPT-SSE is evaluated on twenty applications from ten famous open softwares, like binutils, gzip, coreutils, and so on. Through comparison with KLEE, OPT-SSE speeded up goal-guided by 4.238 times, increased the goal-guided success rate about 31%, and increased instruction coverage about 8.95%, branch coverage about 8.28%.

Key words: guided symbolic execution; branch coverage; instruction coverage; search strategy; program slice

* 基金项目: 国家自然科学基金(91318301, 61170066, 6147179)
Foundation item: National Natural Science Foundation of China (91318301, 61170066, 6147179)
收稿时间: 2016-01-14; 修改时间: 2016-02-19, 2016-03-21; 采用时间: 2016-04-14; jos 在线出版时间: 2018-04-27
CNKI 网络优先出版: 2018-05-02 14:33:56, http://kns.cnki.net/kcms/detail/11.2560.TP.20180502.1433.001.html

大多数应用软件中存在多个功能选项.在 Linux 系统下,目录“/bin”和“/usr/bin”下的绝大多数应用程序包含多个功能选项,表 1 中列举了从 Linux 系统下随机挑选的应用程序,并且统计了各自的功能选项数量,如 coreutils8.22 中的 ls 程序包含 54 个选项,who 程序中包含 14 个选项,mkdir 程序中包含 6 个选项,diffutils3.2 中 diff 程序包含 46 个选项等.可以发现,每个功能选项或几个功能选项组合,对应程序中一部分独立的执行路径.图 1 给出了一个典型的示例程序.该程序从命令行可输入 *a*,*b*,*c*,*d*,*e*,*f* 等选项,每个选项的输入将执行其对应的功能函数,如 *a* 选项对应 *exe_a* 函数等,选项 *e* 和选项 *f* 必须在选项 *c* 使用之后有效.本文把程序的命令行选项开关称为功能标签.

Table 1 Parts of applications containing function label random relected in Linux system

表 1 Linux 系统下的随机挑选的部分包含功能标签的应用程序

程序名称	功能标签个数	程序名称	功能标签个数
Bash2.1	21	Diff (diffutils3.2)	46
Wget1.16.1	132	Firefox37.0	18
Ls (coreutils8.22)	54	Ssh (openssh6.7)	43
Who (coreutils8.22)	14	Objdump (Binutils2.25)	45
Hwclock (util-Linux)	23	Xwd1.06	13
Ypdomainname (net-tools)	10	Vipw (Shadow-utils4.1.5)	6
Arp (net-tools1.6)	8	Foomatic-rip (cups-filters1.0.67)	5
Zenity	83	Blkid (util-Linux)	18
Sprof3.74	6	…	…

```
void main(int argc,char**argv){
   …
   i=decode_switches(argc,argv);
   exe_options(·);
   …
}

void exe_options(·){
   …
   if (a_mode){
      exe_a(·);   //执行选项a相关的函数
   }
   if (b_mode){
      exe_b(·);
   }
   if (c_mode){
      exe_c(·);
      if (d_mode){
         exe_d(·);   //bug point;
      } else if (f_mode){
         exe_f(·);
      }
   }
   …
}
```

```
static int decode_swithes(int argc,char**argv){
   …
   static bool a_mode, b_mode, c_mode, d_mode, e_mode;
   a_mode=b_mode=c_mode=d_mode=e_mode=false;
   while (true){
      int oi=–1;
      int c=getopt_long(argc,argv,“abcdefghi”,long_options, & oi)
      if (c==–1)
         break;
      switch(c){
         case ‘a’:
      a_mode=true; break;
         case ‘b’:
      b_mode=true; break;
         case ‘c’:
      c_mode=true; break;
         case ‘d’:
      d_mode=true; break;
         case ‘e’:
      e_mode=true; break;
         case ‘f’:
      f_mode=true; break;
      …
   }
   …
}
```

Fig.1 Sample program

图 1 示例程序

针对这种带功能标签的应用软件,其测试手段主要依赖于静态分析和动态分析两大类方法.静态分析可以通过控制流、数据流分析等方法[1]获取功能标签、程序路径及程序缺陷的相关信息,但由于不能构造测试用例,无法验证其分析结果的真实性.动态分析方法主要分为基于输入数据变异的动态分析方法[2]和符号执行分析方法[3]:基于输入数据变异的动态分析方法在不知道功能标签信息的情况下,难以自动生成功能标签信息,因而在测试的过程中极其容易陷入只对部分或极少数的功能路径进行分析,具有很强的随机性;符号执行方法把程序的外部输入数据视为符号变量,运行并分析程序路径上的所有语句,对数据依赖于外部输入的变量进行符号化,不断收集条件分支语句的条件约束建立相应的约束系统,再利用约束求解器判定并求解执行路径对应的约

束集,能够自动生成可行路径的测试例.通过精确地模拟程序的执行、跟踪变量的所有取值,理论上,符号执行可以做到对任意可执行路径的覆盖,实现对所有功能标签相关路径的测试例求解.因此,与其他测试方法相比,在功能覆盖及全局路径的覆盖效果上,符号执行方法更能适应于该类带功能标签的应用软件.但程序路径爆炸和求解开销过大,一直是限制符号执行分析效率的主要难题,尤其是在不知道程序执行路径片段和各个功能标签关系的前提下,利用符号执行进行程序分析会面临以下 3 个问题.

- 问题 1:对于给定的可能存在缺陷的执行路径片段,难以确定较优的路径搜索策略进行动态目标制导.

如图 1 和后文图 3,假设需要制导的路径位置处于功能标签 *c* 和 *e* 的相关路径中,当选择采用深度优先搜索策略(DFS)时,必须遍历所有和功能标签 *a* 和 *b* 相关的路径;当采用广度优先搜索策略(BFS)进行路径制导时,如果目标点较深,也将遍历与功能标签 *c* 和 *e* 之外的功能标签相关路径;当选择采用随机搜索策略时,将会以极小的概率制导该路径片段.

- 问题 2:难以对某功能标签的所有相关路径进行正确性验证.

如图 1 和后文图 3,假设只需要验证功能标签 *e* 相关的执行路径,和问题 1 类似,在不知道功能标签信息前提下,不论采用何种路径搜索策略,无法只选择某功能标签相关路径.如果采取种子模式,即构造一个带选项 *e* 的测试例进行混合执行,由于不知道功能标签之间的关系,难以构造测试例成功到达功能标签 *e* 相关的执行路径中.

- 问题 3:容易卡在某个包含多层循环的功能标签代码片段上.

如果某块功能代码中包含多层循环结构,一旦符号执行分析进入该代码中,容易导致过度的时间消耗,难以执行到其他的功能代码上,这样不利于对程序的全局覆盖测试.

针对以上 3 个问题,本文提出一种基于程序功能标签切片的制导符号执行分析方法(OPT-SSE),根据程序的功能文档建立相应的功能标签集合,利用静态分析方法对程序进行不同功能划分,在程序依赖图上生成不同功能标签的切片,把执行路径有序的映射到不同功能上,对于给定需要制导的目标点,提取与目标点相关的切片,通过对切片相关节点进行标记,为符号执行的路径选择提供制导信息.利用预定义的功能标签流制导规则裁剪无关路径,不仅可以加速目标点制导过程以及提升特定功能模块的覆盖率,通过功能切片的制导信息分离,还提升了对整个程序的覆盖率.

本文第 1 节描述 OPT-SSE 的基本实现框架和流程,简要分析该模型如何对图 1 给出的示例程序进行制导加速及全局覆盖率提升,并给出模型后面算法和规则描述的基本定义.第 2 节描述功能标签流排序算法.第 3 节描述基于符号执行的功能标签流制导规则.第 4 节描述测试与分析.第 5 节对符号执行相关工作进行总结.第 6 节对本文主要工作进行总结.

1 基于程序功能标签切片的制导符号执行分析模型

1.1 模型概述

图 2 给出了功能切片符号执行模型 OPT-SSE(OPT-sliced symbolic execution)的主要执行流程,具体如下.

(1) 获取功能标签.分析软件的功能文档,获取功能标签集合,如命令行功能选项集合.

(2) 静态控制流分析与制导规则生成.首先,生成软件代码的中间代码,生成每个基本块的相关信息,再利用控制流分析生成控制流图 CFG;其次,遍历该程序控制流图,确定每个和功能标签相关的基本块,生成功能标签控制流图 OPT-CFG;最后,在功能标签控制流图基础上,遍历生成功能标签路径 OPT-Path,并利用排序规则进行排序.

(3) 制导符号执行.

- 将特定功能或特定代码目标点映射到对应的一条或多条功能标签路径.
- 在符号执行分析过程中,引入针对功能标签路径的制导规则.制导规则包括两种:一种为不带功能标签路径排序信息的制导规则,另一种是带功能标签路径排序信息的制导规则.
- 针对特定功能对应的功能标签路径,生成相应的测试例集合,并进行缺陷分析.
- 针对特定代码目标点进行制导分析,需要生成相应的可达测试例集合.

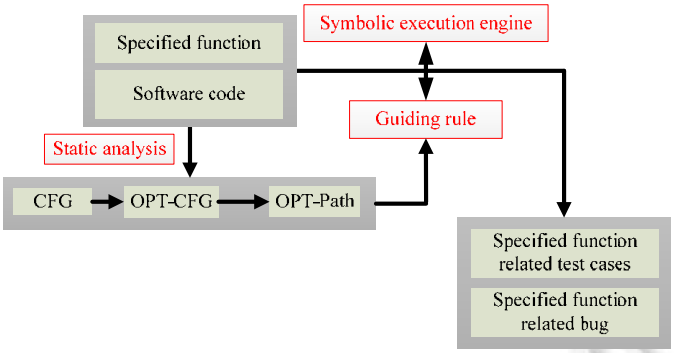


Fig.2 Guided symbolic execution model based on program function label slice (OPT-SSE)

图 2 基于程序功能标签切片的制导符号执行模型(OPT-SSE)

以图 1 的程序为示例,图 3 和图 4 展示了 OPT-SSE 模型的优化思想.

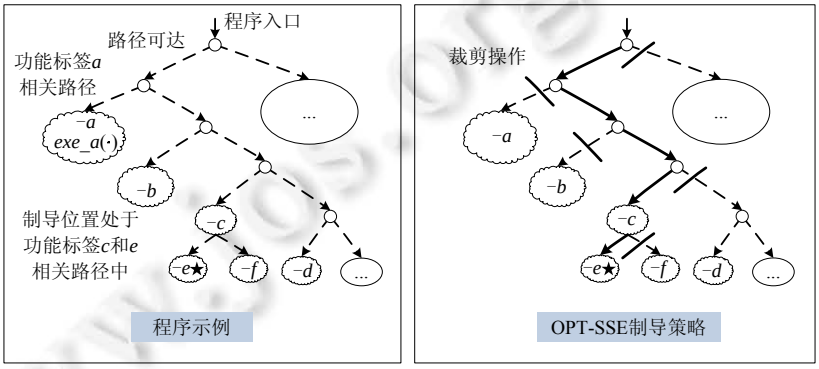


Fig.3 OPT-SSE—Speeding up goal-guided process

图 3 OPT-SSE—加速目标点制导

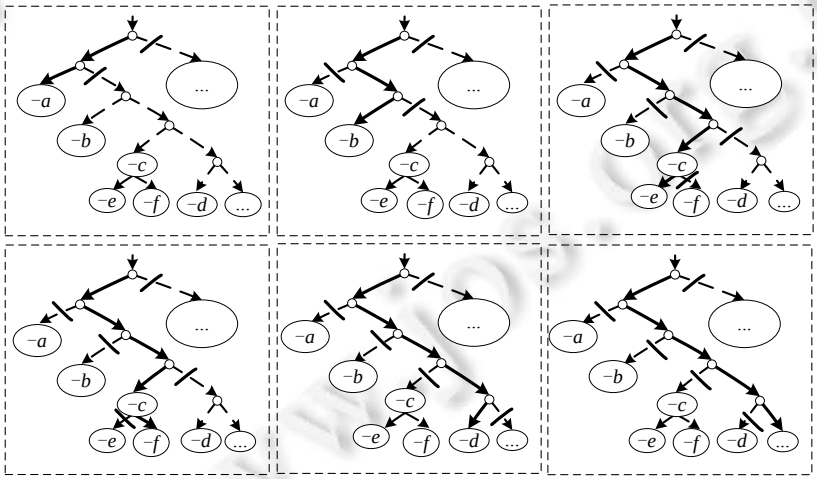


Fig.4 OPT-SSE—Improving whole coverage

图 4 OPT-SSE—提升全局覆盖率

图 3 描述了 OPT-SSE 模型加速目标点制导过程:通过静态分析目标点和各个功能标签的隶属关系,为符号执行过程提供特定功能代码分析的制导信息,在符号执行框架下,定义针对不同指令的制导语义规则,尽早地裁

剪无关的路径分支,以加速动态符号执行的目标制导过程.其中,实线箭头指向的路径是需要制导的目标路径,斜杠表示的是分支裁剪操作.图 4 描述了 OPT-SSE 模型提升全局覆盖率过程:通过对不同的功能代码进行有效分割,可对整个程序进行并行化符号执行;同时,采用多个客户端对不同的功能代码块进行分析,可避免卡在对某个功能代码块的分析上,以提升全局覆盖率.即同时采用多个符号,执行客户端对不同的实线箭头指向的路径进行制导分析.

1.2 模型基本定义

定义 1(程序功能标签). 一个测试程序的功能文档中可获取有限个功能标签,构成一张功能标签符号表 $OPT=\{option_0,option_1,\ldots,option_{num}\}$,$num$ 为功能标签的个数,$option_i$ 表示为程序的第 i 个功能标签($0<i<num$).

定义 2(功能类型分支判定条件). 对于功能标签符号表 $OPT=\{option_0,option_1,\ldots,option_{num}\}$的程序,其功能类型分支判定条件集用 $OPTcondition$ 描述.用树结构描述程序中任一分支判定条件 $condition$ 对应的表达式,操作符 $operators$ 为中间节点,操作数 $operands$ 为叶子节点,判定分支判定条件 $condition$ 是否为功能类型分支判定条件,根据以下等价关系:

$$\exists operand(operand\in operands\wedge operand\in OPT)\equiv condition\in OPTcondition.$$

定义 3(程序控制流图). 程序控制流图 CFG 用四元组$\langle N,E,entry,exit\rangle$表示,其中,$N$ 表示该过程的节点集合,E 是边的集合,$entry$ 表示入口节点,$exit$ 表示退出节点.任一节点表示为一组连续顺序执行语句的基本块,其中,N 和 $entry$ 节点包含的最后一条语句为分支语句.每条边是一个表示从节点 n_i 到节点 n_j 可能存在控制流转移的有序节点对$\langle n_i,n_j\rangle$,并满足:n_i 是 n_j 的直接前驱,n_j 是 n_i 的直接后继.每个节点可以有多个直接前驱节点,但至多有两个直接后继节点.$entry$ 是一个没有前驱的节点,$exit$ 出是一个没有后继的节点.

定义 4(程序执行路径). 对于一个程序控制流图 CFG,任意存在控制流转移的有序节点对$\langle n_i,n_{i+1}\rangle$,从节点 n_i 执行到节点 n_{i+1} 必须满足分支条件 $condition_i$,定义为 $n_i \xrightarrow{condition_i} n_{i+1}$,那么对于程序执行路径集合 $Path$,其任一条路径用有限个有序节点$\langle n_1,\ldots,n_j\rangle$表示,定义为 $n_1 \xrightarrow{condition_1} n_2,\ldots,n_{j-1} \xrightarrow{conditon_{j-1}} n_j$,并满足:

$$\langle n_1,\ldots,n_j\rangle\in Path\equiv \underset{1<i<j}{\wedge} condition_i\wedge n_1=\{entry\}\wedge n_j=\{exit\}.$$

定义 5(程序执行路径片段). 对于一个程序控制流图 CFG,其生成的路径片段集合为 $PathSeg$,一条执行路径片段用有限个有序节点$\langle n_1,\ldots,n_j\rangle$表示,定义为

$$\langle n_1,\ldots,n_j\rangle\in PathSeg\equiv\exists\langle n'_1,\ldots,n'_k\rangle\left(\langle n'_1,\ldots,n'_k\rangle\in Path\wedge\exists i\left(\underset{i<x<i+j}{\wedge} n'_x=n_x\right)\right).$$

定义 6(功能标签控制流图). 一个功能标签控制流图 $OPT\text{-}CFG$ 可由程序控制流图 CFG 遍历生成,用四元组$\langle N',E',entry,exit\rangle$表示,其中:$N'$表示该过程的节点集合,每个过程节点表示某个功能标签激活后直接到达的基本块;E'是边的集合;$entry$ 表示入口节点;$exit$ 表示退出节点.每条边用有序节点对 $\langle\langle n'_i,n'_j\rangle\rangle$ 描述,表示从节点 n'_i 到节点 n'_j 之间存在并且只存在一条功能标签控制流.节点 n'_i 到节点 n'_j 之间存在一条功能标签控制流边的充要条件为:程序存在一条从 n'_i 的语句到达 n'_j 的语句的执行路径

定义 7(功能标签执行流). 对于一个功能标签控制流图 $OPT\text{-}CFG$,功能标签执行路径集合用 $OPT\text{-}Path$ 表示,对于任意控制流 $\langle\langle n'_i,n'_{i+1}\rangle\rangle$,从节点 n'_i 执行到节点 n'_{i+1} 必须满足约束集合 $constraint_i$ 定义为 $n'_i \xrightarrow{constraint_i} n'_{i+1}$,其任一条路径用有限个有序节点 $\langle\langle n'_1,\ldots,n'_j\rangle\rangle$ 表示,并满足:

$$\langle\langle n'_1,\ldots,n'_j\rangle\rangle\in OPT\text{-}Path\equiv \underset{1<i<j}{\wedge} constraint_i\wedge n'_1=\{entry\}\wedge n'_j=\{exit\}.$$

性质 1(功能标签执行流约束集). 对于节点 n'_i 执行到节点 n'_{i+1} 的约束集合 $constraint_i$,一定满足:

$$\forall\rho(\rho\in constraint_i\wedge(\rho=\rho_1\wedge\ldots\wedge\rho_k))\rightarrow \underset{1\leqslant j<k}{\wedge}\rho_j\notin OPTcondition\wedge\rho_k\in OPTcondition,$$

$$\langle\langle n'_i,n'_{i+1}\rangle\rangle\rightarrow\exists\langle n_1,\ldots,n_j\rangle(\langle n_1,\ldots,n_j\rangle\in PathSeg\wedge n_1=n'_i\wedge n_j=n'_{i+1}).$$

性质 2(用 *OPT-Path* 对 *Path* 进行分类). 存在一个函数 Ψ,实现 $Path$ 到 $OPT\text{-}Path$ 的映射关系:

$$\Psi(X)=Y\ (X\in Path,\ Y\in OPT\text{-}Path).$$

证明:由定义可知:对于给定的程序,其程序控制流图 *CFG* 生成执行路径集合 *Path*;功能标签控制流图 *OPT-CFG* 生成功能标签执行路径集合 *OPT-Path*.对于任意一条可执行路径 $X=\langle n_1,\ldots,n_j\rangle\in Path$,通过删除其中的与功能类型分支判定条件语句无关的过程节点,得到节点有序序列 $\langle\langle n'_1,\ldots,n'_k\rangle\rangle$,一定满足: $\forall i(1<i<k-1\wedge\langle n'_i,n'_{i+1}\rangle\in E')$和 $n'_1=\{entry\}\wedge n'_k=\{exit\}$,由根据定义 6 可知, $Y=\langle\langle n'_1,\ldots,n'_k\rangle\rangle\in OPT\text{-}Path$.因此,任意一条可执行路径 X 可通过函数 Ψ映射到唯一的一条功能标签执行流 Y. □

定义 8(偏序关系算子). 偏序算子⊙={≺,≻}用来描述分支节点的后续节点的大小关系或者同一程序下任一两条执行路径的大小关系,对于节点 n(分支条件为 *condition*)的 true 后续节点 n_t和 false 后续节点 n_f,则定义为 $n\xrightarrow{condition}n_t\succ n\xrightarrow{\neg condition}n_f$ 或者$\langle n,n_t\rangle\succ\langle n,n_f\rangle$.

定义 9(程序控制流执行路径的偏序关系). 程序控制流中两条不同执行路径$\langle n_{11},\ldots,n_{1i}\rangle$和$\langle n_{21},\ldots,n_{2j}\rangle$,偏序关系的判定定义如下:

$$\exists k(1\leqslant k\leqslant\min(i,j)\wedge\mathop{\wedge}_{1\leqslant m\leqslant k}(n_{1m}=n_{2m})\wedge(\langle n_{1k},n_{1k+1}\rangle\odot\langle n_{2k},n_{2k+1}\rangle))\equiv\langle n_{11},\ldots,n_{1i}\rangle\odot\langle n_{21},\ldots,n_{2j}\rangle.$$

定义 9 表明,对于一个程序中任意两条不同路径,一定存在某个分叉节点,使得一条路径执行 true 分支,另一条路径执行 false 分支.那么执行 true 分支的路径大于执行 false 分支的路径,这样可对一个程序中所有不同的可执行路径进行大小关系排序.

定义 10(功能标签执行流的偏序关系). 如果程序中任意映射到两条功能标签执行流$\langle\langle n_{11},\ldots,n_{1i}\rangle\rangle$,$\langle\langle n_{21},\ldots,n_{2j}\rangle\rangle$的执行路径存在相同的偏序关系,那么这两条功能标签执行流也存在类似的偏序关系,即满足:

$$\forall\langle n_1,\ldots,n_k\rangle,\langle n'_1,\ldots,n'_m\rangle(\Psi(\langle n_1,\ldots,n_k\rangle)=\langle\langle n_{11},\ldots,n_{1i}\rangle\rangle\wedge\Psi(\langle n'_1,\ldots,n'_m\rangle)=\langle\langle n_{21},\ldots,n_{2j}\rangle\rangle\wedge$$
$$\langle n_1,\ldots,n_k\rangle\odot\langle n'_1,\ldots,n'_m\rangle)\rightarrow\langle\langle n_{11},\ldots,n_{1i}\rangle\rangle\odot\langle\langle n_{21},\ldots,n_{2j}\rangle\rangle.$$

性质 3. 同一程序下, 任意不同功能标签执行流存在偏序关系:

$$\forall\langle\langle n_{11},\ldots,n_{1i}\rangle\rangle,\langle\langle n_{21},\ldots,n_{2j}\rangle\rangle(\langle\langle n_{11},\ldots,n_{1i}\rangle\rangle\in OPT\text{-}Path\wedge\langle\langle n_{21},\ldots,n_{2j}\rangle\rangle\in OPT\text{-}Path)\rightarrow\langle\langle n_{11},\ldots,n_{1i}\rangle\rangle\odot\langle\langle n_{21},\ldots,n_{2j}\rangle\rangle.$$

证明:设存在$\langle n_1,\ldots,n_k\rangle$,$\langle n_{x1},\ldots,n_{xp}\rangle$, $\langle n'_1,\ldots,n'_m\rangle$,$\langle n'_{x1},\ldots,n'_{xh}\rangle\Psi(\langle n_1,\ldots,n_k\rangle)=\langle\langle n_{11},\ldots,n_{1i}\rangle\rangle$,$\Psi(\langle n_{x1},\ldots,n_{xp}\rangle)=\Psi(\langle n'_1,\ldots,n'_m\rangle)=\langle\langle n_{21},\ldots,n_{2j}\rangle\rangle$,$\Psi(\langle n'_{x1},\ldots,n'_{xh}\rangle)=\langle\langle n_{21},\ldots,n_{2j}\rangle\rangle$, 满足:当 $\langle n_1,\ldots,n_k\rangle\succ\langle n'_1,\ldots,n'_m\rangle$ 时, $\langle n_{x1},\ldots,n_{xp}\rangle\prec\langle n'_{x1},\ldots,n'_{xh}\rangle$. 根据假设,一定存在 $n_{2y}(1<y\leqslant j)$或者 $n_{1d}(1<d\leqslant i)$,n_{2y}和 n_{1d}其中至少有一个节点具有两个相同的功能标签控制流前驱节点,这样与定义 6 相矛盾,假设不成立.所以,任意不同功能标签执行流之间存在偏序关系. □

定义 11(带制导信息的符号执行状态). 符号执行环境下生成的程序状态集合为$\mathbb{S}$,任一状态 $S\in\mathbb{S}$用五元组$\langle s,\rho,\sigma,g,f\rangle$表示,其中,$s$ 表示状态 S 下一步执行的语句序列,ρ表示状态 S 的约束条件集合,σ表示状态 S 的符号变量到符号值的映射关系,g 表示状态 S 下一步需要制导的基本块信息,f 表示状态 S 需要制导的功能标签流.

2 功能标签流排序算法

功能标签路径生成和排序算法 *OptPathGenerationAndSorting* 第 1 行~第 8 行通过对每个过程内控制流图进行线性扫描,获取包含功能标签分支语句的基本块,再通过控制流分析确定满足功能标签分支条件的后继基本块,加入到功能标签节点集合 *OptNode* 中;第 9 行通过调用 *OptCfgGeneration* 函数生成过程控制流图对应的功能标签控制流图 *OptCfg*;第 10 行、第 11 行对第 1 条功能标签路径的节点数量初始化;第 12 行通过调用 *OptPathSorting* 生成排序好的功能标签路径集合.

功能标签路径生成和排序算法. *OptPathGenerationAndSorting*.

输入:*FG*:程序控制流图.

输出:*SorteOptPath*:排序好的功能标签流.

```
1:  OptNode←{EntryNode,ExitNode}
2:  foreach Node in CFG
3:    if Node is from opt true branch then
4:      OptNode←OptNode∪{Node}
```

```
5:        OptNode.Map[Node]←{Node}
6:        Node.succOptNum←0
7:      endif
8:    endfor
9:    OptCfg←OptCfgGeneration(CFG,EntryNode,EntryNode)
10: PathOrder←0
11: SortedOptPath[PathOrder].NodeNum←0
12: return OptPathSorting(OptCfg,EntryNode)
```

功能标签控制流图生成算法 *OptCfgGeneration* 利用两遍遍历从程序控制流图生成功能标签控制流图,算法第 1 行~第 13 行第 1 遍深度遍历生成功能标签控制流图的中间图,该图会出现某个功能标签节点可能出现多个相同的前驱节点;算法第 14 行~第 23 行通过对每个功能标签节点进行检查,对于出现多个相同的前驱节点的功能标签节点,复制一个别名功能标签节点,并建立和功能标签节点的映射关系,这样就能确立功能标签路径的唯一性.图 5 给出了一个样例.

功能标签控制流图生成算法. *OptCfgGeneration*.

输入:*CFG*:程序控制流图;

CurrentNode:当前节点;

CurrentOptNode:当前功能标签节点.

输出:*OptCfg*:功能标签控制流图.

```
1:   foreach currentNode.succNode[i] in currentNode
2:     if currentNode.succNode[i]∈OptNode then
3:       if cuttentOptNode.succOptNum=0 or (cuttentOptNode.succOptNum>0 and currentNode.succNode[i]≠
           CuttentOptNode.succOptNode[CuttentOptNode.succOptNum−1]) then
4:       CuttentOptNode.succOptNode[CuttentOptNode.succOptNum]←currentNode.succNode[i]
5:       CuttentOptNode.succOptNum←CuttentOptNode.succOptNum+1
6:       endif
7:       if currentNode.succNode[i]≠ExitNode then
8:         OptCfgGeneration(CFG,currentNode.succNode[i],currentNode.succNode[i])
9:       endif
10:    else
11:      OptCfgGeneration(CFG,currentNode.succNode[i],CuttentOptNode)
12:    endif
13: endfor
14: foreach OptNode.succOptNode[i], OptNode.succOptNode[j] in OptNode
15:    if OptNode.succOptNode[i]=OptNode.succOptNode[j] then
16:      New newOptNode
17:      OptNode.Map[OptNode.succOptNode[i]]←OptNode.Map[OptNode.succOptNode[i]]∪{newOptNode}
18:      foreach OptNode.succOptNode[j].succOptNode[k] in OptNode.succOptNode[j]
19:        newOptNode.succOptNode[k]←OptNode.succOptNode[j].succOptNode[k]
20:      endfor
21:      OptNode.succOptNode[j]←newOptNode
22:    endif
23: endfor
```

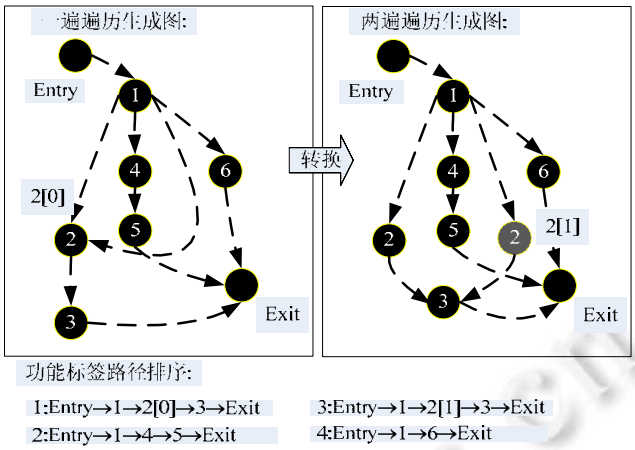


Fig.5 Uniqueness generation for function label path

图 5 功能标签路径唯一性确立

功能标签流排序算法 *OptPathSorting* 对功能标签控制流图深度遍历生成功能标签路径集合,根据路径生成的先后次序正好确定了功能标签路径的偏序关系.

功能标签流排序. ***OptPathSorting***.

输入:*OptCfg*:功能标签控制流图;

CurrentOptNode:当前功能标签节点.

输出:*SortedOptPath*:排序好的功能标签流.

1: **foreach** *currentOptNode.succOptNode*[*i*] **in** *currentOptNode*
2: **if** *currentOptNode.succOptNode*[*i*]≠*ExitNode* **then**
3: *SortedOptPath*[*PathOrder*][*NodeNum*]←*currentOptNode.succOptNode*[*i*]
4: *SortedOptPath*[*PathOrder*].*NodeNum*←*SortedOptPath*[*PathOrder*].*NodeNum*+1
5: *OptPathSorting*(*OptCfg*,*currentOptNode.succNode*[*i*])
6: **else**
7: *SortedOptPath*[*PathOrder*][*NodeNum*]←*ExitNode*
8: *SortedOptPath*[*PathOrder*].*NodeNum*←*SortedOptPath*[*PathOrder*].*NodeNum*+1
9: *PathOrder*←*PathOrder*+1
10: *SortedOptPath*[*PathOrder*].*NodeNum*←0
11: **endif**
12: **endfor**

3 基于符号执行的功能标签流制导规则

以下定义了两种功能标签流制导规则:无排序信息的功能标签流制导规则和带序信息的功能标签流制导规则.每种制导规则都实现了 3 种指令处理语义规则:调用指令处理语义规则、分支指令处理语义规则、其他指令处理语义规则.由于分支指令处理语义涉及的情况多,表 2 定义了规则中使用的部分基本符号,作为任意语义描述的前提条件,使用真值表(表 3 和表 4)对前置条件的取值进行描述,并对应不同的语义情况,表 5 为表 3、表 4 中前置条件对应的输出集合.

Table 2 Basic symbol definition of rule (as precondition for any semantic description)

表 2 规则基本符号定义(作为任意语义描述的前提条件)

DEDINITION	
s: if *condition* then bb_1 else bb_2; *bb*	ρ_1: $\rho \wedge condition$
s_{true}: bb_1; *bb*	ρ_2: $\rho \wedge \neg condition$
S_{false}: bb_2; *bb*	S_0: one statement
$f:\langle\langle n_1',...,n_k'\rangle\rangle$	$p:\langle\langle n_1',...,n_i'\rangle\rangle$
σ': rewrite σ after one step execution	

Table 3 Precondition truth Table 2 on branch instruction processing semantics of function label flow guided rule without ordering information

表 3 无排序信息的功能标签流制导规则中分支指令处理相关语义前置条件真值表 2

condition	1	2	3	4	5	6	7	8	9	10
$i=k$	false	false	–	–	–	–	–	–	–	–
$1 \leqslant i \leqslant k$	true	true	true	true	true	true	true	true	true	true
$g = n_i'$	true	true	true	true	true	true	true	true	true	true
$g=\varepsilon$	false	false	false	false	false	false	false	false	false	false
$\rho_1 \wedge \tilde{\sigma}$	true	false	true	false	true	false	true	true	true	true
$\rho_2 \wedge \tilde{\sigma}$	false	true	false	true	false	true	true	true	true	true
$bb_1 = n_i'$	true	–	false	–	false	–	false	false	false	false
$bb_2 = n_i'$	–	true	–	false	–	false	false	false	false	false
$bb_1 \in N'$	true	–	false	–	true	–	false	false	true	true
$Bb_2 \in N'$	–	true	–	false	–	true	false	true	false	true

Table 4 Precondition truth Table 3 on branch instruction processing semantics of function label flow guided rule without ordering information

表 4 无排序信息的功能标签流制导规则中分支指令处理相关语义前置条件真值表 3

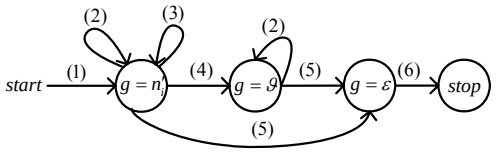
condition	11	12	13	14	15	16	17	18	19	20
$i=k$	false	false	false	false	false	true	true	true	true	–
$1 \leqslant i \leqslant k$	true	true	true	true	true	true	true	true	true	–
$g = n_i'$	true	true	true	true	true	true	true	true	true	–
$g=\varepsilon$	false	false	false	false	false	false	false	false	false	true
$\rho_1 \wedge \tilde{\sigma}$	true	true	true	true	true	true	true	true	true	–
$\rho_2 \wedge \tilde{\sigma}$	true	true	true	true	true	true	true	true	true	–
$bb_1 = n_i'$	true	true	false	true	false	true	true	false	false	–
$bb_2 = n_i'$	true	false	true	false	true	true	false	true	false	–
$bb_1 \in N'$	true	false	true	true	true	true	true	true	true	–
$Bb_2 \in N'$	true	true	true	true	true	true	true	true	true	–

Table 5 Precondition and related output assemble

表 5 前置条件和对应的输出集合

pre	$\zeta(\langle s,\rho,\sigma,g,f\rangle)$	*pre*	$\zeta(\langle s,\rho,\sigma,g,f\rangle)$
1	$\{\langle s_{\text{true}},\rho_1,\sigma,n_{i+1}',f\rangle\}$	11	$\{\langle s_{\text{true}},\rho_1,\sigma,n_{i+1}',f\rangle,\langle s_{\text{false}},\rho_2,\sigma,n_{i+1}',f\rangle\}$
2	$\{\langle s_{\text{false}},\rho_2,\sigma,n_{i+1}',f\rangle\}$	12	$\{\langle s_{\text{true}},\rho_1,\sigma,n_{i+1}',f\rangle,\langle s_{\text{false}},\rho_2,\sigma,n_i',f\rangle\}$
3	$\{\langle s_{\text{true}},\rho_1,\sigma,n_i',f\rangle\}$	13	$\{\langle s_{\text{true}},\rho_1,\sigma,n_i',f\rangle,\langle s_{\text{false}},\rho_2,\sigma,n_{i+1}',f\rangle\}$
4	$\{\langle s_{\text{false}},\rho_2,\sigma,n_i',f\rangle\}$	14	$\{\langle s_{\text{true}},\rho_1,\sigma,n_{i+1}',f\rangle,\langle s_{\text{false}},\rho_2,\sigma,\varepsilon,f\rangle\}$
5	$\{\langle s_{\text{true}},\rho_1,\sigma,g,f\rangle\}$	15	$\{\langle s_{\text{true}},\rho_1,\sigma,\varepsilon,f\rangle,\langle s_{\text{false}},\rho_2,\sigma,n_{i+1}',f\rangle\}$
6	$\{\langle s_{\text{false}},\rho_2,\sigma,g,f\rangle\}$	16	$\{\langle s_{\text{true}},\rho_1,\sigma,\vartheta,f\rangle,\langle s_{\text{false}},\rho_2,\sigma,\vartheta,f\rangle\}$
7	$\{\langle s_{\text{true}},\rho_1,\sigma,n_i',f\rangle,\langle s_{\text{false}},\rho_2,\sigma,n_i',f\rangle\}$	17	$\{\langle s_{\text{true}},\rho_1,\sigma,\vartheta,f\rangle,\langle s_{\text{false}},\rho_2,\sigma,\varepsilon,f\rangle\}$
8	$\{\langle s_{\text{true}},\rho_1,\sigma,n_i',f\rangle,\langle s_{\text{false}},\rho_2,\sigma,\varepsilon,f\rangle\}$	18	$\{\langle s_{\text{true}},\rho_1,\sigma,\varepsilon,f\rangle,\langle s_{\text{false}},\rho_2,\sigma,\vartheta,f\rangle\}$
9	$\{\langle s_{\text{true}},\rho_1,\sigma,\varepsilon,f\rangle,\langle s_{\text{false}},\rho_2,\sigma,n_i',f\rangle\}$	19	$\{\langle s_{\text{true}},\rho_1,\sigma,\varepsilon,f\rangle,\langle s_{\text{false}},\rho_2,\sigma,\varepsilon,f\rangle\}$
10	$\{\langle s_{\text{true}},\rho_1,\sigma,\varepsilon,f\rangle,\langle s_{\text{false}},\rho_2,\sigma,\varepsilon,f\rangle\}$	20	$\varnothing$

针对目标功能标签流 $f=\langle n'_1,...,n'_k\rangle$ 的符号状态 $S=\langle s,\rho,\sigma,g,f\rangle$,图 6 用有限状态机模型描述了制导信息 g 状态的转换过程,根据符号执行时下一步将要执行的基本块信息或语句信息确定制导信息的类型.当符号执行刚开始时,g 为初始化状态 n'_1(即入口基本块),当下一步执行将跳转到非功能标签节点或者执行当前基本块的下一条指令,如果 g 为符号 ε,则表明 S 不满足,无条件终止该符号状态;如果 g 为符号 ϑ,则表明 S 满足功能标签流 f.



(1):$i=1$ (2):跳转到非功能标签节点或执行当前基本块下一条指令
(3):跳转到的基本块等于$n'_i, i=i+1$
(4):最后一个相关功能标签节点可达
(5):跳转到不相关的功能标签节点
(6):无条件

Fig.6 Finite state machine model for guided symbolic state information conversion

图 6 符号状态制导信息转换的有限状态机模型

无排序信息的功能标签流制导规则:

Guiding Rule without order information: $\langle s,\rho,\sigma,g,f\rangle \twoheadrightarrow \zeta(\langle s,\rho,\sigma,g,f\rangle)$.

调用指令处理语义规则:

$$\frac{F(...)\{bb_{F\,entry};...;bb_{F\,ret}\}}{\langle F_{call};s_0,\rho,\sigma,g,f\rangle \twoheadrightarrow \{\langle bb_{F\,entry},\rho,\sigma',g,f\rangle\}},\ \frac{F(...)\{bb_{F\,entry};...;bb_{F\,ret}\}}{\langle F_{ret};s_0,\rho,\sigma,g,f\rangle \twoheadrightarrow \{\langle s_0,\rho,\sigma',g,f\rangle\}}.$$

调用指令处理语义 1 描述的是当调用一个函数时的制导符号状态的变化情况,调用指令处理语义 2 描述的是当从一个函数返回时的制导符号状态的变化情况.

分支指令处理语义规则:

$$\frac{\boxed{1}}{\langle s,\rho,\sigma,g,f\rangle \twoheadrightarrow \{\langle s_{\text{true}},\rho_1,\sigma,n'_{i+1},f\rangle\}},\ \frac{\boxed{11}}{\langle s,\rho,\sigma,g,f\rangle \twoheadrightarrow \{\langle s_{\text{true}},\rho_1,\sigma,n'_{i+1},f\rangle,\langle s_{\text{false}},\rho_2,\sigma,n'_{i+1},f\rangle\}},$$

$$\frac{\boxed{2}}{\langle s,\rho,\sigma,g,f\rangle \twoheadrightarrow \{\langle s_{\text{false}},\rho_2,\sigma,n'_{i+1},f\rangle\}},\ \frac{\boxed{12}}{\langle s,\rho,\sigma,g,f\rangle \twoheadrightarrow \{\langle s_{\text{true}},\rho_1,\sigma,n'_{i+1},f\rangle,\langle s_{\text{false}},\rho_2,\sigma,n'_i,f\rangle\}},$$

$$\frac{\boxed{3}}{\langle s,\rho,\sigma,g,f\rangle \twoheadrightarrow \{\langle s_{\text{true}},\rho_1,\sigma,n'_i,f\rangle\}},\ \frac{\boxed{13}}{\langle s,\rho,\sigma,g,f\rangle \twoheadrightarrow \{\langle s_{\text{true}},\rho_1,\sigma,n'_i,f\rangle,\langle s_{\text{false}},\rho_2,\sigma,n'_{i+1},f\rangle\}},$$

$$\frac{\boxed{4}}{\langle s,\rho,\sigma,g,f\rangle \twoheadrightarrow \{\langle s_{\text{false}},\rho_2,\sigma,n'_i,f\rangle\}},\ \frac{\boxed{14}}{\langle s,\rho,\sigma,g,f\rangle \twoheadrightarrow \{\langle s_{\text{true}},\rho_1,\sigma,n'_{i+1},f\rangle,\langle s_{\text{false}},\rho_2,\sigma,\varepsilon,f\rangle\}},$$

$$\frac{\boxed{5}}{\langle s,\rho,\sigma,g,f\rangle \twoheadrightarrow \{\langle s_{\text{true}},\rho_1,\sigma,\varepsilon,f\rangle\}},\ \frac{\boxed{15}}{\langle s,\rho,\sigma,g,f\rangle \twoheadrightarrow \{\langle s_{\text{true}},\rho_1,\sigma,\varepsilon,f\rangle,\langle s_{\text{false}},\rho_2,\sigma,n'_{i+1},f\rangle\}},$$

$$\frac{\boxed{6}}{\langle s,\rho,\sigma,g,f\rangle \twoheadrightarrow \{\langle s_{\text{false}},\rho_2,\sigma,\varepsilon,f\rangle\}},\ \frac{\boxed{16}}{\langle s,\rho,\sigma,g,f\rangle \twoheadrightarrow \{\langle s_{\text{true}},\rho_1,\sigma,\vartheta,f\rangle,\langle s_{\text{false}},\rho_2,\sigma,\vartheta,f\rangle\}},$$

$$\frac{\boxed{7}}{\langle s,\rho,\sigma,g,f\rangle \twoheadrightarrow \{\langle s_{\text{true}},\rho_1,\sigma,n'_i,f\rangle,\langle s_{\text{false}},\rho_2,\sigma,n'_i,f\rangle\}},\ \frac{\boxed{17}}{\langle s,\rho,\sigma,g,f\rangle \twoheadrightarrow \{\langle s_{\text{true}},\rho_1,\sigma,\vartheta,f\rangle,\langle s_{\text{false}},\rho_2,\sigma,\varepsilon,f\rangle\}},$$

$$\frac{\boxed{8}}{\langle s,\rho,\sigma,g,f\rangle \twoheadrightarrow \{\langle s_{\text{true}},\rho_1,\sigma,n'_i,f\rangle,\langle s_{\text{false}},\rho_2,\sigma,\varepsilon,f\rangle\}},\ \frac{\boxed{18}}{\langle s,\rho,\sigma,g,f\rangle \twoheadrightarrow \{\langle s_{\text{true}},\rho_1,\sigma,\varepsilon,f\rangle,\langle s_{\text{false}},\rho_2,\sigma,\vartheta,f\rangle\}},$$

$$\frac{\boxed{9}}{\langle s,\rho,\sigma,g,f\rangle \twoheadrightarrow \{\langle s_{\text{true}},\rho_1,\sigma,\varepsilon,f\rangle,\langle s_{\text{false}},\rho_2,\sigma,n'_i,f\rangle\}},\ \frac{\boxed{19}}{\langle s,\rho,\sigma,g,f\rangle \twoheadrightarrow \{\langle s_{\text{true}},\rho_1,\sigma,\varepsilon,f\rangle,\langle s_{\text{false}},\rho_2,\sigma,\varepsilon,f\rangle\}},$$

$$\frac{\boxed{10}}{\langle s,\rho,\sigma,g,f\rangle \twoheadrightarrow \{\langle s_{\text{true}},\rho_1,\sigma,\varepsilon,f\rangle,\langle s_{\text{false}},\rho_2,\sigma,\varepsilon,f\rangle\}},\ \frac{\boxed{20}}{\langle s,\rho,\sigma,g,f\rangle \twoheadrightarrow \varnothing}.$$

如前置条件真值表(表 3)所示前置条件 1 对应的表达式为

$$SAT(i=k)=\text{false}\wedge SAT(1\leqslant i\leqslant k)=\text{true}\wedge SAT(g=n_i')=\text{true}\wedge SAT(g=\varepsilon)=\text{false}\wedge$$
$$SAT(\rho_1\wedge\tilde{\sigma})=\text{true}\wedge SAT(\rho_2\wedge\tilde{\sigma})=\text{false}\wedge SAT(bb_1=n_i')=\text{true}\wedge SAT(bb_1\in N')=\text{true}.$$

图 7 对应了分支指令处理语义规则的各种语义:语义 1 和语义 2 表示只有一个条件分支可满足,并且直接跳转到目标功能标签流对应的相关节点;语义 3 和语义 4 表示只有一个条件分支可满足,并且直接跳转到非功能标签节点;语义 5 和语义 6 表示只有一个条件分支可满足,并且直接跳转到与目标功能标签流不相关的功能标签节点;语义 7~语义 19 表示两个条件分支均可满足,分别跳转到与目标功能标签流不相关的功能标签节点、与目标功能标签流相关的功能标签节点、非功能标签节点的各种组合情况;语义 20 表示当制导符号状态下一步需要制导的基本块信息 $g=\varepsilon$时,直接删除该制导符号状态.

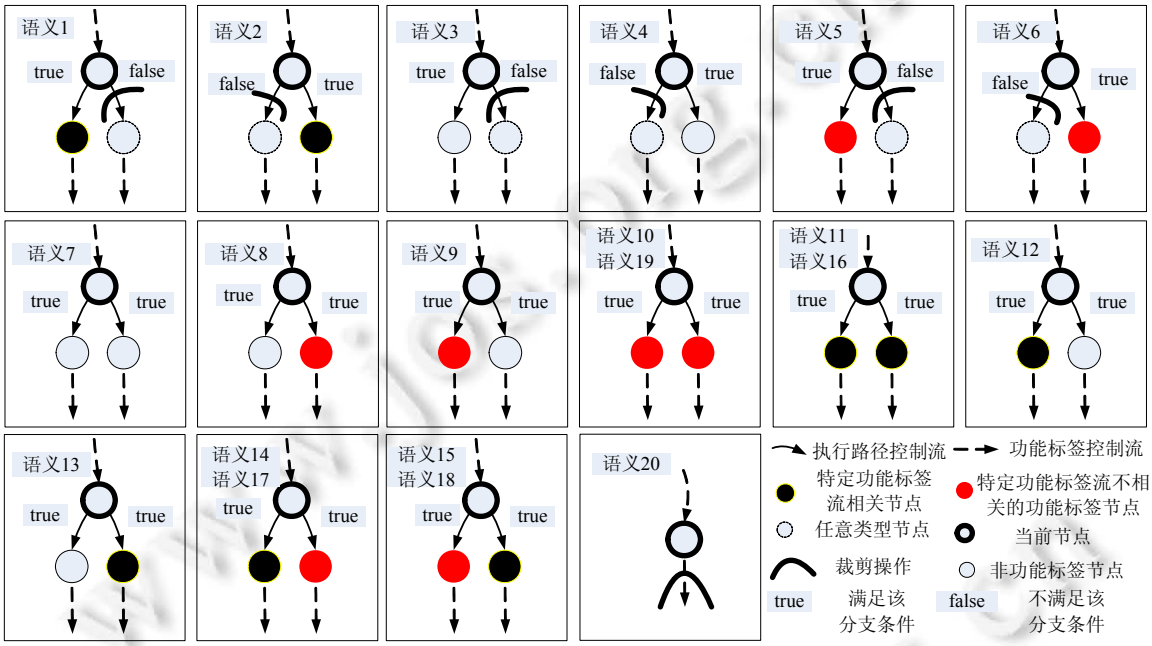


Fig.7 Branch instruction processing semantics of function label flow guided rule without ordering information

图 7 无排序信息的功能标签流制导规则中分支指令处理相关语义

其他指令处理语义规则:

$$\frac{s_0 \text{ is not } br \text{ statement}}{\langle s_0;s_1,\rho,\sigma,g,f\rangle\rightarrow\{\langle s_1,\rho,\sigma',g,f\rangle\}}.$$

调用指令处理语义规则和其他指令处理语义规则与无排序信息的功能标签流制导规则中各条语义的前置条件、输入、输出完全一样;分支指令处理语义规则与无排序信息的功能标签流制导规则下第 1 条~第 7 条和第 10 条~第 20 条共 18 条语义的前置条件、输入、输出完全一样;第 8 条和第 9 条语义更改为以下 4 条语义操作:前两条语义实现图 8 中的裁剪策略 1,后两条语义实现了图 8 中的裁剪策略 2.裁剪策略 1 表示当制导的两个分支中,其中一个基本块属于制导的特定功能标签之外的功能标签时,另一个基本块属于非特定功能标签,并满足属于制导的特定功能标签之外的功能标签的基本块在外侧.由于内侧的基本块或后续基本块仍然可能属于特定功能标签,所以只能裁剪掉包含属于制导的特定功能标签之外的功能标签的分支.裁剪策略 2 表示当制导的两个分支中的一个基本块属于制导的特定功能标签之外的功能标签时,另一个基本块属于制导的特定功能标签,并且属于制导的特定功能标签之外的功能标签基本块处于内侧.如果根据排序信息知道特定功能标签流处于这两侧功能标签流之间,那么可以裁剪掉这两个分支.

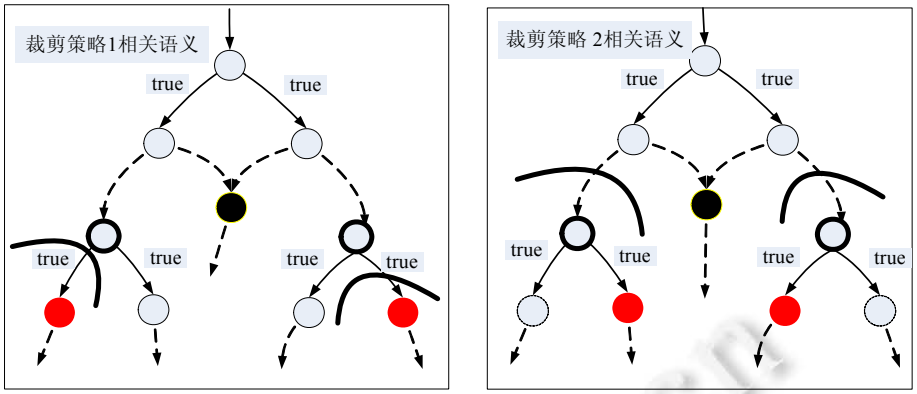


Fig.8 Differece between branch instruction processing semantics of function label flow guided rule without ordering information and that with ordering information

图 8 有排序信息的功能标签流制导规则中分支指令处理语义规则与无排序信息的功能标签流制导规则中分支指令处理语义规则区别

有排序信息的功能标签流制导规则:

Guiding Rule with order information: $\langle s,\rho,\sigma,g,f\rangle \Rightarrow \zeta(\langle s,\rho,\sigma,g,f\rangle)$.

$$\frac{\boxed{8}\ SAT(f \prec p)=\text{false}}{\langle s,\rho,\sigma,g,f\rangle \Rightarrow \{\langle s_{\text{true}},\rho_1,\sigma,n_i',f\rangle,\langle s_{\text{false}},\rho_2,\sigma,\varepsilon,f\rangle\}},\quad \frac{\boxed{9}\ SAT(f \succ p)=\text{false}}{\langle s,\rho,\sigma,g,f\rangle \Rightarrow \{\langle s_{\text{true}},\rho_1,\sigma,\varepsilon,f\rangle,\langle s_{\text{false}},\rho_2,\sigma,n_i',f\rangle\}},$$

$$\frac{\boxed{8}\ SAT(f \prec p)=true}{\langle s,\rho,\sigma,g,f\rangle \Rightarrow \{\langle s_{\text{true}},\rho_1,\sigma,\varepsilon,f\rangle,\langle s_{\text{false}},\rho_2,\sigma,\varepsilon,f\rangle\}},\quad \frac{\boxed{9}\ SAT(f \succ p)=true}{\langle s,\rho,\sigma,g,f\rangle \Rightarrow \{\langle s_{\text{true}},\rho_1,\sigma,\varepsilon,f\rangle,\langle s_{\text{false}},\rho_2,\sigma,\varepsilon,f\rangle\}}.$$

根据上述对各种指令的制导语义规则的描述,相对于 KLEE 而言,OPT-SSE 在整个符号执行过程中只会增加分支指令解释执行时的时间开销,这个时间开销主要是判定跳转到的下一个基本块是否包含特定功能标签的分支语句带来的,OPT-SSE 在静态分析环节可以标记任意基本块是否包含功能标签类分支,因此在制导过程中可直接判定某节点是否为功能标签相关节点,不会有额外的时间开销,但需要判定某功能标签节点是否为特定功能标签相关节点的时间开销上限为$|f|$倍(即功能标签流 f 包含功能标签的个数).对于整个程序而言,功能标签相关节点数量只占所有节点数量的极小比例,因此对比 KLEE,OPT-SSE 制导时间增加的时间开销可以忽略.

4 测试与分析

OPT-SSE 实验平台在以下环境实现:处理器:Intel(R) Xeon(R) CPU E7-4830 2.13GHz;内存:64GB;操作系统:Ubuntu 12.04.

OPT-SSE 包括静态分析和动态分析两个模块:原型系统在 llvm 上实现静态分析环节,包括生成控制流图、标记关键指令、生成程序功能标签流等工作,为动态分析提供辅助信息;整个动态分析环节在 klee 上实现,包括对上述各条制导规则的实现.

本文从开源库中选择了 wla_dx、Vim、Unrtf、Binutils、Sed、Wdiff、which、Findutils、Gzip、Coreutils 这 10 个不同的开源软件作为测试对象,并挑选这些软件中的 20 个不同目标进行测试,基本信息见表 6.

表 6 描述了各个软件对象的版本和功能、测试目标以及对应 llvm IR 静态指令数量等信息.实验主要针对 OPT-SSE 的指令覆盖率、分支覆盖率、代码目标制导效率等方面的性能进行测试,并且将其与 klee 进行横向比较,体现本文工作的优化能力.

表 7 给出了 wla-gb 等 20 个软件目标的详细实验数据,实验设置每次对目标的测试时间上限为 10h.在 OPT-SSE 上设置程序功能标签流的最大数量为 8;在 klee 上对每个目标分别进行宽度、深度、随机这 3 种路径搜索策略的测试,选取其中覆盖率最优的一组数据.

Table 6 Basic information of subjects

表 6 测试软件对象基本信息

软件目标	版本	测试目标名称	静态指令数量	软件功能介绍
wla_dx	9.5	wla-gb	23 242	跨平台反汇编器
Vim	7.0	vim/xxd	291557/61393	文本编辑器
Unrtf	0.21.9	unrtf	12 939	Rtf 格式转换工具
Binutils	2.25	readelf	45 315	二进制格式文件处理工具
Sed	4.2.2	seed	23 117	行在线编辑器
Wdiff	1.2.2	wdiff	7 757	文件相似性比较工具
Which	2.21	which	7 293	命令路径查询工具
Findutils	4.4.2	find/xargs/locate	40739/7838/25566	文件路径查询工具
Gzip	1.6	gzip	19 214	压缩工具
Coreutils	8.1	dd/dircolors/mkdir/pathchk/ factor/expr/split/mkfifo	9073/6935/9512/6574/ 6463/24578/10353/6214	文件系统工具集合

Table 7 Experimental data for each subject

表 7 对各个软件目标测试的实验数据

测试目标	测试方法	*MinTestCaseNum*	*MinExeInstrNum*	*MaxInstrCoverage* (%)	*MaxBrCoverage* (%)
wla-gb	KLEE	142 883	2 228 224	10.16	6.92
	OPT-SSE	159 093	6 023 283	11.24	7.77
vim	KLEE	58 323	36 506 833	3.78	2.87
	OPT-SSE	110 641	158 072 832	6.56	4.91
xxd	KLEE	103 168	18 677 760	47.76	40.45
	OPT-SSE	99 959	50 135 040	50.76	43.09
unrtf	KLEE	8 866	36 226 805	8.57	7.35
	OPT-SSE	7 633	38 992 925	30.16	20.98
readelf	KLEE	4 342	12 343 533	4.81	3.87
	OPT-SSE	7 473	28 454 252	6.81	5.99
seed	KLEE	66 350	24 444 928	19.62	15.45
	OPT-SSE	166 565	39 945 674	28.99	24.38
wdiff	KLEE	47 469	13 172 736	43.46	33.33
	OPT-SSE	32 345	8 842 079	45.98	35.78
which	KLEE	165 689	16 711 680	25.56	19.46
	OPT-SSE	154 442	13 267 773	37.67	30.54
find	KLEE	51 462	69 533 696	17.6	14.69
	OPT-SSE	67 675	124 346 643	23.22	19.8
xargs	KLEE	7 548	4 093 912	23.53	16.81
	OPT-SSE	4 567	5 292 922	33.34	28.21
locate	KLEE	170 261	70 451 200	8.92	6.85
	OPT-SSE	323 272	135 787 662	15.21	12.45
gzip	KLEE	145 957	516 550 259	20.53	17.20
	OPT-SSE	234 544	791 230 011	32.78	31.1
dd	KLEE	290 078	104 565 669	27.30	21.71
	OPT-SSE	423 444	411 895 643	34.28	30.09
dircolors	KLEE	567 832	68 688 328	27.44	21.92
	OPT-SSE	344 318	36 765 156	39.21	38.23
mkdir	KLEE	4 843	9 986 269	43.33	36.82
	OPT-SSE	56 333	23 454 330	58.9	55.2
pathchk	KLEE	68 754	18 136 042	41.61	33.98
	OPT-SSE	323 321	20 002 123	64.32	43.78
factor	KLEE	5 768	33 333 865	36.68	29.43
	OPT-SSE	8 967	43 345 441	39.22	32.78
expr	KLEE	454	3 421 222	7.1	5.64
	OPT-SSE	3 343	2 334 0989	21.2	18.89
split	KLEE	25 260	1 899 247	28.11	22.07
	OPT-SSE	127 650	7 356 452	38.38	33.26
mkfifo	KLEE	2 853	2 210 963	41.47	33.92
	OPT-SSE	6 750	17 864 332	48.22	39.1

表 7 分别统计了各个目标在 KLEE 和 OPT-SSE 上的 *MinTestCaseNum*、*MinExeInstrNum*、*MaxInstrCoverage*、*MaxBrCoverage* 等指标数据,其中,

- *MaxInstrCoverage* 表示测试 10h 中取得的最大指令覆盖率.
- *MaxBrCoverage* 表示测试 10h 中取得的最大分支覆盖率.
- *MinTestCaseNum* 表示测试的 10h 中,达到最大指令覆盖率和最大分支覆盖率的最小测试例生成数量.
- *MinExeInstrNum* 表示测试 10h 中,达到最大指令覆盖率和最大分支覆盖率的最小执行指令数量.

图 9 描述了各个测试目标 KLEE 和 OPT-SSE 的 *MaxInstrCoverage* 指标对比情况,图 10 描述了各个测试目标 KLEE 和 OPT-SSE 的 *MaxBrCoverage* 指标对比情况.

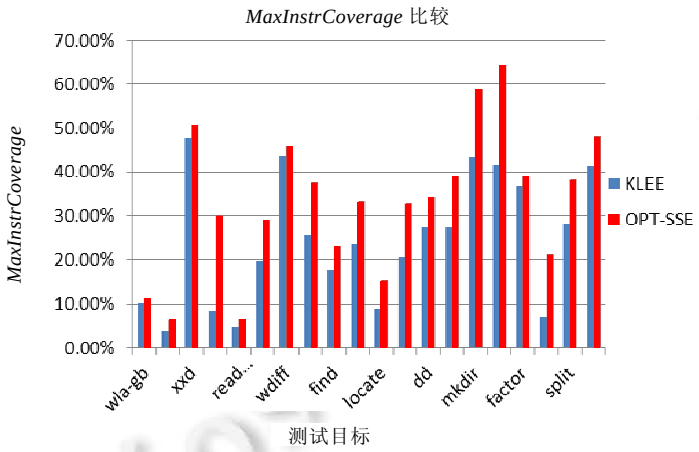


Fig.9 Instruction coverage comparison between OPT-SSE and KLEE on different subjects

图 9 OPT-SSE 和 KLEE 对不同目标的指令覆盖率比较

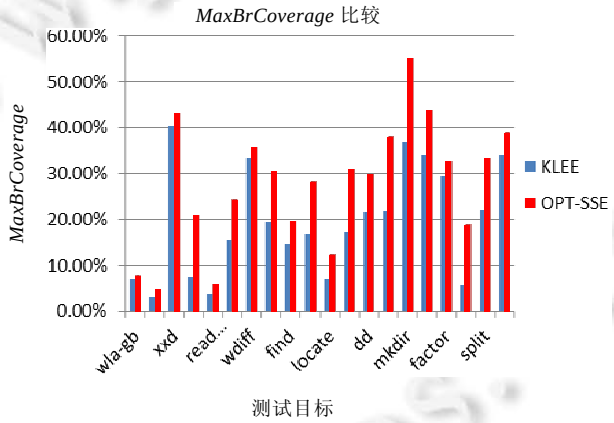


Fig.10 Comparison of branch coverage between OPT-SSE and KLEE on different subjects

图 10 OPT-SSE 和 KLEE 对不同目标的分支覆盖率比较

可以发现,与 KLEE 相比,OPT-SSE 在指令覆盖率和分支覆盖率上得到了一定程度的优化,其中,expr、unrtf、locate、vim 等目标在指令覆盖率和分支覆盖率提升最为显著,大约为 KLEE 的 1.5 倍~3.5 倍.另外可以发现,在 wdiff、which、dircolors 等多个目标上,出现了 OPT-SSE 的 *MinExeInstrNum* 指标比 KLEE 要小的情况.其原因可能是,OPT-SSE 在不同程序功能标签流上更早遇见较深的循环结构,使得 *MaxInstrCoverage* 和 *MaxBrCoverage* 两个指标一直得不到改善.但在整体覆盖率上,仍然比 KLEE 有所提升.

图 11 和图 12 分别给出了 wla-gb 在测试过程中指令覆盖率和分支覆盖率的实时变化情况.可以发现,指令覆盖率和分支覆盖率从测试开始到最后,OPT-SSE 比 KLEE 的优化效果明显.

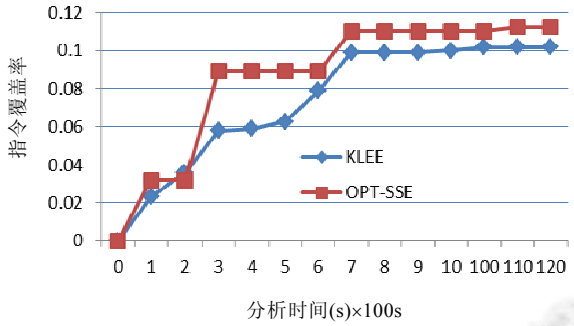


Fig.11 Comparison of instruction coverage and time overhead between OPT-SSE and KLEE when analyzing wla-gb

图 11 OPT-SSE 和 KLEE 分析 wla-gb 时的指令覆盖率和执行时间对比

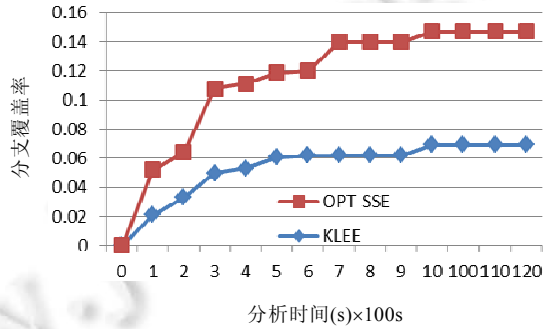


Fig.12 Comparison of branch coverage and time overhead between OPT-SSE and KLEE when analyzing wla-gb

图 12 OPT-SSE 和 KLEE 分析 wla-gb 时的分支覆盖率和执行时间对比

表 8 给出了 OPT-SSE 和 KLEE 在代码目标制导能力方面的对比情况.

Table 8 Comparison between OPT-SSE and KLEE on code goal-guided capability

表 8 OPT-SSE 和 KLEE 在代码目标制导能力上的对比

测试目标	测试方法	代码目标制导时间开销(平均值(s))	代码目标制导加速比	制导成功数量	成功率提升比例(%)
wla_dx	KLEE OPT-SSE	123.28 50.57	2.42X	12 18	30
Vim	KLEE OPT-SSE	88.43 62.20	1.43X	4 10	30
Unrtf	KLEE OPT-SSE	267.44 67.21	3.98X	7 9	15
Binutils	KLEE OPT-SSE	232.33 129.92	2.49X	6 12	30
Sed	KLEE OPT-SSE	24.50 15.23	1.61X	8 18	50
Wdiff	KLEE OPT-SSE	122.88 23.64	5.19X	6 15	45
Which	KLEE OPT-SSE	56.19 20.12	2.79X	12 12	0
Findutils	KLEE OPT-SSE	29.56 8.23	3.59X	8 19	55
Gzip	KLEE OPT-SSE	439.30 28.30	15.5X	3 8	25
Coreutils	KLEE OPT-SSE	91.52 27.04	3.38X	14 20	30

图 13 描述了 OPT-SSE 与 KLEE 相比,在代码目标制导速度和成功率上的提升情况.针对各个测试目标,本文利用静态分析[4−6]获取初步脆弱性可疑点集合,然后从集合中筛选 20 个脆弱点作为代码目标集,分别利用 OPT-SSE 和 KLEE 进行 1h 的制导分析.设置 1h 的时间上限,是考虑多个目标在 1h 后指令覆盖率变化很小.表中统计了各个目标分别在 KLEE 和 OPT-SSE 上的代码目标制导时间开销、代码目标制导加速比、制导成功数量以及成功率提升等指标的详细数据,其中,代码目标制导时间开销为 KLEE 和 OPT-SSE 都能成功制导的代码目标时间开销平均值.选择 KLEE 和 OPT-SSE 都能成功制导的代码目标,是为了更好地比较代码目标制导加速情况.本文设定:代码目标制导加速比=KLEE 的代码目标制导时间开销/OPT-SSE 的代码目标制导时间开销,成功率提升比例=(OPT-SSE 的制导成功数量−KLEE 的制导成功数量)/20.通过数据观察可以发现,在代码目标制导加速比方面,OPT-SSE 在每个目标上都有所提升;在成功率提升比例方面,除 which 外,其他目标都有显著的提升.

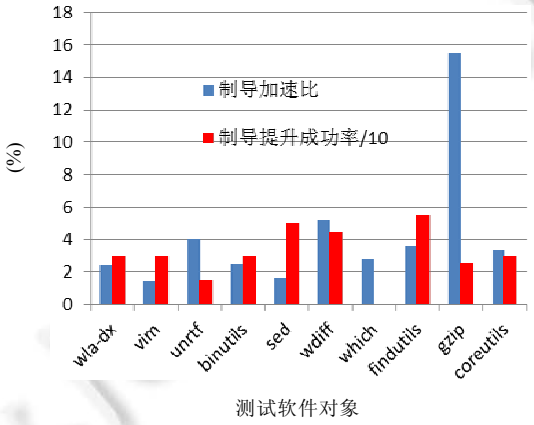


Fig.13 Improvement of OPT-SSE on code goal-guided speed and success rate compared with KLEE

图 13 OPT-SSE 与 KLEE 相比在代码目标制导速度和成功率上的提升情况

5 相关工作

近年来,符号执行在程序的正确性验证、缺陷的发现和重现、复用代码的检测、自动调试能力增强等方向产生了良好的应用,但由于存在程序路径爆炸增长和约束求解时间开销过大问题,制约其通用能力的发展.学术界和工业界一直把如何拓展符号执行的应用面和提升符号执行效率作为软件程序分析领域的基础性课题[7].符号执行在程序分析上的应用能力主要体现在以下几点.

(1) 不同程序语言上的应用

陆续产生了具备分析 c/c++、JAVA、JAVAscript、python 等语言能力的符号执行原型系统.如,斯坦福大学 Cadar 等人先后开发了 EXE[8]和 KLEE[9],这两个系统都应用于 c/c++程序对象.KLEE 重写了 EXE 的符号执行分析引擎,通过分析 c/c++程序目标的 llvm 字节码,提升了路径覆盖率和缺陷发现能力.NASA 的 Robust 软件工程开发小组开发了 JAVA pathfinder[10,11],通过分析 JAVA 程序的字节码,具备应用于并发性 JAVA 程序分析的能力.伯克利大学的 Saxena 等人设计了一种适用于字符串求解的约束语言 Kaluza[12],适用于求解各种事件空间以及数值空间,在此基础上,构建一个适用于 JAVAscript 语言的符号执行框架,具备检测命令注入缺陷的能力.洛桑联邦理工大学的 Bucur 等人开发了 Chef[13],通过对 python 解释器的相关包裹函数进行符号插桩处理,对 python 语言对象和解释器进行不同层次的符号执行,构建了适应于 python 语言的符号执行引擎.

(2) 不同系统平台上的应用

主要是针对 Linux 平台及 windows 平台,直接对二进制程序进行分析,通过把二进制翻译成符号执行引擎可识别的中间语言,可以消除不同程序语言的影响,并且适应于闭源程序对象.如,微软研究院的 Godefroid 等人开发了 DART[14]和 SAGE[15],专门用于 windows 平台的应用程序分析,展现了显著的效果.洛桑联邦理工大学的

Vitaly 等人开发了 S2E[16,17].该系统利用 qemu translator[18]将 Linux 二进制程序翻译成 llvm 字节码,再结合 KLEE 实现对 Linux 二进制程序的分析.卡耐基梅隆大学的 Sang 等人开发了 Mayhem[19],通过使用 Bap 平台[20],将 Linux 二进制转换成 BIL 语言,再结合符号执行引擎后端进行分析,在 debian 系统上发现大量的缺陷.

(3) 不同功能程序对象上的应用

包括对应用级程序、内核级程序、设备驱动级程序、固件级程序等不同功能程序的符号执行分析.如,经典符号执行引擎[4,5,10,14,21−23]只适应于应用级程序分析,S2E 通过对操作系统插桩处理并添加特权指令的支持,具备分析内核级程序的能力.洛桑联邦理工大学的 Kuznetsov 等人在 S2E 基础上构建了 DDT[24],通过对设备驱动程序相关接口进行有效配置,成功应用于设备驱动程序的分析.后来,康斯威星大学的 Matthew 等人开发了 symdrive[25].该系统对分析的设备中各种 I/O 操作、DMA 操作、中断操作进行符号化,并结合静态分析裁剪设备驱动程序的无关路径,提升了设备驱动级程序分析的有效路径覆盖能力.康斯威星大学 Davidson 等人开发的 FIE[26]以及 EURECOM 大学 Zaddach 等人开发的 Avatar[27],这些系统通过对符号执行引擎支持的运行环境对固件代码建模,成功地用于嵌入式系统上的固件代码的缺陷分析.

从符号执行技术应用发展趋势[28]可看出,符号执行应用的程序语言对象多元化,从支持高级语言到二进制语言,再到解释型语言,中间语言翻译平台[20,29−34]的不断发展,逐渐加强了符号执行在程序语言上的通用性.但是符号执行对于各种程序语言的分析效率仍然具有较大的提升空间,尤其是复杂、大规模程序对象的应用效果一直受制于路径空间爆炸和约束求解开销过大,在一定程度上,通过以下方法得以优化或缓解.

(1) 路径搜索策略的提升

符号执行需要采用特定的路径搜索策略进行状态遍历,一般经典符号执行引擎集成多个路径搜索策略.如,KLEE[9]中实现了深度优先、宽度优先、随机选择、多策略交替选择等路径搜索策略.深度优先选择策略容易搜索到完整的执行路径,但对程序的整体覆盖率较低;宽度优先选择策略会产生很多路径片断,很难分析到较深的路径;随机选择选择策略可以避免类似动态分析过程中由于碰见复杂外部库无法跳出来的情况;多策略交替选择策略通过采用交替地采用深度优先、宽度优先、随机选择策略分析一段固定的时间,结合了各自的优势.PEX 定义了一种适应度函数,用来评估待选择的分支离未覆盖过分支的距离,通过这个度量值选择最佳分支,取得的较高的覆盖率[35].SAGE 中采用一种约束产生式的状态遍历方法(generational-based),先否定所有的路径条件,然后逐渐将最深的路径条件反转回来,可以优先遍历到不同的深度路径[21].Li 等人[36]提出了优先选择路径分支执行重复频率最低的分支,可以在某些代码场景产生更高的覆盖率.

(2) 约束求解优化

KLEE 在求解模块采用路径预判断、约束表达式简化、约束集简化、反例缓存机制等技术降低求解开销.Romano 等人[37]建立了表达式匹配规则系统,尽量利用历史约束集信息确定新约束集是否可解,在一定程度上降低了对求解器的查询次数.S2PF[38]提出,避免一遇见条件分支就调用约束求解器,而是通过累积多个条件分支后再进行求解:如果约束集可解,则说明该执行路径上之前的约束集都可解,避免了之前的求解开销;如果约束集不可解,则回溯到之前没有求解的条件分支再次执行,从一定概率上节省了总的约束求解开销.秦晓军等人[39]提出了一种基于懒符号执行的约束求解算法,自动识别循环结构.通过推迟变量实例化等方法,有效地缓解了循环结构的路径组合爆炸问题,降低了求解次数.

(3) 冗余状态归约

Li 等人针对符号状态定义弱等价关系[40],如果一组状态满足该关系,那么可以用一个状态替代该组状态.例如,某循环体中的判定条件与符号变量有关,并且某个变量依赖于外部库的符号返回值,那么对于每次循环,都会产生一个新的符号变量,这样会出现大量呈弱等价关系的状态.这种方法在一定程度上加速了符号执行过程.Bugrara 等人[41]提出,如果一组约束覆盖的代码行和另一组约束相同,则认为这组约束对应的状态是多余的,可以消除,并利用动态切片方法[42]确定不同约束集是否覆盖相同代码行.该方法提升了代码覆盖率.

(4) 状态合并方法

如果两条路径在某个代码点交汇,那么可以把两个状态的约束集合并成一个状态的约束集.这样可以大量

削减状态的总数量,但有可能出现对合并后约束集的求解开销大于合并之前分别求解的总开销,尤其容易发生在大规模程序中[43].针对这一情况,出现了相应的折中方法,如,

- Boonstoppel 等人[44]提出,如果两个状态效果相同,并且约束集中取值不同的变量与的后续路径选择无关,那么这两个状态合并后约束集在后续求解中带来的额外开销会降低,可以选择合并符合该条件的状态集合.
- Kuznetsov 等人[45]提出,针对每次状态合并之前,评估合并后约束集中新增符号变量所增加的求解器查询次数,以此确定潜在的合并点.

(5) 执行分段和合成方法

Le 利用静态分析根据循环体、外部库等代码特征将程序片段化,再通过动态执行获取各个片段的相关接口信息以及自动合成各个片段,可以更好地处理程序循环结构和动态库带来的符号化问题[46].Ramos 等人[47,48]提出,分别对各个用户定义函数体进行符号执行,再对触发函数体缺陷的输入进行合成,避免直接从主函数分析时出现的深度路径不可达情况,可产生更高的语句覆盖率.Sinha 等人[49]提出,通过对并发性程序进行分阶段执行,有效分离线程的过程内和过程间操作.第 1 阶段获取线程全局变量信息,对线程序列进行有效分割;第 2 阶段利用执行序列的一致性把路径片段组合成完整线程,具备并发性缺陷发现能力.Zamfir 等人[50]提出,对并发性程序的序列路径合成和线程调度合成,可准确地重现并发性缺陷.

(6) Concolic 执行方法

该方法需要使用种子数据,即初始测试例,在对种子数据进行具体执行的过程中,收集执行路径的其他分支相关符号约束,构造后续状态集合,并在具体执行完种子数据后,采用特定状态选择策略进行符号执行.Concolic 执行方法可以借助具体执行特定的测试例快速制导具有一定深度的代码,可解决纯符号执行不适应复杂目标的深度路径可达问题,在多个符号执行引擎[8,9,14,15]中得到了体现.如,KLEE 中利用种子数据进行符号执行,一旦执行完种子数据,就利用特定搜索策略对种子数据的分支状态进行遍历;CUTE 采用不断随机生成测试例进行具体执行,一旦生成的测试例不能覆盖新的分支,将切换到符号执行[51],这种 Concolic 执行方法提升了 4 倍以上的覆盖能力;DASE[52]构造特定输入种子,并确定其中固定字段,在种子模式执行分析过程中,可避免对这些固定数据的约束求解,提高了分析效率.

(7) 并行化方法

Cloud9[53]在 KLEE 的基础上、MergePoint[54]在 Mayhem 的基础上分别构建了并行符号执行框架.他们采用在集群的某个节点上启动符号执行,动态地选择当前节点上产生的不同状态,并分离到集群上的其个节点上继续执行,实时保持不同节点上的负载均衡,取得了良好的效果.Junaid 等人[55]提出利用不同测试例划分程序路径范围,对不同程序路径范围进行并行性测试,大幅度提升覆盖率.

大部分符号执行性能优化方法只适应于特定的程序场景,需要结合多种优势才能应对程序对象多样性、复杂性的通用测试需求.针对一个路径数量过多的程序,不管采用何种路径选择策略,如果要做到全局覆盖,很难达到理想的效果,需要其他优化能力的不断提升.基于符号执行面临的现状,近几年出现了一种实用性较强的应用——制导符号执行分析方法.该方法不以程序的全局覆盖为目标,只对程序中感兴趣的程序片段或性质进行验证,适用于补丁程序的可靠性分析、特定场景的测试例生成、特定缺陷发现等应用方向.制导符号执行一般通过静态分析或附加的限制条件信息增强其分析的导向性,利用预定义好的制导规则指导符号执行如何进行更好的路径搜索达到制导目的,其效果更易于提升不同代码规模程序对象的应用.制导符号执行分析方法出现了一系列应用,如,Kin 等人[56]提出一种快速的程序行可达性判定和测试例生成算法,在静态控制流图上计算每个程序分支和目标代码行的距离值,每执行分支时,选择离目标代码最近的程序分支,能够快速到达目标代码行;Paul 等人[57]通过优先选择程序分支到敏感指令(读和写)的距离最近的分支进行制导分析,发现很多读写相关的缺陷;Zhang 等人[58]提出通过有限状态机模型对正规性质进行定义,利用符号执行对程序正规性质进行制导分析,有效地验证程序中内存泄漏等路径敏感缺陷;DiSE[59]先通过静态分析检查某软件系统不同版本间的差异,并利用制导符号执行产生差异代码可达路径条件,可以检查程序更改后带来的影响;Taneja 等人[60]通过构建

程序控制流图,确定与修改代码区域无关的条件分支,在符号执行过程中裁剪这些无关分支,能够更快地产生回归测试例,已验证程序修补后的正确性;Marinescu 等人[61,62]利用制导符号执行实现对已知补丁集合的进行覆盖测试,通过计算每条指令到补丁代码的条件数量作为度量距离值,采用最弱前置条件分析[63]确定目标不可达的基本块,最后对多个初始种子中选择出离补丁代码最近的种子数据,可以为后续符号执行制导补丁代码提供距离最近的状态;Domagoj 等人[64]先运用静态分析确定二进制程序中的脆弱点,再借助事先在控制流边上标记到达脆弱点的静态跳转次数信息对这些脆弱点集合进行制导符号执行分析;Ge 等人[65]利用制导符号执行分析方法对静态分析得到的缺陷报告进行动态验证,提高了静态分析的准确性;Guo 等人[66]通过静态分析确定程序正确的部分,在符号执行过程中对这些不包含缺陷的路径分支进行实时的裁剪,提升了分析速度.

6 结 论

本文提出了一种基于程序功能切片的符号执行制导分析方法 OPT-SSE.该方法参考程序的功能文档,在控制流图上把提取控制功能执行路径的关键基本块,并标识成相应的功能标签.OPT-SSE 利用功能标签流对程序进行有序划分,对于给定的代码目标点,提取与之相关的功能标签切片,然后根据预定义的制导规则对该切片进行制导分析.实验结果表明,该方法能够显著加速代码目标制导速度和成功率;并且通过并行的方式制导分析不同的程序功能切片,能够避免符号执行卡在某个循环结构体上,从而提升对整个程序的分支和指令覆盖率.

OPT-SSE 在下一步工作中需要对以下不足进行优化.

(1) 采取静态切片提取功能标签受影响的基本块.可能会分析出部分可能执行不到的代码片段,从而影响后续的符号执行制导效果,在下一步工作中需要借助动态切片进一步优化.

(2) 本文把程序的选项定义为功能标签,在今后的工作中,可以对功能标签进行更宽泛的应用,例如,把程序中所有不可变性质归纳为相应的功能标签,如固定输入格式和结构.利用更多的功能标签信息,可以进一步提升符号执行制导效率.

References:

[1] Nielson F, Nielson HR, Hankin C. Principles of Program Analysis. Berlin, Heidelberg: Spriner-Verlag, 1999.

[2] Chen TY. Adaptive random testing. In: Proc. of the Advances in Computer Science (ASIAN 2004). Berlin, Heidelberg: Spriner-Verlag, 2005. 320−329.

[3] King JC. Symbolic execution and program testing. Communications of the ACM, 1976,19(7):385−394.

[4] Gan ST, Qin XJ, Chen ZN, Wang LZ. Software vulnerability code clone detection method based on characteristic metrics. Ruan Jian Xue Bao/Journal of Software, 2015,26(2):348−363 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/4786.htm [doi: 10.13328/j.cnki.jos.004786]

[5] Qin XJ, Gan ST, Chen ZN. A static detecting technology of software code secure vulnerability based on first-order logic. SCIENTIA SINICA Informationis, 2014,44(1):108−129 (in Chinese with English abstract).

[6] Bessey A, Block K, Chelf B, *et al*. A few billion lines of code later: Using static analysis to find bugs in the real world. Communications of the ACM, 2010,53(2):66−75.

[7] Cadar C, Godefroid P, Khurshid S, *et al*. Symbolic execution for software testing in practice:preliminary assessment. In: Proc. of the 33rd Int'l Conf. on Software Engineering. ACM Press, 2011. 1066−1071.

[8] Cadar C, Ganesh V, Pawlowski PM, *et al*. EXE: Automatically generating inputs of death. ACM Trans. on Information and System Security, 2006,12(2):322−335.

[9] Cadar C, Dunbar D, Engler D. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In: Proc. of the 8th USENIX Conf. on Operating Systems Design and Implementation. USENIX Association, 2008. 209−224.

[10] Reanu CS, Mehlitz PC, Bushnell DH, *et al*. Combining unit-level symbolic execution and system-level concrete execution for testing NASA software. In: Proc. of the 2008 Int'l Symp. on Software Testing and Analysis. 2008. 15−26.

[11] Păsăreanu CS, Rungta N. Symbolic PathFinder: Symbolic execution of Java bytecode. In: Proc. of the IEEE/ACM Int'l Conf. on Automated Software Engineering. 2010. 179−180.

[12] Saxena P, Akhawe D, Hanna S, Mao F, McCamant S, Song D. A symbolic execution framework for JavaScript. In: Proc. of the 2010 IEEE Symp. on Security and Privacy (SP). IEEE, 2010. 513−528.

[13] Bucur S, Kinder J, Candea G. Prototyping symbolic execution engines for interpreted languages. ACM SIGPLAN Notices, 2014, 49(4):239−254.

[14] Godefroid P, Klarlund N, Sen K. Dart: Directed automated random testing. In: Proc. of the 2005 ACM SIGPLAN Conf. on Programming Language Design and Implementation. 2005. 213−223.

[15] Godefroid P, Levin MY, Molnar D. SAGE: Whitebox fuzzing for security testing. Queue, 2012,10(3):40−44.

[16] Chipounov V, Kuznetsov V, Candea G. S2E: A platform for in-vivo multi-path analysis of software systems. ACM SIGPLAN Notices, 2011,39(1):265−278.

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

[18] Chipounov V, Candea G. Dynamically translating x86 to LLVM using QEMU. In: Proc. of the Dynamic Binary Translator. 2010.

[19] Cha SK, Avgerinos T, Rebert A, *et al*. Unleashing mayhem on binary code. In: Proc. of the IEEE Symp. on Security & Privacy. IEEE Computer Society, 2012. 380−394.

[20] Nethercote N, Seward J. Valgrind: A framework for heavy-weight dynamic binary instrumentation. In: Proc. of the Proerence on Computer Aided Verification. 2011.

[21] Godefroid P, Levin M, Molnar D. Automated whitebox fuzz testing. In: Proc. of the 15th Annual Network and Distributed System Security Symp. 2008.

[22] Sen K, Marinov D, Agha G. Cute: A concolic unit testing engine for c. In: Proc. of the 10th European Software Engineering Conf., Held Jointly with 13th ACMSIGSOFT Int'l Symp. on Foundations of Software Engineering. 2005. 263−272.

[23] Nikolai T, de Halleux J. Pex-white box test generation for .NET. In: Proc. of the Tests and Proofs, Second International Conference (TAP 2008). 2008. 134−153.

[24] Kuznetsov V, Chipounov V, Candea G. Testing closed-source binary device drivers with DDT. In: Proc. of the USENIX Annual Technical Conf. 2010. 4−5.

[25] Renzelmann MJ, Kadav A, Swift MM. SymDrive: Testing drivers without devices. In: Proc. of the 10th USENIX Conf. on Operating Systems Design and Implementation. USENIX Association, 2012. 279−292.

[26] Davidson D, Moench B, Jha S, Ristenpart T. FIE on firmware: Finding vulnerabilities in embedded systems using symbolic execution. In: Proc. of the SEC 2013 22nd USENIX Conf. on Security. 2013. 463−478.

[27] Zaddach J, Bruno L, Francillon A, Balzarotti D. Avatar: A framework to support dynamic security analysis of embedded systems' firmwares. In: Proc. of the Network and Distributed System Security Symp. 2014. 23−26.

[28] Cadar C, Sen K. Symbolic execution for software testing: Three decades later. Communications of the ACM, 2013,56(2):82−90.

[29] Necula GC, McPeak S, Rahul SP, Weimer W. CIL: Intermediate language and tools for analysis and transformation of C programs. In: Proc. of the 2007 ACM SIGPLAN Conf. on Programming Language Design and Implementation. New York: ACM Press, 2007. 89−100.

[30] Brumley D, Jager I, Avgerinos T, *et al*. BAP: A binary analysis platform. In: Proc. of the Computer Aided Verification. Berlin, Heidelberg: Spriner-Verlag, 2011. 463−469.

[31] Zhao J, Nagarakatte S, Martin MMK, Zdancewic S. Formalizing the LLVM intermediate representation for verified program transformations. In: Proc. of the 39th Annual ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages (POPL 2012). New York: ACM Press, 2012. 427−440.

[32] Vanegue J. Static binary analysis with adomain specific language. In: Proc. of the EKOPARTY 2008. 2008.

[33] Balakrishnan G, Gruian R, Reps T, Teitelbaum T. Codesurfer/X86—A platform for analyzing x86 executables. In: Proc. of the Int'l Conf. on Compiler Construction (CC 2005). 2005. 250−254.

[34] Dullien T, Porst S. REIL: A platform-independent intermediate representation of disassembled code for static code analysis. In: Proc. of the CanSecWest 2009. 2009.

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

[36] Li Y, Su Z, Wang L, *et al*. Steering symbolic execution to less traveled paths. In: Proc. of the 2013 ACM SIGPLAN Int'l Conf. on Object Oriented Programming Systems Languages & Applications (OOPSLA 2013). 2013. 19−32.

[37] Romano A, Engler D. Expression reduction from programs in a symbolic binary executor. In: Proc. of the Model Checking Software. Berlin, Heidelberg: Spriner-Verlag, 2013. 301−319.

[38] Zhang YF, Chen ZB, Wang J. S2PF: Speculative symbolic PathFinder. ACM Sigsoft Software Engineering Notes, 2012,37(6):1−5.

[39] Qin XJ, Zhou L, Chen ZN, Gan ST. Software vulnerable trace's solving algorithm based on lazy symbolic execution. Chinese Journal of Computers, 2015,38(11):2290−2230 (in Chinese with English abstract).

[40] Li Y, Cheung SC, Zhang X, *et al*. Scaling up symbolic analysis by removing z-equivalent states. ACM Trans. on Software Engineering & Methodology, 2014,23(4):1−32.

[41] Bugrara S, Engler D. Redundant state detection for dynamic symbolic execution. In: Proc. of the USENIX Annual Technical Conf. 2013. 199−211.

[42] Agrawal H, Horgan JR. Dynamic program slicing. ACM Sigplan Notices, 1990,25(6):246−256.

[43] Hansen T, Schachte P, Søndergaard H. State joining and splitting for the symbolic execution of binaries. In: Proc. of the Runtime Verification. Berlin, Heidelberg: Spriner-Verlag, 2009. 76−92.

[44] Boonstoppel P, Cadar C, Engler D. RWset: Attacking path explosion in constraint-based test generation. In: Proc. of the Theory and Practice of Software, 14th Int'l Conf. on Tools and Algorithms for the Construction and Analysis of Systems. Berlin, Heidelberg: Springer-Verlag, 2008. 351−366.

[45] Kuznetsov V, Kinder J, Bucur S, *et al*. Efficient state merging in symbolic execution. In: Proc. of the ACM SIGPLAN 2012 Conf. on Programming Language Design and Implementation (PLDI 2012). 2012. 193−204.

[46] Le W. Segmented symbolic analysis. In: Proc. of the 2013 35th Int'l Conf. on IEEE Software Engineering. 2013. 212−221.

[47] Ramos DA, Engler D. Under-constrained symbolic execution: Correctness checking for real code. In: Proc. of the 2015 24th USENIX Conf. on Security Symp. 2015. 49−64.

[48] Ramos DA, Engler DR. Practical, low-effort equivalence verification of real code. In: Proc. of the Computer Aided Verification. Berlin, Heidelberg: Spriner-Verlag, 2011. 669−685.

[49] Sinha N, Wang C. Staged concurrent program analysis. In: Proc. of the 18th ACM Sigsoft Int'l Symp. on Foundations of Software Engineering. ACM Press, 2010. 47−56.

[50] Zamfir C, Candea G. Execution Synthesis: A technique for automated software debugging. In: Proc. of the Eurosys. 2009. 321−334.

[51] Majumdar R, Sen K. Hybrid concolic testing. In: Proc. of the 29th Int'l Conf. on Software Engineering. Washington: IEEE Computer Society, 2007. 416−426.

[52] Wong E, Zhang L, Wang S, *et al*. DASE: Document-assisted symbolic execution for improving automated software testing. In: Proc. of the 2015 IEEE/ACM 37th IEEE Int'l Conf. on Software Engineering (ICSE). IEEE, 2015. 620−631.

[53] Bucur S, Ureche V, Zamfir C, *et al*. Parallel symbolic execution for automated real-world software testing. In: Proc. of the 6th Conf. on Computer Systems. ACM Press, 2011. 183−198.

[54] Avgerinos T, Rebert A, Sang KC, *et al*. Enhancing symbolic execution with veritesting. In: Proc. of the ICSE 2014. 2014. 1083−1094.

[55] Siddiqui JH, Khurshid S. Scaling symbolic execution using ranged analysis. ACM Sigplan Notices, 2012,47(10):523−536.

[56] Ma KK, Phang KY, Foster JS, Hicks M. Directed symbolic execution. In: Proc. of the 18th Int'l Static Analysis Symp. (SAS). Venice: Berlin, Heidelberg: Spriner-Verlag, 2011.

[57] Marinescu PD, Cadar C. Make test-zesti: A symbolic execution solution for improving regression testing. In: Proc. of the 2012 Int'l Conf. on Software Engineering. 2012. 716−726.

[58] Zhang Y, Clien Z, Wang J, *et al*. Regular property guided dynamic symbolic execution. In: Proc. of the 2015 IEEE/ACM 37th IEEE Int'l Conf. on Software Engineering (ICSE). IEEE, 2015. 643−653.

[59] Yang G, Person S, Rungta N, *et al*. Directed incremental symbolic execution. ACM Sigplan Notices, 2011,46(1):504−515.

[60] Taneja K, Xie T, Tillmann N, *et al*. eXpress: Guided path exploration for efficient regression test generation. In: Proc. of the Companion ICSE. 2011. 311−314.

[61] Marinescu PD, Cadar C. High-coverage symbolic patch testing. In: Proc. of the 19th Int'l Conf. on Model Checking Software. Springer-Verlag, 2012. 7–21.

[62] Marinescu PD, Cadar C. KATCH: High-coverage testing of software patches. In: Proc. of the 2013 9th Joint Meeting on Foundations of Software Engineering. ACM Press, 2013. 235–245.

[63] Dijkstra EW. A Discipline of Programming. Prentice Hall, Inc., 1976.

[64] Babic D, Martignoni L, McCamant S, Song D. Statically-directed dynamic automated test generation. In: Proc. of the 2011 Int'l Symp. on Software Testing and Analysis. ACM Press, 2011. 12–22.

[65] Guo SJ, Kusano M, Wang C, Yang ZJ, Gupta A. Assertion guided symbolic execution of multithreaded programs. In: Proc. of the ACM SIGSOFT Symp. on the Foundations of Software Engineering (FSE). 2015.

[66] Ge X, Taneja K, Xie T, *et al*. DyTa: Dynamic symbolic execution guided with static verification results. In: Proc. of the Int'l Conf. on Software Engineering. 2011. 992–994.

附中文参考文献:

[4] 甘水滔,秦晓军,陈左宁,王林章.一种基于特征矩阵的软件脆弱性代码克隆检测方法.软件学报,2015,26(2):348−363. http://www.jos.org.cn/1000-9825/4786.htm [doi: 10.13328/j.cnki.jos.004786]

[5] 秦晓军,甘水滔,陈左宁.一种基于一阶逻辑的软件代码安全性缺陷静态检测技术.中国科学:信息科学,2014,44(1):108−129.

[39] 秦晓军,周林,陈左宁,甘水滔.基于懒符号执行的软件脆弱性路径求解算法.计算机学报,2015,38(11):2290−2230.

甘水滔(1986－),男,江西靖安人,博士,工程师,CCF 专业会员,主要研究领域为网络安全,系统安全,软件理论.

秦晓军(1975－),男,博士,高级工程师,主要研究领域为网络安全,软件理论,操作系统.

王林章(1973－),男,博士,教授,CCF 杰出会员,主要研究领域为模型驱动的软件测试与验证,安全测试,软件测试自动化.

周林(1986－),男,博士,工程师,CCF 专业会员,主要研究领域为网络安全.

谢向辉(1958－),男,博士,研究员,博士生导师,CCF 专业会员,主要研究领域为计算机系统结构,并行计算.

陈左宁(1957－),女,研究员,博士生导师,中国工程院院士,CCF 会士,主要研究领域为操作系统原理,并行计算.