软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2015,26(1):1−13 [doi: 10.13328/j.cnki.jos.004658]

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

相关路径静态分析中协同式逆向推理方法*

郭 曦[1,3], 王 盼[2]

[1](华中农业大学 信息学院 计算机科学系,湖北 武汉 430070)

[2](武汉电力职业技术学院,湖北 武汉 430079)

[3](School of Computer Science, College of Computing, Georgia Institute of Technology, Atlanta 30332, USA)

通讯作者: 郭曦, E-mail: seyeyesx@163.com

摘 要: 相关路径生成,是程序动态分析中的一种重要方法.通过对目标执行路径的获取和分析来生成与其相关的近邻执行路径,在程序行为特征分析、编译优化和调试等研究方向有重要的作用.现有的方法主要通过改变路径节点序列来生成近邻的路径集合,由于缺乏关键节点的路径引导信息,导致生成大量冗余或者无效的路径集合.提出采用协同式逆向分析的近邻路径生成方法,针对目标路径的后置条件,采用逆向符号分析方法产生程序各个基本块的前置条件作为执行路径的引导信息.同时,通过调整距离因子 k 的取值,可以有针对性地生成与目标路径的编辑距离不超过 k 的近邻路径集合.实验结果表明:与现有方法相比,该方法在准确性和效率方面有明显的优势.

关键词: 逆向分析;近邻路径;最弱前置条件;符号执行

中图法分类号: TP311

中文引用格式: 郭曦,王盼.相关路径静态分析中协同式逆向推理方法.软件学报,2015,26(1):1−13. http://www.jos.org.cn/1000-9825/4658.htm

英文引用格式: Guo X, Wang P. Technique of cooperative reverse reasoning in related path static analysis. Ruan Jian Xue Bao/Journal of Software, 2015,26(1):1−13 (in Chinese). http://www.jos.org.cn/1000-9825/4658.htm

Technique of Cooperative Reverse Reasoning in Related Path Static Analysis

GUO Xi[1,3], WANG Pan[2]

[1](Department of Computer Science, College of Informatics, Huazhong Agriculture University, Wuhan 430070, China)

[2](Wuhan Electric Power Technical College, Wuhan 430079, China)

[3](School of Computer Science, College of Computing, Georgia Institute of Technology, Atlanta 30332, USA)

Abstract: Related execution path generation, which generates the similar execution path according to the acquisition and analysis of the target execution path, is a key technique in the dynamic program analysis, and it is important to the domain of program characteristic analysis, compilation optimization and debugging. Current analysis mainly generates the similar execution path by altering the node list of the path, but lacks the guiding information of the key node, and thus a lot of redundant and infeasible paths are generated. A technique of k similar paths generation based on cooperative reverse analysis is proposed. Aiming at the post-condition of the target paths, the pre-condition of the basic block of the program is calculated by the reverse symbolic analysis, which can be used as the guidance information of the execution paths. Meanwhile, the similar paths that are k distance from the target execution path can be obtained. Experimental results show that the proposed method has an obvious advantage in the aspects of accuracy and efficiency.

Key words: reverse analysis; similar path; weakest precondition; symbolic execution

* 基金项目: 国家自然科学基金(61173138, 61272452, 91118003, 61003268); 湖北省自然科学基金(2014CFB144); 中央高校基本科研业务费专项资金(0900206154); 武汉大学博士研究生短期出国(境)研修专项经费

收稿时间: 2014-02-08; 修改时间: 2014-04-03; 定稿时间: 2014-05-21; jos 在线出版时间: 2014-08-19

CNKI 网络优先出版: 2014-08-19 14:32, http://www.cnki.net/kcms/doi/10.13328/j.cnki.jos.004658.html

随着软件规模的日益扩大,在软件调试和测试阶段所投入的成本已经上升到总投入的 60%以上[1],其中,对程序执行路径的分析是软件调试和测试阶段的重要内容.为了能够高效地分析程序行为特征等信息,研究人员通过生成测试用例的方式来对程序的执行路径进行分析;同时,对执行路径进行比较,以识别不可行的近邻路径条件、构建代码的行为特征和轮廓.近邻路径的生成方法已经成为程序动态分析领域的重点和难点问题.

近邻路径的生成方法主要包括静态分析方法和动态分析方法.

静态分析方法通过构造程序的控制流图(control flow graph,简称 CFG)分析程序执行路径在 CFG 中所经过的节点来生成近邻的路径集合.文献[2]使用静态反汇编的分析方法,以程序的基本块(basic block)为单位进行分析,该方法通过记录数据的起始地址和访问地址来进行路径的相似性比对.由于该方法需要进行反汇编分析,故对反汇编的准确性提出了较高的要求;同时,程序中的循环结构在 CFG 中的表示方法也会对静态分析的精度产生直接的影响.静态分析需要对程序的执行语义进行抽象,其抽象语义是完整程序语义的近似,导致抽象过程具有较大的计算开销,故在实际分析过程中具有较多的限制.

动态分析方法通过分析程序的执行序列来生成近邻的执行路径,主要方法为计算序列的编辑距离,代表性的工作为 Bayer 等人提出的可伸缩的程序分析方法[3],该方法采用污点跟踪技术提取程序的执行序列,通过与执行路径关联的输入数据进行路径的相似性比对.现有的动态分析方法虽然比静态分析方法有明显的优势,但在实际分析过程中,传统的符号执行(symbolic execution,简称 SE)[4]等路径生成方法仅仅通过改变路径谓词的方法来改变路径的执行轨迹,对于因路径分支增加而出现的程序状态空间爆炸问题缺乏有效的分析手段,故在针对目标路径所生成的近邻路径集合中,由于缺乏路径引导信息而包含了较多的冗余路径或无关的路径.

针对以上问题,本文提出一种基于协同式逆向分析的 k 近邻路径生成方法,通过使用后向符号分析方法分析 CFG 语句中变量的取值范围,以生成程序的输入集合.最弱前置条件(weakest precondition,简称 WP)[5]作为一种后向符号分析方法,可以对程序结构进行建模并分析其执行语义,通过对程序中每条语句的后置断言和该语句表达式的计算,生成该语句的最弱前置条件,即,当前语句的输入集合.符号执行工具在最弱前置条件信息的引导下,可以有效地减少因枚举 CFG 所有路径而出现的无效路径过多的问题,生成与目标路径近邻的可行路径集合.为了量化路径之间的近邻程度,通过引入因子 k 来计算路径之间的距离.通过调整 k 的取值,可以生成与目标路径的编辑距离不超过设定值的路径集合.在最弱前置条件计算过程中,循环结构的分析对计算结果有直接的影响,在一定程度上,CFG 中循环结构的执行次数可以通过因子 k 来表示.故,本文的方法相对于传统的最弱前置条件方法有较好的适应性.

本文的第 1 节介绍国内外相关的研究进展.第 2 节介绍与本文相关的概念,着重分析最弱前置条件的性质.第 3 节讨论 k 近邻路径的生成方法.第 4 节为实验结果与分析.最后总结全文.

1 相关工作

程序分析的目标是验证程序所具有的性质,例如指定语句的可达性、多线程程序中的变量之间是否存在互斥或竞争等.传统的人工分析方法要求研究人员理解程序的功能、结构、语义,并采用合理的研究方法,故分析效率较低.目前的程序分析都采用自动化或半自动化的分析方法,如定理证明[6]、模型检测[7]、约束求解[8]等.程序分析中的一个重要问题就是近邻路径的生成问题,通过分析程序输入变量的取值范围,使用改变路径谓词的方法,使程序沿着指定的路径执行.

目前的近邻路径分析方法主要有静态的程序结构比较方法和动态的程序行为比较方法.

静态的程序结构分析方法通常采用反汇编等方法对程序进行分析,常用的反汇编分析工具有 IDA Pro,Objdump 等.文献[9]通过比较程序中系统调用的地址的方法来分析路径的相似性.文献[10]从程序中的函数与调用图的角度来判断程序的相似性.文献[2]以基本块为单位来比较代码的相似性,然后通过分析操作数据的地址以及访问长度的方法进行扩展,该方法对于有较大范围改动的代码的分析效率较低.文献[11]也以基本块为单位进行分析,通过分析程序的控制流图,并采用符号执行的方法对程序的执行语义进行分析,但是由于缺乏对循环变量等关键数据结构的处理,从而使分析的结果不够精确.由于静态的分析方法在处理复杂数据结构方面

效率较低,且缺乏对程序的执行语义和程序上下文逻辑关系的分析,在代码相似性分析和近邻路径生成方面有较大的局限性.

动态的程序行为比较方法相对于静态的分析方法有明显的优势,它通过监控程序中敏感数据的状态信息和可能的流动来对程序的行为进行相似性比较,可以对更加复杂的程序数据结构和行为特征进行有效的分析(例如路径混淆或者代码加壳等).文献[12]通过分析程序状态的变化来描述程序的执行语义,同时提取执行序列以比较执行的相似程度.PSE[13]通过程序过程间的后向符号分析来对程序中的路径条件进行计算,但该方法产生的路径条件并不完备,对执行路径的生成有一定的影响.ESC/Java[14]通过定理证明的方法分析执行语句的前置条件和后置条件对当前语句的影响,但该方法需要通过手动插桩的方法来引导路径的分析.DSD-Crasher[15]从生成程序反例的角度分析路径的相似性,但该方法在生成最弱前置条件过程中可能受到其他变量的影响而导致路径条件不可解.文献[16]使用符号执行和最弱前置条件相结合的分析方法引导相似执行路径的生成,但是缺乏对最弱前置条件的约简分析.相对于以上方法,本文的方法通过对程序中循环变量谓词等关键数据结构的分析,使用后置条件和语句的语义信息生成对应的最弱前置条件,并逐步逆向分析到程序的起始位置,获得程序输入变量的有效输入域;再以最弱前置条件为引导信息,采用符号执行的方法生成与指定路径近邻的路径集合,同时通过距离因子 k 来调整生成的近邻路径与目标路径的编辑距离.与传统的符号执行方法相比,本文的方法有更好的灵活性,且生成的路径集合的冗余度和可行路径的比例有明显的优势.

2 协同式逆向分析

定义 1(控制流图(control flow graph,简称 CFG)). 程序的控制流图可以形式化地表示为(N,E,s,f),它是由若干节点和边组成的有向图,其中,N为节点的集合,E为边的集合,s为起始节点,f为出口节点.控制流图是程序执行序列的抽象表示,每个节点 $n\in N$ 代表一个基本块,包含了顺序语句、循环语句、条件语句等;边的集合 $E=N\times N$ 反映了基本块之间的控制流关系,每一个控制流图均有唯一的起始节点 s 和出口节点 f.

定义 2(关联变量(related variable)). 记控制流图中与节点 n 相关联的节点集合为 $N_r=\{n_i|\langle n_i,n\rangle\in E,\langle n,n_i\rangle\in E\}$,其中,$N_r$所包含的语句集合记为 $Stmt(N_r)$.$Stmt(N_r)$中的变量组成的集合即为节点 n 的关联变量,记为 $Rel(n)$.

定义 3(路径条件(path condition,简称 PC)). 在控制流图中,通过改变条件语句和循环语句的关键谓词可以产生新的路径,故路径条件是控制流在迭代操作过程中所产生的数据流值.条件语句集合 S 或循环语句集合 L 在判定节点 n 处的路径条件定义为 $PC(n)=\{v|v\in Rel(n)\wedge Stmt(n)\in S,v\in Rel(n)\wedge Stmt(n)\in L\}$,故路径条件实际上是由路径变量所组成的一阶逻辑公式.

定义 4(静态单赋值(static single assignment,简称 SSA)). 静态单赋值通过修改变量 a 在每次定义时的下标$(a_0,a_1,\ldots)$以使得程序中的每个变量都有唯一的一处定义,可以提高数据流分析和程序优化的效率和精度.在控制流图中,通过加入φ函数节点:$a=\varphi(a_0,a_1,\ldots)$来实现变量的定义和使用,即“定义-使用链(use-define chain)”,能够更加精确地描述变量的定义和使用之间的关系.在对程序的符号变量进行分析的过程中,通过使用静态单赋值的分析方法,可以消除控制流图中冗余的依赖关系,并区分关联变量的定义、使用和在内存中的地址.在对循环语句等复杂数据结构进行分析的过程中,由于控制流图具有单一的起始节点和出口节点,故当循环结构出现异常情况时,则在控制流图中有一条指向出口节点的边,以提高符号分析处理异常退出情况的能力.

定义 5(最弱前置条件(weakest precondition,简称 WP)). 在验证程序所具有的属性过程中,每一条语句的前后都有一个由变量的逻辑表达式所组成的约束条件,语句中的变量依据执行语义必须满足对应的约束条件.其中,语句前面的谓词是程序在当前语句中变量的约束条件,即前置条件;相应地,语句后面的谓词是语句执行后所满足的新的约束条件,即后置条件.在后向符号分析过程中,通过已知的后置条件来推理当前语句的前置条件,其中,最弱前置条件是确保一条语句在正常执行后能够使得后置条件有效的最小前提条件.在实际分析过程中,最后一条执行语句的后置条件是程序变量在执行完时的输出域.通过该后置条件和程序最后一条执行语句来计算其最弱前置条件,该前置条件又作为上一条语句的后置条件.通过这样的迭代计算,一直到控制流图的起始节点.

下面通过对程序中最常见的语句形式:赋值语句、顺序语句、分支语句和循环语句的诠释来分析最弱前置条件的表示形式.对于赋值语句 $x=E$,设 Q 是其后置条件,则前置条件 P 可以定义为 $P=Q_{x\to E}$,表示为 $\{Q_{x\to E}\}x=E\{Q\}$.在计算过程中,使用 E 来替换所有 x 的实例,即,使用函数$\varphi[x/E]$来表示该替换过程.在控制流图中,若一个节点有出边,则会使用相应的φ函数来执行替换操作.在实际分析过程中,往往会出现已知的前置条件和经过计算所得出的前置条件不一致的情况,故对于断言 P 和 P',若 P'逻辑蕴含 P:$P'\vdash P$,可以使用以下形式来表示推理的过程:

$$(\{P\}S\{Q\},P'\vdash P,Q\vdash Q')\vdash(\{P'\}S\{Q'\}).$$

对于相邻的顺序语句 S_1 和 S_2,设 S_1 和 S_2 的前置条件和后置条件具有如下形式:$\{P_1\}S_1\{P_2\},\{P_2\}S_2\{P_3\}$,则顺序语句的推理规则为$(\{P_1\}S_1\{P_2\},\{P_2\}S_2\{P_3\})\vdash(\{P_1\}S_1,S_2\{P_3\})$.该规则表明:经计算所得到的第 2 条语句的前置条件可以作为第 1 条语句的后置条件,从而得到第 1 条语句的前置条件.具体地,对于赋值语句序列 S_1:$x_1=E_1$,S_2:$x_2=E_2$,其前置条件和后置条件按照计算的顺序可以表示为

$$\{P_{3\,x_2\to E_2}\}x_2=E_2\{P_3\},\{(P_{3\,x_2\to E_2})_{x_1\to E_1}\}x_1=E_1\{P_{3\,x_2\to E_2}\}.$$

即,语句序列 S_1 和 S_2 在后置条件 P_3 的作用下得到的最弱前置条件为 $(P_{3\,x_2\to E_2})_{x_1\to E_1}$.

分支语句的形式可以表示为 **if** *Exp* **then** S_1 **else** S_2,由于布尔表达式 *Exp* 的真值可能为 True 或 False,故需要分析 **then** 和 **else** 两个分支条件,其推理规则为

$$(\{Exp\wedge P\}S_1\{Q\},\{\neg Exp\wedge P\}S_2\{Q\})\vdash(\{P\}\ \textbf{if}\ Exp\ \textbf{then}\ S_1\ \textbf{else}\ S_2\{Q\}).$$

由于循环语句循环体的执行次数受制于循环条件的取值,故循环体执行的次数决定了不同的执行路径.循环语句中的最弱前置条件是通过循环不变式的计算来获得的,循环不变式的取值不受循环条件和循环体中语句的影响.循环语句的语义可以表示为$\{P\}$ **while** *Exp* **do** *S* **end** $\{Q\}$,前置条件 P 使得循环体在结束循环时,后置条件 Q 是可满足的.对于循环不变式 I,循环语句的推理规则为$(\{I\wedge Exp\}S\{I\})\vdash(\{I\}$ **while** *Exp* **do** *S* **end** $\{I\wedge\neg Exp\})$,循环语句的最弱前置条件必须使得循环不变式 I 为真,同时,I 必须使得后置条件是可满足的.循环结构需要以下条件是可满足的:$P\vdash I$,$\{I\wedge Exp\}S\{I\}$,$(I\wedge\neg Exp)\vdash Q$.在实际分析过程中,通过循环体的执行来计算前置条件,由于循环不变式 I 往往需要通过归纳法来获得,通常情况下,前置条件 P 可以作为循环不变式 I,故 I 需要对后置条件进行弱化,同时使得控制流图中循环体的起始节点在循环开始时是可满足的,且在退出循环体时,循环体出口节点的后置条件也是可满足的.

在最弱前置条件的计算过程中,谓词转换函数 wp 用来将当前语句和谓词转换为另一个谓词,即 $wp(Stmt,Q)=P$.当程序中出现赋值语句时,则在最弱前置条件计算过程中需要使用φ函数来实现相应的转换.由于最弱前置条件是一种后向符号分析方法,需要将控制流图的出口节点语句和后置条件作为转换函数 wp 的参数,通过计算来获得其前置条件.该算法从出口基本块开始逐步迭代到入口基本块,所经过的节点的后置条件均转换为可以求解的最弱前置条件.算法中使用变量 *List* 来表示待求解的后置条件集合,当迭代求解到起始节点时,*List* 集合里面的后置条件均经过转换函数 wp 处理为对应的最弱前置条件,故该循环的终止条件是 *List* 集合为空.其中,*simplify* 函数的作用是对约束条件集合进行约简操作,将不可解的谓词条件从集合中排除.常见的约简有:

- $(x\leqslant y)\wedge(x\neq y)\to x<y$;
- $(typeOf(x)=T)\wedge(x=null)\to$false;
- $subType(a,t_1)\wedge subType(a,t_2)\wedge subType(t_1,t_2)\to subType(a,t_1)$.

传统的程序调用图(call graph)在构建过程中直接通过分析过程调用来实现,但不可避免地会分析大量重复的过程调用,导致调用图中存在冗余的边.故在最弱前置条件计算过程中,本文以待分析的控制流图中的节点为驱动,通过最弱前置条件分析的结果为引导,生成与待分析节点相关的精简的程序调用图.算法 1 为加入后向符号分析的程序调用图的生成算法.

算法 1. 带过程间最弱前置条件的程序调用图生成算法.

Input: *CFG*, φ_{post};

Output: Call graph under *wp*.

callGraphWithWP(CFG,φ_{post})

1. var $CG\leftarrow wp(CFG,\varphi_{post})$;
2. *var* $CG'\leftarrow\{cg\in CG|cg$ can be solved$\}$;
3. **if** $CG'=\varnothing$
4. **foreach** $m\in CG$ **do**
5. select one *method* from *m*;
6. select $s\notin subroutines$ in $method\wedge(m\wedge method=s$ can be solved);
7. **if** *s* does not exist
8. **continue**;
9. **endif**
10. $CFG'\leftarrow CFG+s$;
11. **return** *callGraphWithWP*(CFG',φ_{post});
12. **endfor**
13. **else**
14. **return** CG';

最弱前置条件在计算过程中所遵守的语法为:

Program::=*Block*+

Block::=*BlockId*:*Stmt*; goto *BlockId**

Stmt::=*Stmt*; *Stmt*|*VarId*:=*Exp*|*assert Exp*|*assume Exp*|*skip*

程序由若干个控制流图中的基本块组成,每个基本块由若干条语句组成,每个基本块有唯一的基本块号*BlockId*以快速定位基本块中的语句.*assert*用来判断表达式*Exp*是否成立,*assume*用来判断当前语句*Exp*是否可达.在控制流图中,记起始节点为*s*,出口节点为*f*,则程序中的语句可以使用定义的语法进行表示.例如,对于分支语句**if** *Exp* **then** S_1 **else** S_2,可以转换为:

start: skip; goto then, else

then: *assume Exp*; S_1; goto End

else: *assume* $\neg Exp$; S_2; goto End

end

对于语句*S*,其前置条件*P*和后置条件*Q*、最弱前置条件的计算可以通过转换函数*wp*(*S*,*Q*)来获得,在最弱前置条件所遵守的语法中,前置条件的计算为:

wp(*assert P*,*Q*)=$P\wedge Q$

wp(*assume P*,*Q*)=$P\wedge Q$

$wp(S_1;S_2,Q)=wp(S_1,wp(S_2,Q))$

对于分支语句,有$wp(S_1$ or $S_2,Q)=wp(S_1,Q)\wedge wp(S_2,Q)$[17].在实际分析过程中,最弱前置条件通过当前语句及其后置条件经转换函数*wp*计算获得,即*wp*(*Stmt*,*post-condition*)=*precondition*,故后置条件是前置条件计算过程中的重要参数,图1为程序的控制流图以及对应语句的后置条件.

为体现分支语句可能的执行路径,图1(a)中省略了条件语句的判断条件.程序经过静态单赋值(SSA)操作后,每个变量在对其进行定义的语句中,通过增加下标的方式进行改名.这样,每个变量均有唯一的一处定义.在图1(b)中,若语句10是可满足的,即存在约束$S_0=0\wedge S_1=S_0+1\wedge S_2=S_1+1\wedge S_2>8$,其中,每个节点的左侧标记了对应的后置条件$\tau_1:s_0\leqslant 0;\tau_2:s_0\leqslant 0;\tau_5:s_1\leqslant 1;\tau_6:s_1\leqslant 1;\tau_9:s_2\leqslant 2$.在图1(c)中,通过改变节点5的取值,使得路径约束条件为$S_0=0\wedge S_1=S_0+1\wedge S_2=S_1+2$,通过判断该符号状态与后置条件$\tau_9:s_2\leqslant 2$的逻辑包含关系,来判断程序的路径可行性.同理,在图1(d)中,通过改变节点1的取值来进行程序性质的验证.

在符号分析过程中,对循环结构的分析精度会直接影响到执行路径生成的效率,由于符号执行的方法在分

析过程中可能会出现状态空间爆炸的问题,故通常使用抽象的方法来计算循环不变式*I*,使得符号执行具有一个有限的状态空间,并通过循环不变式和循环表达式的析取操作来产生新的循环控制条件.在控制流图中,循环体的出口节点有一条指向循环体入口节点的回边,即存在由循环体中的节点所组成的环结构.

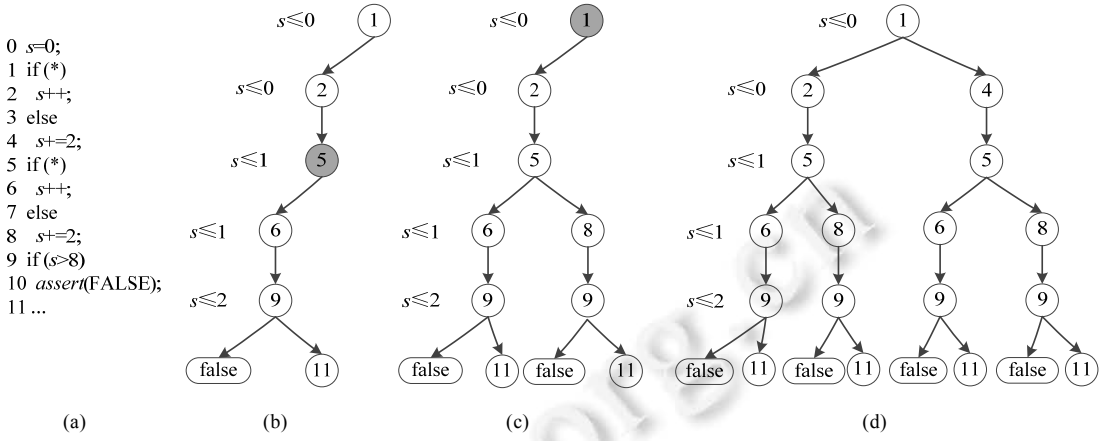

Fig.1 Control flow graph with post-condition

图 1 带后置条件的控制流图

图 2 为对一个带循环的程序进行符号分析以及对循环结构的处理过程.

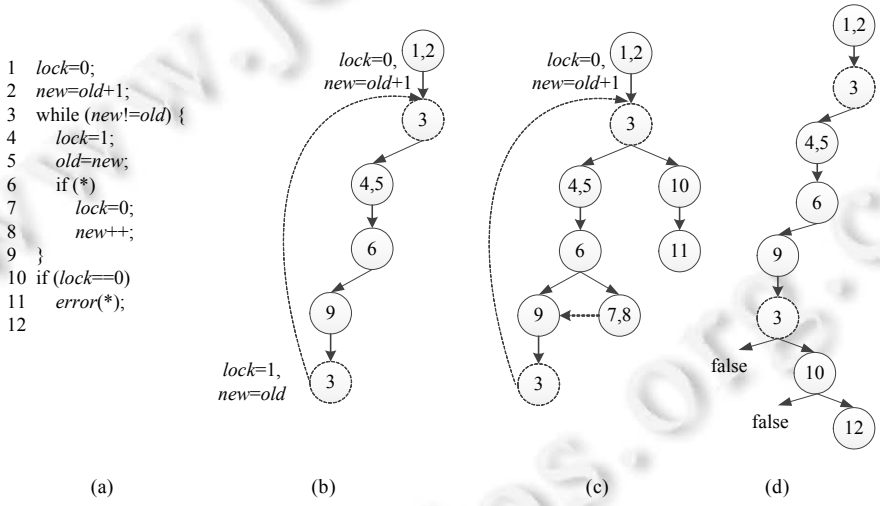

Fig.2 Example of loop structure

图 2 循环结构分析示例

图 2(b)为程序的部分控制流图,其中,语句 1 和语句 2 以及语句 4 和语句 5 均用同一个节点标记.当程序执行图 2(b)所示的执行轨迹时,经静态单赋值操作后,路径条件为

$$lock_0=0\wedge new_0=old_0+1\wedge(new_0!=old_0)\wedge lock_1=1\wedge old_1=new_0.$$

注意到图 2(b)上部的节点 3 处的变量状态为 $lock_0=0,new_0=old_0+1$,当循环体完成第 1 次迭代时,变量状态为 $lock_1=1\wedge old_1=new_0$.此时,可以对变量状态进行抽象操作,即:

$$\Delta(lock_0=0\succ lock_1=1)=\text{true},\Delta(new_0=old_0+1\succ old_1=new_0)=\text{true}.$$

其中,Δ为抽象函数.这样,图 2(b)下部的节点 3 有一条指向上部节点 3 的回边.此时,由于 $old_1=new_0$,故循环条件不

可满足.此时的路径条件为 $lock_0=0\wedge new_0=old_0+1\wedge new_0=old_0$,由于其不可解,故在图 2(d)中在下部的节点 3 处继续对循环体进行迭代分析.

程序中的循环结构分析主要有线性循环分析和非线性循环分析,本文主要针对非线性循环的执行条件进行研究.对于具有 n 维非线性多项式约束条件的循环结构,有如下定理.

定理 1. 对于 n 维空间中的一个集合 A,映射 $X\mapsto F(X),X\in A$.对于循环体中的赋值语句 $X=F(X)$,若条件 $k\|X-F(X)\|\geqslant\|F(X)-F^2(X)\|,0<k<10$ 是可满足的[18],则对于循环结构存在输入 X,使得程序循环条件恒为真的充要条件是 $F(X)$在 A 上存在不动点.

证明:

- 首先对必要性进行证明.

当程序循环判断条件恒为真时,对于循环体存在如下无穷迭代序列:

$$s=\{X_0,X_1,X_2,\ldots,X_n,\ldots\}=\{F(X_0),F^2(X_0),F^3(X_0),\ldots\}\subseteq A.$$

设 $i>j$,

$$\|X_i-X_j\|\leqslant\|X_i-X_{i-1}\|+\|X_{i-1}-X_{i-2}\|+\ldots+\|X_{n+1}-X_n\|\leqslant\sum_{m=j}^{i}k^m\,\|X_0-F(X_0)\|=\frac{k^j-k^{i+1}}{1-k}\,\|X_0-F(X_0)\|\leqslant\frac{k^j}{1-k}\,\|X_0-F(X_0)\|.$$

由于 $\lim_{i,j\to\infty}\|X_i-X_j\|\to 0$,故存在不动点 x,使得 $\lim_{i\to\infty}X_i\to x$. 故:

$$x=\lim_{i\to\infty}X_i=\lim_{i\to\infty}F(X_i)=F(\lim_{i\to\infty}X_i)=F(x).$$

- 证明充分性.

若 $F(X)$在 A 上存在不动点 x,由于 $F^i(x)=x\in A$ 对于任意的 i 均成立,即循环判断条件恒为真.证毕. □

3 近邻路径生成

定义 6(编辑距离(edit distance)). 对于两个字符串 S 和 T,若其中一个字符串经过一系列插入、删除、替换操作可以转换为另一个字符串,其操作次数最少的数值用来定义这两个字符串的编辑距离.实际分析过程中,使用动态规划的方法来对编辑距离进行计算,其动态规划公式为:

- $i==0\ \&\&\ j==0, edit(i,j)=0$;
- $i==0\ \&\&\ j>0, edit(i,j)=j$;
- $i>0\ \&\&\ j==0, edit(i,j)=i$;
- $i>0\ \&\&\ j>0, edit(i,j)=\min(edit(i-1,j)+1, edit(i,j-1)+1, edit(i-1,j-1)+f(i,j))$.

其中,

- 函数 $edit(i,j)$用来计算 S 的子串[0...i]与 T 的子串[0...j]之间的编辑距离;
- 函数 $f(i,j)$用来计算 $S[i]$转换到 $T[j]$所需要的操作次数,若 $S[i]!=T[j]$,则 $f(i,j)=1$;否则,$f(i,j)=0$.

为了对所生成路径的相似程度进行量化分析,通过计算编辑距离的方法分析两条执行路径之间的相似程度,因子 k 用来记录编辑距离的值.对于路径 l 和与其编辑距离不超过 k 的路径集合 L:

$$L=\{\varepsilon\mid\varepsilon\in Paths\wedge\Delta(l,\varepsilon)\leqslant k\},$$

其中,算子Δ用来计算编辑距离.WP 的定义如下:$WP:\{(s,pc,path,k)\in Prog\times PC\times Paths\times k\to PC\}$,其中,$(s,pc,path,k)$为经过后向符号分析后所生成的新的路径条件.通过 SMT 求解器,可以得到路径 l 所对应的输入变量的近似值,其中,s 为目标路径,pc 为与 s 对应的路径条件.当 k 的取值为 0 时,本文方法分析当前路径的最弱前置条件,用来生成目标路径的最弱前置条件,表示为 $WP(Prog,PC,Paths,k)=WP(s,pc,path)$;当对 k 的取值没有限制,即 $k\to\infty$时,由于取消了距离因子的限制,本文的方法即为传统的最弱前置条件分析方法,用来生成从该节点开始进行逆向分析以生成所有节点的最弱前置条件,表示为 $\lim_{k\to\infty}WP(Prog,PC,Paths,k)=WP(Prog,PC)$.一般情况下,当 $k\in(0,\infty)$时,在距离因子 k 的作用下生成与当前路径的距离不超过 k 的路径集合表示为

$$WP(Prog,PC,Paths,k)=\vee_{l\in L}WP(l,\varphi).$$

经过最弱前置条件分析后,可以生成变量的输入域 I^{WP}.但依据该变量所生成的近邻路径集合中可能存在不

可行的路径,即,执行路径不包含控制流图的出口节点.对于程序 $Prog$,其运行过程可以表示为输入集合 I 到输出结合 O 上的映射:$I\to O$.设程序的后置条件为 Q,$Prog$ 在输入 $i(i\in I)$上的执行过程可以表示为 $Prog(i)\models Q$.为了分析输入域与程序执行语义间的依赖问题,我们引入如下定义.

定义 7(无效输入域 $I^{infea}=\{i|i\in I,Prog(i)\models\neg Q\}$). 它表示程序 $Prog$ 在输入域 I 上的所有可行的路径均不能使后置条件 Q 得以满足.对于该无效的输入域,需要从最弱前置条件所生成的输入域中删除.相应地,经过删除操作后所生成的有效输入域 I^{infea} 可以表示为 $I^{fea}=\{i|i\in I^{wp}\backslash I^{infea},Prog(i)\models Q\}$.在距离因子 k 的作用下,经后向符号分析所得的变量的输入域 I^{wp} 与输入集合 I 的关系为 $\lim_{k\to\infty}I^{wp}=I$. $I,I^{wp},I^{infea},I^{fea}$ 之间的关系如图 3 所示,其中,

$$I^{fea}\cup I^{infea}=I^{wp}.$$

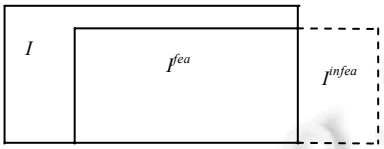

Fig.3 Relationship between the input areas

图 3 几种输入域之间的关系

对于不同的距离因子 k_1 和 k_2,使用后向符号分析所生成的输入域为 I_1 和 I_2,符号执行工具在该输入域上所生成的路径在程序控制流图的出口节点处对应的后置条件为 Q_1 和 Q_2,称 k_1 比 k_2 更优当且仅当如下条件成立:

$$(k_1\succ k_2)\Leftrightarrow(Q_1\to Q_2\wedge I_1\subseteq I_2).$$

在最优距离因子 k 的计算过程中,不同 k 的取值会使得控制流图中每个节点的最弱前置条件可能有所不同,控制流图的规模随着程序规模的扩大而扩大,故在实际分析过程中,往往需要对控制流图进行必要的压缩.本文依据控制流图节点的最弱前置条件对控制流图中的节点进行等价类划分,将具有相同的最弱前置条件的节点合并为压缩的控制流图中的一个节点,同时删除这些节点之间的边.下面给出控制流图中节点的等价关系的定义.

定义 8. 控制流图 G 上的等价关系是节点集合 V 上的二元关系,若节点 u 和 v 是等价的:$u\equiv v$,当且仅当 $P(u)=P(v)$且 $Q(u)=Q(v)$,其中,P 和 Q 分别为节点的前置条件和后置条件.等价关系满足传递性、对称性、自反性,故符号≡描述了节点集合 V 上的等价关系.

如图 4 所示,图 4(a)为原始的符号分析树,图 4(b)和图 4(c)为经压缩操作后的控制流图,具体的压缩规则为:对于控制流图 $G(V,E)$中的节点 u 和 v,其中,$u,v\in E,\langle u,v\rangle\in E$,节点 2 和节点 4 由于有相同的前驱节点(节点 1)和后继节点(节点 6 和节点 7),故节点 2 和节点 4 可以合并为同一个节点,同时删除节点中重复的边.由于在最弱前置条件计算过程中已经对循环结构进行了相应的预处理,故删除从节点 7 到节点 5 的边,如图 4(b)所示.同理,节点 6 和节点 7 具备合并的条件,从而压缩后的控制流图如图 4(c)所示.故 G 中不存在环结构.

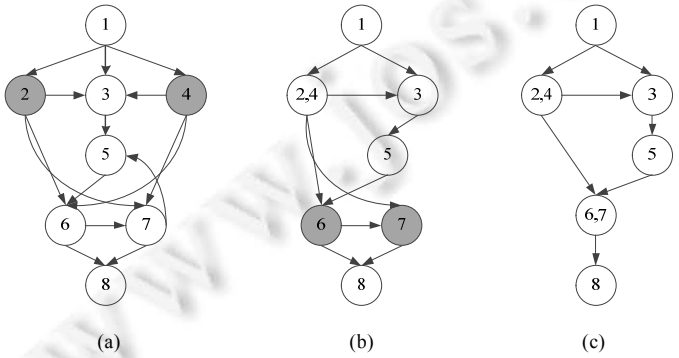

Fig.4 Example of the compression of CFG

图 4 控制流图的压缩示例

对于节点 u,设与其有相同前置条件和后置条件的节点集合为 $U=\{x|x\in V,u\equiv x\}$,即,U 为包含节点 u 的等价类节点.设经过压缩的控制流图为 $G'(V',E')$,在 G'的基础上所生成的近邻路径集合 $S(G')$与在原始的控制流图 G 上生成的近邻路径集合 $S(G)$是等价的,即,$S(G')=S(G)$.下面将对其正确性进行证明.

定理 2. 在 G'中,节点 u,v 的等价类节点分别为 U 与 V,其中,$U\subset U'$,$V\subset V'$,若$\langle u,v\rangle\in E'$成立,当且仅当对于任意的节点 u,v,有$\langle u,v\rangle\in E$ 成立,其中,$\{\langle u,v\rangle|u\in U,v\in V\}$.

证明:该定理的充分性是显然的,下面对必要性进行证明.

对于节点 $u\in U,v\in V$,因 U 中任意两个节点的后置条件是相同的,故对于节点 $u_i\in U$,有$\langle u_i,v\rangle\in E$;因 V 中任意两个节点的前置条件是相同的,故对于节点 $u_i\in U,v_i\in V$,有$\langle u_i,v_i\rangle\in E$,从而有$\langle u,v\rangle\in E$ 成立.证毕. □

定理 3. 在原始的控制流图 G 中,记节点 u 和 v 之间的最短距离为 D,在压缩的控制流图 G'中,节点 u 和 v 的等价类节点分别为 U 和 V,对于 $u_i\in U,v_i\in V$,有 $D(G,u,v)=D(G',u_i,v_i)$.

证明:记节点 u 和 v 之间的路径 l:$u\to n_1\to n_2\to\ldots\to n_{D-1}\to v$,对于路径 l 中的任意两个节点 n_i 和 n_j,记其对应的在 G'中的等价类节点分别为 N_i 和 N_j,都有 $N_i\neq N_j$,则 l':$U\to N_i\to N_{i+1}\to\ldots\to N_{i+(D-1)}\to V$ 是一条 u 和 v 之间长度为 D 的路径,其中,N_{i+1} 为 n_i 的后继节点对应的等价类集合.因为若不成立,则存在节点 N_i 和 N_j 使得 $N_i=N_j$.从而,节点 N_i 和 N_j 有相同的后置条件.故从节点 u 开始,有一条路径 $u\to n_1\to\ldots\to n_i\to n_{j+1}\to\ldots\to n_{D-1}\to v$ 到达 v.故该路径不是节点 u 和 v 之间的最短路径,与假设相矛盾.故在 G'中也存在一条 U 和 V 之间长度为 D 的路径.假设 G'中在节点 U 和 V 中存在一条路径 $U\to K_1\to K_2\to\ldots\to K_k\to V$,其中,$k$ 为该路径的长度,使得 $k<D$,对于$\langle K_1,\ldots,K_k\rangle$中的某一分量$\langle k_1,\ldots,k_k\rangle$,根据定理 2,有$\langle k_i,k_i+1\rangle\in E$,$0<i<k$,从而在 G 中的节点 u 和 v 之间存在一条长度为 k 的路径.这是错误的,故 l'为 G'中节点 u 和 v 之间的最短路径.证毕. □

4 实验分析

通过构建程序的控制流图,并使用后向符号分析方法对当前节点的执行语义和后置条件进行计算,获得对应的最弱前置条件,并逐渐迭代到控制流图的起始节点.在该过程中,距离因子 k 的取值作用于循环结构和分支结构,从而对最弱前置条件的结果产生影响.本节通过实验来讨论本文方法的有效性,我们使用 WALA 作为实验开发平台,通过对平台提供的 T.J.Watson Libraries 工具进行修改,使其具备后向符号分析功能.符号分析工具采用 Java PathFinder,约束求解器使用 CVC3.Java PathFinder 从控制流图的起始节点,在最弱前置条件的作用下生成与出口节点后置条件相关的近邻路径集合;CVC3 在最弱前置条件和 Java PathFinder 分析过程中,能实时计算各循环节点和分支节点处的由路径谓词组成的约束条件,通过计算约束条件的可解性来分析当前路径的可行性.本文的整体分析流程如图 5 所示.

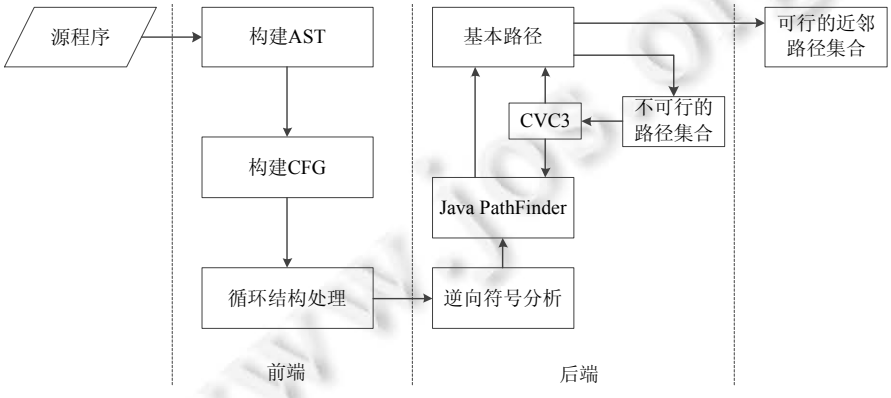

Fig.5 Procedure of the analysis

图 5 整体分析流程

图 5 中,前端(front end)对程序进行预处理,使用 GCC 作为程序的编译器,通过修改配置文件中的预处理选项编译生成中间文件(.I 文件),并将该中间文件作为程序分析工具 T.J.Watson Libraries 的输入.通过对 GCC 中语法扫描器 parse.y 中语义动作的修改,生成程序的抽象语法树(AST),在 AST 的基础上对程序以基本块为单位进行划分,最后构建程序的控制流图.后端(rear end)主要实现对程序中分支结构和循环结构的处理,在距离因子的作用下,通过最弱前置条件的计算来获得程序对应的输入域,并通过符号执行工具和约束求解器来生成基本路径集合,最后,通过判断计算获得的后置条件和设定的后置条件之间的逻辑蕴含关系来判断路径的可行性.本文的实验基于 Intel Core i7 3.5GHz 的处理器,内存大小为 8G,操作系统为 Ubuntu 13.10,内核版本为 3.11.6.测试基准程序(benchmark)使用 DaCapo 和常见的开源 Java 程序,其中,hedc 是一款网络爬虫软件,weblech 是一款下载工具,lusearch 为一款文件索引工具,sunflow 是一款照片处理程序,avrora 是一款仿真工具,hsqldb 为关系数据库的引擎,antlr 是一款扫描器生成工具,batik 是一款浏览器软件.由于其中有大量的分支语句、循环语句,可以较好地满足方法的使用条件.

ESC/Java 是目前路径分析和验证过程中一种最具代表性的方法,它是一种静态 Java 程序分析器,通过对程序的前置条件和后置条件进行计算分析来获得程序的可行路径,本文将其作为进行对比分析的方法.为了比较 ESC/ Java 和本文的方法在近邻路径生成方面的性能与耗费,设计了如下实验.

- 实验 1:循环结构分析

ESC/Java 通过设定循环体执行次数(设为 k 次)作为上界来生成执行路径,通过对控制流图中的循环体结构进行展开并将其转换为非循环结构,通过在循环展开的节点处插桩相应的引导信息,使得对于不同的 k 值有不同的路径编码,满足该条件的路径称为 k 次迭代路径.ESC/Java 需要在运行之前就设置 k 的取值,对于超过 k 的路径,可能会出现遗漏的现象.在 k 的不同取值的情况下,测试基准程序中 k 次迭代路径的数目.表 1 为测试基准程序的基本信息以及对循环结构处理后路径条件数量的对比,其中,圈复杂度(cyclomatic complexity,简称 CC)用来对程序的复杂程度进行衡量,它的值可以用来表示基本路径的条数,在基本路径集合中通过验证程序后置条件来筛选出可行的路径集合.圈复杂度的值越大,则程序的结构也越复杂,其计算方法为 $CC(G)=e-n+2$,其中,e 为 CFG 中边的个数,n 为 CFG 中节点的个数.路径条件数用来记录在经过预处理后程序中循环节点和分支节点的数目,由于加入了最弱前置条件作为路径生成的引导信息,从而能够及时地对不可解的路径谓词进行分析.从表中可以看出,本文方法在路径条件规模方面比 ESC/Java 有明显的约简.在时间开销方面,ESC/Java 和本文的方法都需要对控制流图中节点的入边进行分析,该过程的时间消耗与控制流图中边的数量呈线性关系.由于本文对控制流图使用邻接表的形式存储,各节点的后继节点按顺序存储,故将最弱条件加入到对应的节点信息中的时间开销与最弱前置条件个数成正比.本文方法在分析过程中将待分析的后置条件进行枚举,并建立待分析节点的后置条件与执行路径的对应关系,在路径执行后,可以通过后置条件来检索对应的执行路径,以提高后置节点到执行路径的转化效率.

Table 1 Basic information of the benchmark

表 1 测试基准程序的基本信息

测试基准程序	CFG中节点数	CFG中的边数	CFG中循环节点数	圈复杂度	ESC/Java路径条件数	本文方法路径条件数
hedc	392	567	31	177	238	163
weblech	238	314	27	78	125	52
lusearch	129	425	80	298	68	26
sunflow	451	715	192	266	83	67
avrora	96	159	43	65	53	41
hsqldb	181	491	15	312	241	105
antlr	356	636	39	282	152	89
batik	209	370	17	163	73	25

随着程序规模的扩大,路径条件也相应地增多,约束求解器在求解过程中不可避免地出现效率较低的情况,故有必要设置相应的溢出处理方法以缓解求解性能的降低.由于本文方法将循环结构进行展开,故一条执行路径中可能包含较多的重复路径片段,我们将这些重复的路径片段与路径条件加以映射,相同的路径条件可以采

用编码的操作进行压缩.故本文方法在操作过程中设置了临界点机制,当路径条件出现溢出时,通过记录当前的路径条件和节点位置,同时将路径条件设置为前驱节点的路径条件,对路径条件进行求解、合并,以减少约束求解器运算的开销.图 6 为距离因子 k 与临界点数量的对应关系,对于不同的 k 值,测试程序的临界点数量最多不超过 9 个.经实验分析可知,因子 k 的取值不超过 20,因为若继续增大,可能会出现约束求解器资源耗尽或者路径条件分析时间过长等现象.从图中可以看出:临界点数量随着因子 k 的增长而出现增长幅度放缓的趋势;同时,在临界点数量较少的条件下,可以生成与目标路径的编辑距离最多为 20 的近邻路径集合.故对于复杂的程序,能够以较低的空间消耗来对近邻路径进行分析.

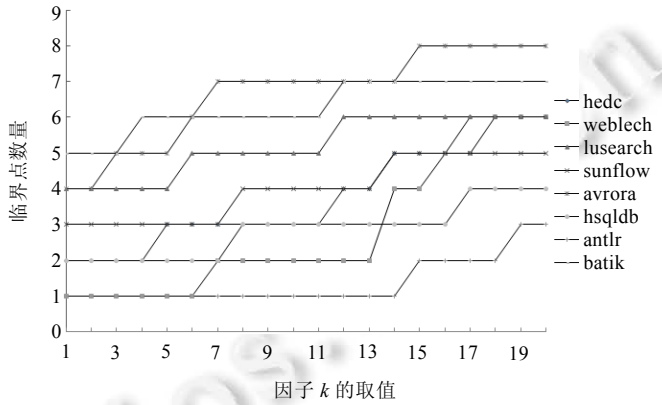

Fig.6 Relationship between the number of critical point and factor k

图 6 临界点数量与因子 k 的关系

- 实验 2:近邻路径生成

本文中,我们引入了加速比(speedup ration)的概念,它表示对于一条待分析的路径,使用 ESC/Java 的方法和本文的方法在生成可行路径集合时的时间开销的比值,图 7 为具体的对比图.从图中可以看出:对于大部分程序,本文的方法在时间开销方面相对于 ESC/Java 有一定程度的提高;但是对于 antlr,出现了时间开销有较大幅度提高的情况,antlr 的加速比约为 1.17.其原因是:该程序中有较多的赋值语句,导致在预处理阶段中,静态单赋值(SSA)在分析过程中给变量在不同定义处加入下标并构建程序的“定义-使用链”,同时,由于大量赋值语句存在于循环结构当中,导致该过程有较大的时间耗费.另外,该程序的循环结构和分支结构的条件语句中存在较多的全局变量,对程序控制流结构有较大的影响.而在 ESC/Java 中,由于没有类似的“定义-使用链”,从而其路径敏感程度要弱于本文的方法.虽然在特殊情况下有一定的优势,但在分析精度方面不及本文的方法.

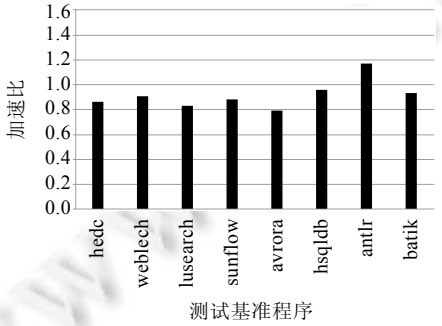

Fig.7 Time consumption comparison

图 7 时间耗费对比

在生成的基本路径中,由于有约束求解器的使用,可以将不可解的路径条件进行删除.但受制于约束求解器的求解能力,符号执行工具仍然会生成一部分不可行的路径.图 8 显示了 ESC/Java 和本文的方法在生成可行路径方面的精度对比,从图中可以看出:本文方法由于有后向符号分析所生成的最弱前置条件作为符号执行工具生成近邻路径的引导信息,可以有针对性地生成基本路径集合,在此基础上,通过分析程序在控制流图出口节点处变量的取值,并分析与设定的后置条件的逻辑包含关系来判断该路径的可满足行.测试集中,sunflow 的可行相似路径百分比提升的幅度最大,其值为 21.9%,测试集的平均提升幅度为 14.1%.

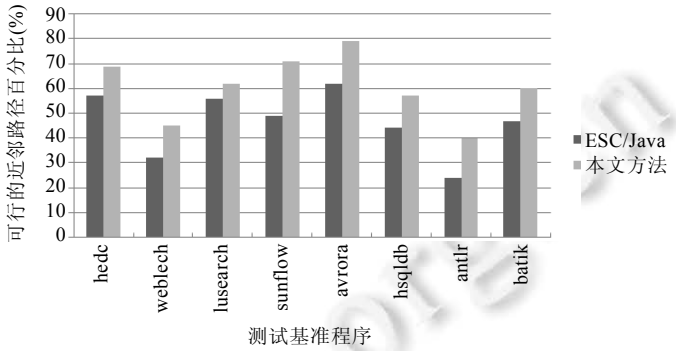

Fig.8 Comparison of the analysis precision

图 8 分析精度对比

5 总结与展望

近邻路径生成是程序分析和验证中的主要问题,现有的分析方法直接使用符号执行等方法生成执行路径,但是这种粗粒度的方法由于缺乏必要的路径引导信息而生成较多的冗余路径和无效路径.本文提出一种后向符号分析方法,通过对程序控制流图中循环节点的处理,并使用静态单赋值的方法进行数据流敏感的分析,在此基础上,从控制流图的出口节点开始,使用后置条件和节点的执行语义计算其最弱前置条件,逐步迭代到起始节点,从而获得程序变量的输入域.在此基础上,使用符号执行和约束求解器,在最弱前置条件信息的引导下生成与目标后置条件近邻的路径集合.实验结果表明:本文的方法可以有效减少路径条件的数量,并使用较少的临界点对程序的多次迭代进行有效的分析;同时,对于大部分程序,本文的方法比 ESC/Java 方法在时间开销方面有一定优势,但是对于有较多赋值语句的程序,由于静态单赋值分析导致在构建“定义-使用链”时有较多的时间耗费,这也是将来工作中需要改进的内容之一.在分析精度方面,由于加入了路径引导信息,本文方法生成的可行路径百分比相对于 ESC/Java 方法均有较大幅度的提高.

将来的主要工作有:结合目前恶意程序在执行过程中采用代码加壳、路径变换等混淆手段对分析工具进行干扰,故我们可以在本文方法的基础上通过对程序行为轮廓的构建,并使用控制流图比较的方法来对程序的行为特征进行分析,使本文的方法具有更好的适应性.

致谢 我们向为本文提出宝贵修改意见的美国佐治亚理工学院计算机科学系的 Alessandro Orso 教授和审稿专家表示衷心的感谢.

References:

[1] Wang R, Feng DG, Yang Y, Su PR. Semantics-Based malware behavior signature extraction and detection method. Ruan Jian Xue Bao/Journal of Software, 2012,23(2):378−393 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/3953.htm [doi: 10.3724/SP.J.1001.2012.03953]

[2] Wang Z, Pierce K, McFarling S. BMAT—A binary matching tool for stale profile propagation. Journal of Instruction-Level Parallelism, 2000,2(1):23−43.

[3] Bayer U, Comparetti PM, Hlauscheck C, Kruegel C, Kirda E. Scalable, behavior-based malware clustering. In: Proc. of the Network and Distributed System Security Symp. (NDSS 2009). San Diego: NDSS Association, 2009. 8−11. http://www.isoc.org/isoc/conferences/ndss/09/slides/11.pdf

[4] King J. Symbolic execution and program testing. Communications of the ACM, 1976,19(7):385−394. [doi: 10.1145/360248.360252]

[5] Dijstra E. A Discipline of Programming, Vol.1. Englewood Cliffs: Prentice Hall, 1976. 12−25.

[6] Nipkow T, Paulson L. Isabelle/HOL: A proof assistant for higher-order logic. LNCS, 2002,2283:120−131. http://www21.in.tum.de/~nipkow/LNCS2283/

[7] Clarke M, Grumberg O, Peled D. Model Checking. 3rd ed., Cambridge: The MIT Press, 1999. 9−15.

[8] Cruz J. Constraint Reasoning for Differential Models. 5th ed., Amsterdam: The IOS Press, 2005. 63−77.

[9] Rabek JC, Khazan RI, Lewandowski SM, Cunningham RK. Detection of injected, dynamically generated, and obfuscated malicious code. In: Proc. of the 2003 ACM Workshop on Rapid Malcode. New York: Association for Computing Machinery, 2003. 76−82. [doi: 10.1145/948187.948201]

[10] Flake H. Structural comparison of executable objects. In: Proc. of the Int'l Conf. on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA 2004). Dortmund: Association for Computing Machinery, 2004. 83−97. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.83.6632

[11] Gao DB, Reiter MK, Song D. Binhunt: Automatically finding semantic differences in binary programs. In: Proc. of the Int'l Conf. on Information and Communications Security. Berlin, Heidelberg: Springer-Verlag, 2008. 238−255. [doi: 10.1007/978-3-540-88625-9_16]

[12] Bailey M, Oberheide J, Andersen J, Mao ZM, Jahanian F, Nazario J. Automated classification and analysis of Internet malware. In: Kruegel C, Lippmann R, Clark A, eds. Proc. of the 10th Int'l Conf. on Recent Advances in Intrusion Detection. Berlin, Heidelberg: Springer-Verlag, 2007. 178−197. [doi: 10.1007/978-3-540-74320-0_10]

[13] Manevich R, Sridharan M, Adams S. PSE: Explaining program failures via postmortem static analysis. In: Proc. of the 7th Fundamental Approaches to Software Engineering (FASE 2004). Barcelona: Association for Computing Machinery, 2004. 63−72. [doi: 10.1145/1029894.1029907]

[14] Flanagan C, Leino K, Lillibridge M, Nelson G, Saxe JB, Stata R. Extended static checking for Java. In: Proc. of the 23rd Int'l Conf. on Programming Language Design and Implementation (PLDI 2002). New York: Association for Computing Machinery, 2002. 234−245. [doi: 10.1145/512529.512558]

[15] Csallner C, Smaragdakis Y, Xie T. Dsd-Crasher: A hybrid analysis tool for bug finding. ACM Trans. on Software Engineering and Methodology, 2008,17(2):1−37. [doi: 10.1145/1348250.1348254]

[16] Gu ZX, Barr ET, Hamilton DJ, Su ZD. Has the bug really been fixed. In: Proc. of the 32nd Int'l Conf. on Software Engineering (ICSE 2010). Cape Town: IEEE Computer Society, 2010. 55−64. [doi: 10.1145/1806799.1806812]

[17] Jaffar J, Murali V, Jorge A, Andrew E. TRACER: A symbolic execution tool for verification. In: Proc. of the 24th Int'l Conf. on Computer Aided Verification (CAV 2012). Berkeley: Springer-Verlag, 2012. 758−766. [doi: 10.1007/978-3-642-31424-7_61]

[18] Li Y. Termination analysis of nonlinear loops. Ruan Jian Xue Bao/Journal of Software, 2012,23(5):1045−1052 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/3982.htm [doi: 10.3724/SP.J.1001.2012.03982]

附中文参考文献:

[1] 王蕊,冯登国,杨轶,苏璞睿.基于语义的恶意代码行为特征提取及检测方法.软件学报,2012,23(2):378−393. http://www.jos.org.cn/1000-9825/3953.htm [doi: 10.3724/SP.J.1001.2012.03953]

[18] 李轶.非线性循环的终止性分析.软件学报,2012,23(5):1045−1052. http://www.jos.org.cn/1000-9825/3982.htm [doi: 10.3724/SP.J.1001.2012.03982]

郭曦(1983－),男,湖北鄂州人,博士,讲师,主要研究领域为信息安全,软件分析,软件测试.

王盼(1987－),女,博士生,讲师,主要研究领域为电力电子功率变换.