软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2016,27(5):1230−1245 [doi: 10.13328/j.cnki.jos.005027]

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

面向危险操作的动态符号执行方法*

王伟光[1,2], 曾庆凯[1,2], 孙 浩[1,2]

[1](计算机软件新技术国家重点实验室(南京大学),江苏 南京 210046)

[2](南京大学 计算机科学与技术系,江苏 南京 210046)

通讯作者: 曾庆凯, E-mail: zqk@nju.edu.cn

摘 要: 针对缺陷检测的需求,提出了面向危险操作的动态符号执行方法.依据所关注的缺陷类型,定义危险操作及危险操作相关路径,通过计算覆盖不同上下文中危险操作的能力,协助动态符号执行选择高效初始输入,并利用危险操作相关信息引导测试流程.缺陷检测成为定位待测程序内危险操作以及对危险操作相关路径进行检测的过程.实现了面向 Linux 平台二进制可执行程序的原型系统 CrashFinder,实验结果表明,该方法能够更快地发现更多缺陷.

关键词: 危险操作;动态符号执行;污点分析;路径选择;缺陷检测

中图法分类号: TP311

中文引用格式: 王伟光,曾庆凯,孙浩.面向危险操作的动态符号执行方法.软件学报,2016,27(5):1230−1245. http://www.jos.org.cn/1000-9825/5927.htm

英文引用格式: Wang WG, Zeng QK, Sun H. Dynamic symbolic execution method oriented to critical operation. Ruan Jian Xue Bao/Journal of Software, 2016,27(5):1230−1245 (in Chinese). http://www.jos.org.cn/1000-9825/5027.htm

Dynamic Symbolic Execution Method Oriented to Critical Operation

WANG Wei-Guang[1,2], ZENG Qing-Kai[1,2], SUN Hao[1,2]

[1](State Key Laboratory for Novel Software Technology (Nanjing University), Nanjing 210046, China)

[2](Department of Computer Science and Technology, Nanjing University, Nanjing 210046, China)

Abstract: Addressing the requirement for defect detection, this paper proposes critical operation oriented dynamic symbolic execution. First, based on the defined critical operations and the relevant critical paths, a set of initial inputs are evaluated by computing the ability of covering critical operations under different contexts, and efficient initial inputs can be selected for the following dynamic symbolic execution. Second, leveraging the critical operations, dynamic symbolic execution is guided to explore paths which are more prone to defects. In this way, defect detection becomes a process of locating critical operations and exploring critical paths. A prototype system called CrashFinder is implemented and tested on a number of Linux x86 executables. The experimental results show that this approach is effective in initial input evaluation and efficient in defect detection.

Key words: critical operation; dynamic symbolic execution; taint analysis; path selection; defect detection

动态符号执行(dynamic symbolic execution)[1,2]是一种将具体执行与符号执行相结合的自动化测试技术,其目标是通过自动生成测试用例来尽量覆盖程序中的可行路径,进而发现程序内的缺陷.动态符号执行方法首先选择一个初始输入,以该输入同时具体执行与符号执行待测程序;执行过程以插装方式收集计算路径分支条件的符号表达式(路径约束),然后按照指定策略对收集的路径约束进行翻转(flip),并借助约束求解器求解生成新

* 基金项目: 国家自然科学基金(61170070, 61572248, 61431008, 61321491); 国家科技支撑计划(2012BAK26B01)

Foundation item: National Natural Science Foundation of China (61170070, 61572248, 61431008, 61321491); National Key Technology R&D Program of China (2012BAK26B01)

收稿时间: 2015-07-29; 修改时间: 2015-10-08, 2015-12-22; 采用时间: 2016-01-05; jos 在线出版时间: 2016-01-16

CNKI 网络优先出版: 2016-01-18 13:50:58, http://www.cnki.net/kcms/detail/11.2560.TP.20160118.1350.004.html

测试用例;在对新用例进行缺陷检测后,动态符号执行方法按照指定调度策略从中选取合适的用例作为下一轮的输入.重复上述路径探索流程,直至达到预设标准或资源耗尽.在路径约束收集过程中,当遇到不能符号化的数据类型或操作时,动态符号执行使用具体值来简化其符号表达式.动态符号执行方法在一定程度上解决了传统测试方法缺乏测试用例集的问题,并在缺陷检测方面表现出良好的效果[1−12].但由于缺乏对缺陷特征的考虑,现有动态符号执行技术存在以下问题:

- 首先,通过执行初始输入所收集的路径约束是动态符号执行工作流程的基础.研究工作[3]采用不同初始输入运行工具 SAGE 对目标程序进行动态符号执行测试,发现不同初始输入对测试的缺陷检测效果产生极大的影响.文献[3]注意到初始输入的选择对动态符号执行方法的重要性,但并没有进一步研究其原因或提供相应选择策略.而且,现有动态符号执行工具大多随机选择初始输入[1−12].因此,为动态符号执行选择高效的初始输入有利于提高其缺陷检测效果.
- 其次,动态符号执行方法通过调度测试用例来指导对待测程序状态空间的探测.在现实测试任务中,程序的路径数量呈指数增长,而测试时间有限.面对路径爆炸问题,现有动态符号执行相关工作[3−12]通常采取的措施为:在测试用例调度阶段,评估备选测试用例在覆盖新基本块方面的能力,选择其中对基本块覆盖率贡献最大的用例优先进行调度,从而在有限的测试资源下达到覆盖更多基本块的目的.上述方法虽然能够获得更好的基本块覆盖率,但近期研究工作[3,13]表明,缺陷的触发与基本块的覆盖之间并没有直接联系.因此,单纯地提高测试过程中的基本块覆盖速度,并不一定能加速缺陷的触发.现有动态符号执行技术的测试用例调度方法缺乏针对缺陷检测的考量.

为了提高动态符号执行方法的缺陷检测效果和效率,提出了面向危险操作的动态符号执行方法.以分析程序内与缺陷发生密切相关的关键操作和不同路径对触发缺陷的贡献为基础,依据对易触发缺陷路径的覆盖能力来选择高效初始输入;按照路径对触发缺陷的贡献大小来生成和调度测试用例,引导测试过程对关键路径进行优先系统遍历,以达到加速触发缺陷的目的.

1 相关工作

近年来,随着约束求解技术的进步,国内外学者提出了许多符号执行相关的方法和工具.根据执行流程的差异,这些方法和工具大致可分为动态符号执行[2]和 Execution-generated Testing(EGT)两类[4].其中,EGT 技术以 EXE[14],KLEE[15]等工具为代表.不同于动态符号执行,此类方法在待测程序入口点引入一个数据结构(称之为符号执行存储块:SESB)存储当前程序计数器(PC)、已收集的路径约束以及进程执行上下文等状态.当程序执行到分支跳转节点时,方法创建一个新的 SESB,将当前条件语句真假分支对应的符号表达式添加进不同 SESB 的路径约束,并更新对应的 PC 值以及进程上下文.如果更新后的路径约束可解,则将对应 SESB 加入到备选集合中,然后根据一定的搜索策略从 SESB 集合中选择其一来继续对待测程序进行探测.动态符号执行与 EGT 方法在运行时所占空间与时间上各具优缺点[4,15].在本节中,我们主要讨论与动态符号执行相关的测试方法和工具.

Godefroid 等人[1]将传统符号执行与具体执行相结合,并借助约束求解器自动生成测试用例,实现了针对 C 语言程序的原型工具 DART.SAGE[3]是由微软开发的二进制级别动态符号执行工具,它追踪程序轨迹并记录执行日志,符号化解释日志中的 x86 指令序列记录约束条件,并通过求解器生成指导跳转方向的测试用例.Smartfuzz[7]与 Avalanche[9]分别引入了整型溢出错误与内存访问错误的缺陷模型.在收集路径约束的同时,插入相关缺陷的触发条件,利用约束求解器检查相应的缺陷是否可以触发,进而检测 Linux 平台可执行程序中的整型溢出缺陷和段错误.以上这些动态符号执行工作在缺陷检测方面都表现出良好效果,然而对于初始输入,这些工具均是简单的随机选取.由于不同初始输入在基本块覆盖以及缺陷检测上的表现不尽相同,随机选择初始输入往往会造成时间与计算资源的浪费,这个问题在具有成本和资源限制的实际测试任务中尤为突出.

动态符号执行方法通过生成并执行可覆盖不同路径的新测试用例来探测待测程序状态空间.如何调度备选测试用例对待测程序进行高效的探测,是相关工作研究的重点.(1) 在二进制级别,SAGE[3],Smartfuzz[7]以及 CrashMaker[8]等现有的动态符号执行工作大多依据备选测试用例可覆盖的新增基本块的数量来评定它们的优

先级,通过调度优先级高的用例来尽快覆盖更多的基本块.由于缺乏缺陷相关知识的指导,这种调度手段会导致在触发缺陷之前生成大量无用测试用例,从缺陷发现角度考虑则是浪费了时间和计算资源.IntHunter[5]计算新测试用例可覆盖的新基本块数量,同时动态监控当前执行是否存在易发生整型溢出的高危操作:如果是,则优先调度这些用例.该工具结合基本块覆盖率贡献与缺陷相关知识对测试用例进行调度,但方法只关注当前执行路径中的高危操作,对易发生缺陷的其他相关路径缺乏主动定位和探索.(2) 在源代码级别,zesti[16]定位执行路径中与关注缺陷相关的关键操作,计算路径约束项与关键操作之间的距离,并据此进行测试用例的生成和调度.不同于其他动态符号执行工作,zesti 以增强现有回归测试用例集合为目的,只对局部代码区域进行针对性检测.CREST[17]提出了基于程序控制流图的路径搜索策略,根据节点与尚未覆盖的分支之间的距离作为标准来调度测试用例.Target[18]利用静态分析定位 C 语言程序中可能发生缓冲区溢出缺陷的代码块位置,然后分析程序控制流图来指导动态符号执行有针对性地产生测试用例.Su 等人[19]结合符号执行与模型检测,对待测程序进行高效的数据流覆盖测试.方法利用静态分析识别待测程序控制流图中每个变量定义-使用组合(def-use pair)的关键分割节点(cut points),用来指导动态符号执行的路径选择.上述源代码级别的方法均基于程序的控制流图以及静态分析,但是在多数商业软件不提供源代码的情形下,二进制文件由于指针别名、间接跳转、代码量大等情况的影响,会对控制流程图的生成以及静态分析方法带来困难[20].

基于以上分析,我们提出一种面向危险操作的二进制级别动态符号执行方法.软件中被用户控制、容易引发缺陷的安全敏感操作被称为危险操作.研究人员发现,软件缺陷多数与危险操作相关.因此,以危险操作为出发点来提高测试技术的效率是相关工作研究的热点之一[21−23].其中,Buzzfuzz[21]提出了导向型模糊测试的概念,利用细粒度污点分析识别输入中可以影响危险操作的关键字节;然后,针对关键字节进行模糊测试,可以有效减少测试用例的数量,提高模糊测试的效率.Taintscope[22]结合动态符号执行与导向型模糊测试,利用动态符号执行识别并求解以绕过 checksum 的检查,提高了后续模糊测试和动态符号执行针对危险操作所生成的测试用例的质量.Dowser[23]针对缓冲器溢出缺陷,结合静态分析和细粒度污点分析,定位循环体内部的数组读写操作和可以影响这些危险操作的关键字节;通过多次动态符号执行学习评估各路径分支对覆盖新数组读写操作的贡献,然后,利用该贡献值选择路径并尝试触发缺陷.然而,在初始输入选择方面,Buzzfuzz 与 Taintscope 均随机选择待测程序合法格式文件做为初始输入,Dowser 则需要人工构造可覆盖指定循环体的输入.在对待测程序状态空间的探索方面,Taintscope 依据关键字节将初始输入部分符号化,然后进行测试用例生成和调度;Dowser 同样以关键字节为基础,计算不同路径分支对覆盖新危险操作的贡献值来进行路径选择.上述工作并没有考虑初始输入相关路径之外可能存在的未被执行的危险操作,在识别可影响危险操作的关键字节时易产生遗漏,从而导致漏报.为此,基于危险操作与缺陷密切相关的思想,我们提出了面向危险操作的动态符号执行方法.以缺陷相关知识为指导,利用细粒度的污点分析识别与缺陷的发生相关的危险操作以及与危险操作相关的路径.通过计算备选初始输入对不同上下文中危险操作的覆盖能力,来协助动态符号执行选择高效的初始输入.并将这些信息用于测试用例的生成和调度,对与已知危险操作相关的路径进行优先遍历;遍历完成后,对未被覆盖的程序状态空间进行快速探索,定位未被检测的危险操作和相关路径,从而对程序中所有危险操作以及其相关的路径进行优先系统的探测,加速缺陷的触发.

2 面向危险操作的动态符号执行方法

2.1 相关知识

2.1.1 程序模型

基于程序控制流图(CFG),我们定义了以下六元组来描述待测程序,用于介绍面向危险操作的动态符号执行方法的相关概念与算法.

定义 1(程序模型). 待测程序可形式化表示为六元组:$P=\langle X,X_0,L,l_0,OP,E\rangle$.

- X:程序变量有限集合,$X_0\subseteq X$ 为程序输入变量集合.
- L:程序位置有限集合,$l_0\in L$,为程序的入口节点.

- *OP*:在变量 *X* 上的操作集合,表示程序执行动作,每个操作 $op\in OP$ 均对应一个程序位置 $l\in L$,函数 $op(l)$ 表示程序位置 l 对应的操作.*OP* 由赋值语句、条件语句、系统函数调用语句以及终止语句组成:
  - 赋值语句:$x:=e$;其中,$x\in X$,e 为变量 *X* 上的算术或逻辑计算表达式;
  - 条件语句:if (c) then l' else l'';其中,c 为变量 *X* 上的逻辑操作表达式,$l',l''\in L$ 为程序位置;
  - 系统函数调用语句:call *fun*;其中,*fun* 为系统库函数,如内存分配函数 *malloc* 等;
  - 终止语句:包含正常终止 halt 与异常终止 abort.
- *E*:程序控制流集合.$E\subseteq L\times L$,对于任意程序位置 $l\in L$,如果 $op(l)$为赋值语句,l'为 l 后继语句位置,记为 $l'=N(l)$,有$(l,l')\in E$;如果 $op(l)$为条件语句:if (c) then l' else l'',则有$(l,l')\in E\wedge(l,l'')\in E$.

2.1.2 动态符号执行

动态符号执行同时具体执行与符号执行待测程序,收集分支条件并且计算路径约束,为约束求解器提供输入来生成新测试用例.动态符号执行过程中,每个程序状态可形式化表示为四元组:$\langle l,C,S,pc\rangle$,其中,

- l 表示当前执行指令的位置.
- C 为程序执行过程中的具体状态空间,对于$\forall x\in X$,$C(x)$代表变量 x 在具体运行时的值;对于表达式 e,$C(e)$代表将表达式内变量$\forall x\in e$ 替换为对应具体值 $C(x)$并求解后得到的值;映射 $C[x\rightarrow v]$代表将 x 在具体状态空间 C 的值更新为具体值 v.
- S 为程序符号状态空间,类似地,$\forall x\in X$,$S(x)$代表变量 x 在执行过程中对应的符号变量值;$S(e)$代表表达式 e 内变量$\forall x\in e$ 替换为对应符号值 $S(x)$后得到的值;映射 $S[x\rightarrow sv]$代表将变量 x 在符号状态空间 S 的值更新为符号值 sv.
- pc 为当前路径的路径约束,即在程序执行中收集的分支条件符号表达式的组合.

动态符号执行方法在具体执行待测程序的同时,根据当前操作类型进行动态的符号化污点传播计算,计算和更新程序状态元组$\langle l,C,S,pc\rangle$.首先,在程序初始状态,方法对状态元组内的变量进行初始化.假设待测程序 P 的初始输入 X_0 包含 n 个字节块:$X_0=\{Input_1,...,Input_{n-1},Input_n\}$,则元组内具体空间 C 对应初始状态 C_0 为初始输入 X_0,即 $C_0=\{Input_1,...,Input_{n-1},Input_n\}$;符号空间 S 的初始状态 S_0,则逐字节地符号化 X_0 内元素 $Input_i$ 为初始符号污点变量θ_i,即 $S(Input_i)=\theta_i(1\leqslant i\leqslant n)$,于是有 $S_0=\{\theta_1,...,\theta_{n-1},\theta_n\}$.在动态符号执行过程中,如果当前程序位置 l 对应的操作 $op(l)$为赋值语句 $x:=e$,则更新当前指令位置为 $l'=N(l)$,同时更新空间 C 与空间 S 内变量 x 新的映射值分别为 $C(e)$和 $S(e)$.如果 $op(l)$为条件语句(if (x) then l' else l''),则依据 $C(x)$的取值判断程序迁移方向,更新相应程序位置 l 以及添加 x 对应的符号表达式到当前路径约束条件 pc.如果 $op(l)$为终止语句 halt 或 abort,则上述流程终止.符号空间 S 内,记映射 Use 表示变量或表达式内包含的初始符号污点变量θ_i的集合.

2.2 基本原理

通过对近年来缺陷案例与缺陷检测研究工作[5,24−26]的调查研究发现,对软件安全产生危害的缺陷,通常是由于对关键操作的相关参数的检查不够完善导致安全性检查语句被旁路,或对关键操作的相关检查语句出现程序员意料之外的组合,最终在执行与软件安全性直接相关的操作时发生错误.我们将操作语句集合中容易发生安全性相关缺陷的关键操作称为危险操作.

定义 2(危险操作). 在目标程序中,既能被用户控制又与软件安全性直接相关的安全敏感操作具有更大可能引发缺陷,这些操作统称为危险操作.

典型的安全敏感操作包括内存分配函数(如 *malloc*,*relloc* 等)、字符串处理函数(如 *strcpy*)等.相关研究[22]表明:当这些操作的参数可被输入影响时,该操作容易被攻击者控制、利用,可造成程序崩溃或控制流被劫持,威胁软件安全性.根据相关研究结果和具体测试目的,可以确定哪些 API 或操作为安全敏感操作.

本文用 $T_{op}(T_{op}\subseteq OP)$表示容易发生测试目标缺陷的安全敏感操作集合.危险操作为集合 T_{op} 内可以被输入数据影响的操作,本文将待测程序中危险操作的集合记做 D_{op},有 $D_{op}\subseteq T_{op}$.

利用选定的初始输入同时具体执行和符号执行待测程序所收集到的路径约束,是动态符号执行方法对待测程序进行探测的基础.不同初始输入往往导致不同测试结果,除了线程竞争等不确定性因素之外,主要是由于

它们所产生的路径约束的差异.由于危险操作与缺陷的发生密切相关,因此我们利用危险操作相关的信息对备选初始输入产生的路径约束进行分析评价[25].路径约束中,程序对危险操作相关参数进行检查和判断的条件语句所对应的路径约束项称为危险操作相关约束项.

定义 3(危险操作相关约束项). 以任意备选初始输入执行待测程序 P 所收集的路径约束 pc,识别的危险操作集合 D_{op}.对于任意路径约束项$\forall pc_i \in pc$,如果存在某个危险操作$\exists dop \in D_{op}$,满足 pc_i 与 dop 可被用户通过相同输入字节控制,则 pc_i 为与危险操作 dop 相关的路径约束项.

具体来说,我们通过识别、收集和计算路径约束内危险操作相关约束项的比重来对备选输入进行评估和选择.路径约束中危险操作相关约束项越多,安全检查性质的条件被旁路以及意外的组合出现的概率越大,程序发生缺陷的可能性也更大[25].本文将包含不同危险操作相关约束项组合的路径约束对应的程序执行路径称为危险操作相关路径.

类似地,在动态符号执行方法对待测程序进行探测时,依据路径约束项与危险操作之间的关系生成和调度测试用例,指导测试流程对待测程序内与危险操作相关的路径进行优先遍历,从而提前检验危险操作的安全性检查是否存在被旁路的可能性.这种调度方法以缺陷发生与危险操作之间关系为出发点,相对于传统基于新增基本块的测试用例调度方法,能够更快地触发缺陷.

图 1 为面向危险操作的动态符号执行方法的结构流程图,包含初始输入选择过程(initial input selecting process)和动态符号执行过程(dynamic symbolic execution process)两部分.

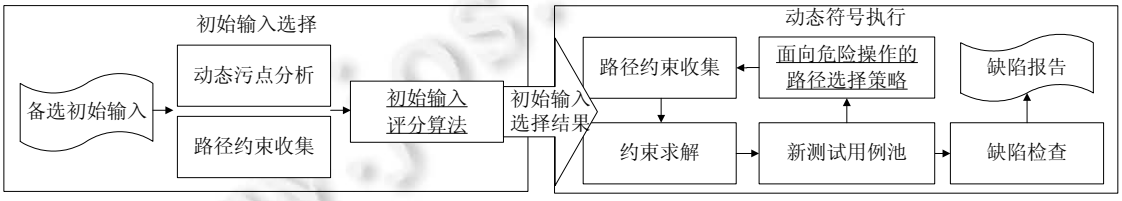

Fig.1 Flowchart of dynamic symbolic execution oriented to critical operation

图 1 面向危险操作的动态符号执行方法流程

初始输入选择过程对备选初始输入进行评估和选择,为动态符号执行过程提供高效的初始输入.初始输入选择过程包括 3 个步骤:首先,利用细粒度动态污点分析监控待测程序执行备选输入,识别程序内危险操作;然后,通过同时具体执行和符号执行备选初始输入完成路径约束的收集,并识别其中的危险操作相关约束项;最后,利用初始输入评分算法,依据被识别的危险操作和危险操作相关约束项,计算路径约束内危险操作相关约束项的比重对不同备选初始输入进行评分,其中,得分高者将被优先选择.

选定合适初始输入后,动态符号执行过程收集路径约束,并利用约束求解器生成新测试用例,对待测程序进行缺陷检测.过程集成面向危险操作的路径选择策略,该策略重复进行危险路径遍历与危险路径定位两个步骤:前者对已知危险操作相关路径进行遍历;后者对待测程序未被覆盖状态空间进行快速探索,定位未被执行的危险操作和相关路径.两者的结合,使测试流程可以对待测程序内所有危险操作和相关的路径进行优先探测.

2.3 初始输入选择

2.3.1 危险操作与危险操作相关约束项的识别

为了使用危险操作相关信息指导初始输入的选择和引导动态符号执行对待测程序的探测,首先需要识别程序内的危险操作.由于静态分析面对二进制可执行程序本身的局限性(如指针别名、间接跳转等不确定问题);同时,传统动态符号执行内符号化污点传播计算只能反映变量间数据依赖关系[5],忽略了用户可通过控制流影响危险操作的情形,因此,我们利用部分控制流敏感的细粒度污点分析来提高危险操作以及危险操作相关约束项的识别精度.

具体地,我们将由用户控制的输入文件作为污染源,将与目标缺陷相关的安全敏感操作集合 T_{op} 作为污点分

析的目的节点.针对污染源,监控文件处理相关系统调用(如 *open*,*close* 等).当文件被打开后,根据文件描述符读取文件内容缓冲区,对其中每个字节分配编号(与上文符号空间内初始符号污点变量 S_0 的设置粒度相同);如果调用 *close* 函数结束文件处理,则结束对相应污染字节的传播分析.

污点传播过程中,针对数据流依赖的污点传播情形,根据不同的指令类型,制定相应的污点传播规则.例如,针对数据赋值指令(如 mov(*var*,*var*1)),则根据源数据污点标记更新目的变量的污点信息:$T(var')\leftarrow T(var1)$,其中,*var*′表示更新后的 *var* 变量,$T(var')$表示可对操作数 *var*′造成影响的初始污点数据字节集合.赋值指令之外,对于一元运算指令(如 not(*var*)等),则制定规则为 $T(var')\leftarrow T(var)$.对于二元运算指令(如 add(*var*1,*var*2)等),有 $T(var1')\leftarrow(T(var1)\cup T(var2))$.针对控制流依赖的污点传播情形,我们借鉴文献[27]的方法,只对程序内具有严格控制流依赖的语句进行污点状态的更新和传递.判断条件语句内变量与条件体内部变量之间是否存在严格控制依赖关系,只需考察条件语句是否为污点变量与常量之间的相等关系判断.以图 2 所示的代码块为例,*key* 为污点数据,当 *key* 等于 99 时,*out* 为 1.即当 *out* 的值为 1 时,用户可以推测 *key* 的具体值,因此,变量 *key* 与 *out* 间存在依赖关系.分析第 2 行判断语句:(*key*==99),其中,关系运算符号为相等判断:"==",且 *key* 为污点变量,属于严格控制依赖,于是有 $T(out)\leftarrow T(key)$.如果第 2 行判断语句为(*key*<99)时用户并不能通过 *out* 为 1 来推测 *key* 的值,则不进行污点传播.

```
1. out=0;
2. if (key==99) {...
3. out=1;}
4. send(out);
```

Fig.2 Example code of strict control dependence

图 2 严格控制流依赖示例代码

污点分析过程中,分析引擎在计算污点数据传播的同时,依据当前关注的缺陷类型设置和维护易发生关注缺陷的安全敏感操作集合 T_{op},用来识别危险操作.具体地,假设当前操作为 *op*,分析引擎首先检查 *op* 是否属于集合 T_{op};然后分析 *op* 的相关参数或操作数是否为污染数据,即判断该操作是否可以被输入数据影响.如果均满足,则依据定义 2 判定 *op* 为危险操作.我们定义危险操作与可对危险操作造成影响的输入字节集合之间的映射 *DByte*,*DByte*(*op*)表示可对危险操作 *op* 的相关参数或操作数造成影响的输入字节的集合.

判断路径约束项是否为危险操作相关约束项,需要分析可影响路径约束项的输入字节是否可以同时影响危险操作.假设以任意输入 $X_0=\{Input_1,...,Input_{n-1},Input_n\}$同时具体执行和符号执行待测程序,所收集到的路径约束为 *pc*,*pc* 由 *m* 个路径约束项组成:$pc=\{pc_1,...,pc_{m-1},pc_m\}$,其中,$m>0$.对于任意危险操作$\forall dop\in D_{op}$,初始输入 X_0 中可对危险操作 *dop* 造成影响的字节集合为 *DByte*(*dop*),有 $DByte(dop)\subseteq X_0$.同时,由于细粒度污点分析在设置初始污点变量 X_0 与动态符号执行在设置初始符号污点变量 $S_0=\{\theta_1,...,\theta_{n-1},\theta_n\}$时采用相同粒度(按字节),因此,对于任意可影响危险操作 *dop* 的字节$\forall Input_i\in DByte(dop)$,有 $S(Input_i)=\theta_i(1\leqslant i\leqslant n)$.为表示可对特定危险操作造成影响的初始符号污点变量集合,定义映射 *SByte*,*SByte*(*dop*)表示所有可以对危险操作 *dop* 造成影响的输入字节在符号空间内所对应的初始符号污点变量的集合,有 $SByte(dop)=\{e|\forall Input_i\in DByte(dop),e=S(Input_i)\}$.基于以上分析,对于$\forall pc_i\in pc$,如果$\exists dop\in D_{op}$,满足$(SByte(dop)\cap Use(pc_i)!=\varnothing)$,则 pc_i 为与危险操作 *dop* 相关的路径约束项.

2.3.2 初始输入评分算法

初始输入选择过程借助上节描述的部分控制流敏感的细粒度污点分析识别程序内危险操作相关信息,然后利用这些信息对备选初始输入进行评分,具体过程如算法 1 描述.算法首先从备选初始输入集合内依次选择元素 *candidate*,以目标缺陷相关操作集合 T_{op} 内元素为目标节点进行细粒度污点分析,识别待测程序中的危险操作集合 D_{op} 以及 *candidate* 内可影响任意危险操作 *dop* 的字节集合 *DByte*(*dop*)(第 2 行).然后以 *candidate* 为输入,同时具体执行和符号执行待测程序,按照当前指令 *op*(*l*)的类型更新程序状态元组:$\langle l,C,S,pc\rangle$(第 4 行~第 14 行);若 *op*(*l*)为条件判断语句,算法依据污点分析结果判断当前条件判断语句对应的路径约束项是否与危险操作相关:如果是,则增加当前备选输入得分(第 8 行~第 10 行).备选输入将按照所得分数进行排序,得分高者将优先被选择作为动态符号执行过程的初始输入.

算法 1. 初始输入评分算法.

输入:备选初始输入集合 *SeedsSe*、目标缺陷相关安全敏感操作集合 T_{op}.

输出:备选初始输入分数列表 *Scorelist*[].

```
for each candidate in SeedsSet{
  (D_op,DByte(dop))=TaintAnalysis(T_op,candidate);  //危险操作以及关键字节识别过程详见第 2.3.1 节
  pc=True; l=l_0; score=0; C=C_0; S=S_0;
  while op(l)!=halt && op(l)!=abort {
    switch (op(l)){                                 //根据指令类型,更新程序状态元组
      case x:=e:
        C[x→C(e)]; S[x→S(e)]; l=N(l); break;
      case if (x) then l′ else l″:
        if (Use(S(x))∩SByte(dop)!=∅)                //判断是否为危险操作相关约束项
          Update(score);
        if C(x)==0
          pc.append(S(x)==0); l=l″;
        else
          pc.append(S(x)!=0); l=l′;}}
  Scorelist[candidate]=score;}
return Scorelist;
```

2.3.3 实例分析

图 3 描述了 nginx 中存在的缓冲区溢出缺陷程序片段[23,28].函数 *ngx_http_process_request_line* 的功能为接收 HTTP 请求并对其进行解析.解析过程分为对报头的处理(第 2 行、第 3 行)以及对 *uri*(通用资源标识符)的处理.函数首先对输入报文的报头进行分析检查;然后,从 *uri_start* 指定的内存区域读取报文中的 *uri* 字段内容,并分配 *uri_data* 来存储对 *uri* 的解析结果.

```
1. ngx_http_process_request_line(ngx_event_t*rev) {...
2. if (rc==NGX_AGAIN)
3.     Operations handling HTTP Request head here
4. u_char*p=rev→data→uri_start;   //uri inputs
5. u_char*u=rev→data→uri_data;    //uri results
6. u_char ch=*p++;    //current character
7. state=sw_usual;
8. while (p⇐r→uri_end) {...
9.     switch(state) {
10.      case sw_usual:
11.        if (ch=='/')
12.        {state=sw_slash; *u++=ch;}
13.        else if (ch=='.') {...}
14.        ch=*p++; break;
15.      case sw_slash:
16.        if (ch=='/')
17.          *u++=ch;
18.        else if (ch=='/')
19.          {state=sw_dot; *u++=ch;}
20.        ch=*p++; break;
21.    case sw_dot:
22.        if (ch=='.')
23.          {state=sw_dot_dot; *u++=ch;}
24.        else if {...}
25.        ch=*p++; break;
26.    case sw_dot_dot:
27.        if (ch=='/'){
28.          state=sw_slash; u-=4;
29.          while (*(u-1)!='/') u--;}
30.        else if {...}
31.        ch=*p++; break;
32.  }}}
```

Fig.3 Overflow vulnerability (CVE-2009-2629)[23,28] in nginx

图 3 nginx 中存在的溢出缺陷(CVE-2009-2629)[23,28]程序片段

程序内缺陷发生在对 *uri* 进行解析过程中,特殊构造的 *uri* 内容(例如"//.../*Maladdr*"),可在第 29 行造成缓冲区下溢错误,从而将字符串如"*Maladdr*"写入堆内存之外的指定区域.利用动态符号执行方法对 nginx 进行检测时,并非所有的请求报文作为初始输入时都能触发该缺陷.例如,当选定的输入报文 *uri* 长度小于 5 个字节时,由于约束求解器对翻转后的路径约束求解时只能改变输入中具体字段的内容,而不能改变输入的长度(否则会引起具体值与符号值之间映射的混乱)[29],因此,动态符号执行过程中,由于报头检查机制以及 *uri* 长度限制,并不能产生可以触发该缺陷的新用例(*uri* 内容含有字符串"//.../"),从而导致漏报.

本例中,函数通过变量 *p* 与 *u* 完成内存读写.这两个变量的相关操作既能被用户通过修改 *uri* 字段的内容影

响,又与程序安全性相关,易造成内存访问越界或空指针解引用等缺陷,符合上文危险操作的定义.另外,具有不同缺陷检测效果的请求报文产生的路径约束之间的主要区别为:函数内第 9 行~第 31 行对可直接影响 *p* 与 *u* 操作的 *ch* 进行比较判断的约束项出现的数量不同,而这些约束项符合上文危险操作相关约束项的定义,因此,我们通过计算路径约束中危险操作相关约束项的比重来对备选初始输入进行评分和排序,能够成功地提高可触发本例缺陷的备选初始输入的优先级.

2.4 动态符号执行

动态符号执行过程对待测程序进行缺陷检测.该过程集成面向危险操作的路径选择策略,按照与危险操作的关联程度对收集到的路径约束进行翻转,并约束求解生成覆盖新路径的新测试用例,使测试流程优先检测危险操作相关路径.如上节所述,我们采用部分控制流敏感的细粒度动态污点分析识别危险操作.由于动态分析固有的不全面性以及潜在的复杂控制流影响,使得当前已识别的危险操作以及与它相关的路径被检测完成后,未被覆盖的程序空间内仍可能存在未被检测的危险操作和相关路径.因此,面向危险操作的路径选择策略循环执行危险路径遍历与危险路径定位两个步骤:前者对已知危险操作相关路径进行遍历检测;后者快速定位未覆盖程序空间内未检测的危险操作和相关路径,以减少漏报.

算法 2 描述了面向危险操作的路径选择策略的主要流程.列表 *eWorklist* 与 *lWorklist* 分别包含危险路径遍历与危险路径定位过程所使用的测试用例集合.算法首先依据初始输入选择过程所选择的初始输入进行初始化操作(第 1 行).当 *eWorklist* 与 *lWorklist* 均不为空时,路径选择策略迭代调用危险路径遍历过程(explore)与危险路径定位过程(locate)(第 5 行~第 7 行).遍历过程循环选择 *eWorklist* 内测试用例为输入,分析识别危险操作及其相关信息,生成并调度测试用例.新测试用例如果可以覆盖新的危险操作或者执行未检测的危险操作相关约束项组合,则添加该用例到 *eWorklist*.列表 *eBlocksCovered* 存储危险路径遍历过程中覆盖的基本块.当 *eWorklist* 为空时,危险路径定位过程以列表 *eBlocksCovered* 为起点,探索当前未被覆盖的程序空间,如果发现未被执行的危险操作或未检测的危险操作相关约束项组合,则将对应的测试用例添加到 *eWorklist*,然后,算法重返至危险路径遍历过程开始新一轮的探测.测试用例生成过程中,算法为每个用例设置和维护一个 *bound* 值(第 3 行).对于每组路径约束,不同于深度优先、广度优先等算法[1]每次选择一个路径约束项翻转,然后求解生成一个新用例,算法借鉴代搜索算法(generational search)[3],每轮分析均翻转路径约束内第 *bound* 个约束项之后的所有约束项,同时生成多个新路径约束,然后约束求解生成多个新测试用例,最后更新新用例的 *bound* 值为其被生成前对应的被翻转的路径约束项编号,以避免重复遍历.

同样以图 3 所示缺陷为例,相对于传统动态符号执行方法依据新测试用例所能覆盖的新增基本块数量来进行测试用例调度,面向危险操作的路径选择策略在识别到变量 *p* 与 *u* 的相关操作为危险操作后,算法随即进入危险路径遍历阶段,优先对函数内第 9 行~第 31 行与变量 *p* 与 *u* 相关的危险操作相关约束项组合进行遍历检查,进而触发缺陷.在对相关约束项组合检查完毕后,算法转入危险路径定位阶段,探索未被覆盖的状态空间,定位未检测的危险操作和相关路径.

算法 2. 面向危险操作的路径选择策略.

输入:初始输入文件 *inputSeed*.

```
1.  eWorklist={inputSeed}; lWorklist={inputSeed};        //初始化测试用例列表
2.  eBlocksCovered={}; lBlocksCovered={};                //两个过程内被覆盖的基本块列表
3.  inputSeed.eBound=0; inputSeed.lBound=0;              //设置 bound 值
4.  Run&Check(inputSeed);                                //执行测试用例和错误检查
5.    while (eWorklist!=∅) && (lWorklist!=∅) {
6.      Explore(eWorklist);                              //危险路径遍历
7.      Locate(lWorklist,eBlocksCovered);}               //危险路径定位
```

2.4.1 危险路径遍历

危险路径遍历过程按照以下原则生成和调度测试用例:

a) 新生成的测试用例可以覆盖未被检测的危险操作相关约束项组合;

b) 被优先调度的测试用例可产生更多未被检测的危险操作相关约束项组合.

算法 3 描述了危险路径遍历过程.算法选择列表 *eWorklist* 的前端用例 *input*,进行污点分析识别危险操作和收集路径约束(第 2 行~第 5 行).然后分析路径约束项 *pc*[*i*]与任意已知危险操作 *dop* 是否可被相同输入字节影响,判断 *pc*[*i*]是否为危险操作相关约束项.如果是,则算法翻转该路径约束项并进行求解,生成的新用例可协助检测由翻转 *pc*[*i*]形成的新的危险操作相关约束项组合(第 7 行~第 11 行).在调度测试用例时,为了更快地检查更多危险操作相关约束项组合,理想的方案是对新用例逐个进行细粒度污点分析和路径约束收集,分析其中未被检查的与危险操作相关的路径约束项数量.但在实际测试任务中,由于动态符号执行需要较多的时空资源[3,7,9],该方案会极大地增加测试开销.因此,我们提出一种启发式方法:首先,对于当前输入 *input*,算法在实施污点分析识别危险操作的同时,额外统计和更新不同字节可影响的不同危险操作的个数(第 4 行),借此反映不同输入字节 *byte* 对危险操作的影响程度,本文称其为"关键字节影响因子",用 *Weight*(*byte*)表示;以此为基础,算法为每个新测试用例计算权重 *eScore*,然后按照权重大小顺序将其加入列表 *eWorklist* 进行调度(第 12 行~第 14 行).具体来说,假设当前输入 *input* 产生的路径约束为 *pc*,约束求解器以翻转路径约束项 *pc*[*i*]后所得新路径约束为输入,求解得到测试用例为 *newInput*,则

$$newInput.eScore=\sum_{k=0}^{len(dbyte[])}Weight(dbyte[k]),dbyte[]=Use(pc[i])\cap SByte(dop).$$

该打分规则分析新测试用例生成前被翻转的路径约束项,综合该约束项中可影响危险操作的输入字节以及这些字节对应的关键字节影响因子来计算新测试用例的权重.算法优先调度那些由权重较大的路径约束项翻转后求解所生成的新用例,因为变化这些路径约束项所产生的新路径约束可同时影响多个危险操作的相关数据,预计可对危险操作相关约束项的组合造成更多变化.危险路径遍历过程忽略了由翻转与危险操作无关的路径约束项所生成的测试用例(第 13 行),如果这部分用例可以覆盖新的基本块,那么它们将在随后的危险路径定位阶段被生成和调度.

算法 3. 危险路径遍历算法.

输入:*eWorklist*.

```
1.  while (eWorklist!=∅) {
2.    input=PickFirstItem(eWorklist);                    //选择评分最高的测试用例
3.    eBlocksCovered+=TraceBlocks(input)                 //更新被 input 覆盖的基本块
4.    (D_op,DByte(dop))+=TaintAnalysis(T_op,input);      //更新危险操作集合
5.    pc=PCCollection(input);                            //动态符号执行,收集路径约束
6.    InsertDC(pc);                                      //插入缺陷触发约束,检查缺陷是否可触发
7.      for (i=input.eBound; i<|pc|; i++) {
8.        if ((SByte(dop)∩Use(pc_i)!=∅) {                //识别危险操作相关路径约束
9.          if (pc[0...(i−1)] and not (pc[i])) has a solution I {
10.           newInput=input+I; newInput.eBound=i;
11.           Run&check(newInput);                       //对新用例进行缺陷检查
12.           newInput.eScore=eScore();                  //根据关键字节影响因子对测试用例评分
13.           if (newInput.eScore!=0)
14.             eWorkList+=newInput; } } } }
```

2.4.2 危险路径定位

危险路径定位过程需要完成以下两个任务.

a) 快速探索待测程序未被覆盖的状态空间;

b) 识别未被执行的危险操作,或者与已知危险操作相关的未被检测的路径约束项组合.

为了能够快速探索待测程序状态空间,现有工作通常统计执行不同测试用例所能覆盖的基本块,然后与当前已被覆盖的基本块比较,计算新增基本块数量来反映测试用例对基本块覆盖的贡献,借此评定测试用例被调度的优先级.以图 4 所示未被覆盖的代码片段为例,假设翻转第 1 行、第 3 行等条件语句后求解生成的新用例分别为 $t_0,t_1,\ldots,t_n$.记 $score(t)$为采用测试用例 t 进行测试时所覆盖的新增基本块数量,有 $score(t_i)=a_i+m$,其中,$0\leqslant i\leqslant n$.假设 $a_0>a_1>\ldots>a_n$,则 t_0 将被优先调度进行路径约束的收集和新测试用例的生成.假定新一轮中,以 t_0 为输入,所生成的新测试用例集合为 *newset*.随着 t_0 的执行,测试用例 $t_1,\ldots,t_n$ 所能覆盖的新增基本块数量更新为 $a_1,\ldots,a_n$,它们原有的分数 $score(t_i)=a_i+m$,其中,$1\leqslant i\leqslant n$,将不再反映其对基本块覆盖的贡献.在计算用例对基本块覆盖的贡献时,若评估算法不仅运行 *newset* 内的新测试用例,而且重复运行当前工作列表内所有的备选用例(如 $t_1,\ldots,t_n$ 等),更新每个用例所能产生的新增基本块[9],则称其为完全覆盖评估方法(total_eval).完全覆盖评估能够精确地计算每个用例所能产生的新增基本块,但是评估过程会消耗大量资源.相对而言,若算法在评估用例时只针对集合 *newset* 内的新用例,统计执行新用例时所覆盖的新增基本块数量[7],则称其为部分覆盖评估方法(part_eval).部分覆盖评估方法能够有效减少评估用例过程所需的资源,但是由于部分覆盖评估方法在评估过程中只关注新生成的用例,所以该方法的精确度不高,且容易受到路径发散问题(divergence)[3,5]的影响.路径发散是指动态符号执行系统预计测试用例 t_i 可以指导程序抵达条件分支指令 c_i,并向预订方向迁移.然而,由于实际运行过程的一些不确定因素(如多线程竞争、浮点数对路径约束收集的影响等),使得执行路径无法抵达 c_i 或无法向预订方向迁移.路径发散现象普遍存在于动态符号执行中,例如,SAGE 生成的测试用例发生路径发散的几率高达 60%[3].我们在实验时也发现,很多由不同路径约束求解所得到的测试用例实际上却执行同一条路径.在本例中,假设测试用例 t_0 与 t_k ($0<k<n$)由于路径发散执行同一路径,随着 t_0 被调度,$score(t_k)$虽然很高,但不能覆盖任何新基本块.当 t_k 被调度时,测试资源浪费在已被执行的路径上,从而降低了部分覆盖评估方法的效率.

```
1. if (in==c0)
2.    a0 new blocks
3. else if (in==c1)
4.    a1 new blocks
5.    ...
6. else if (in==cn)
7.    an new blocks
8. m new blocks
```

Fig.4 Example code of test case evaluation

图 4 测试用例评估示例代码

我们在计算不同测试用例对基本块覆盖贡献时,引入了对执行效率和路径发散问题的考量:首先,每轮测试用例评估时,只计算集合 *newset* 内新测试用例的权重,以减少评估过程的资源消耗;其次,维护一个基本块首地址的集合 H_b,在对新用例进行评估时,该集合累计更新运行不同用例所覆盖的基本块.具体来说,对于任意新测试用例 t,令 $cover(t)$为采用 t 进行测试时可以覆盖的基本块集合,首先计算用例 t 能产生的新增基本块数量为 $score(t)=|cover(t)-H_b|$,然后更新集合 $H_b=H_b\cup cover(t)$.以上述代码片段为例,对于首先被评估的 t_0,有 $score(t_0)=a_0+m$,$H_b=H_b\cup cover(t_0)$;后续被评估的用例分数变化为 $score(t_i)=a_i$,$H_b=H_b\cup score(t_i)$,其中,$1\leqslant i\leqslant n$.采用这种计算方法,由于路径发散而执行重复路径的 t_k 可以被正确地鉴别,因为有 $score(t_k)=0$.由第 3.2.3 节的实验可知,虽然此评估方法提高了前期被评估的用例的优先级,相对于完全覆盖评估方法损失了部分精确度,导致测试任务中用例调度次数的增加,但该方法可有效地减少用例评估过程所需时间;同时,相对于部分覆盖评估,该评估方法可消除由路径发散引入的冗余用例,从而达到加速探索待测程序的目的.

危险路径定位的流程如算法 4 所示.定位算法以危险路径遍历过程中被覆盖的基本块为起点,更新当前已被覆盖的基本块列表 *lBlocksCovered*(第 1 行).然后选择备选测试用例列表 *lWorklist* 前端的用例作为输入,按照动态符号执行流程生成新测试用例(第 8 行~第 11 行).对于每个新测试用例,算法按照上文描述的评估方法计算它们对基本块覆盖的贡献(第 12 行).新用例将按照贡献值依次进行调度.对于贡献为 0 的测试用例,为了能够快速地定位新危险操作和相关路径,定位算法在调度过程中选择忽略这部分用例(第 15 行、第 16 行).需要指出的是,虽然相关工作[3,7]实验结果表明,运行这类用例不仅会消耗大量测试时间,而且对缺陷的触发几乎没有贡献,但在极端情况下,这种忽略特定测试用例的做法可能会导致漏报.算法运行过程中,需要中断危险路径定位过程,转而进行危险路径遍历的情形有如下两种:

(1) 危险路径定位过程执行到新的危险操作,需要调用遍历过程对该操作相关路径进行遍历检查.对此,

危险路径定位算法针对被调度的用例实施污点分析,更新危险操作和可影响危险操作的字节集合.如果集合 D_{op} 内发现新的危险操作,或者 *DByte*(*dop*)内存在可以影响危险操作的新的字节,则将当前用例加入危险路径遍历的测试用例列表 *eWorkList*(第 3 行~第 6 行),然后将控制流程转交给遍历算法(第 17 行).

(2) 危险路径定位过程以新的上下文环境执行已知危险操作,需要分析当前执行路径中是否引入了危险操作相关约束项新的组合.如果有,则调用遍历过程对这些组合进行检查.对此,算法首先分析新用例是否由翻转危险操作相关约束项然后求解所生成:若是,则分析该用例是否可以覆盖新的基本块,借此来判断是否引入未被检测的危险约束项组合.

如果两者均满足,则将当前测试用例添加到列表 *eWorkList*(第 13 行、第 14 行),流程转至遍历算法(第 17 行).

算法 4. 危险路径定位算法.

输入:*lWorklist*; *eBlocksCovered*.

```
1.  lBlocksCovered+=eBlocksCovered;                    //更新当前已被覆盖基本块列表
2.  while (lWorklist!=∅) {
3.    input=PickFirstItem(lWorklist);
4.    (D_op,DByte(dop))+=TaintAnalysis(T_op,input);
5.    if new D_op or DByte(dop) detected {             //如 input 执行新危险操作,则转至遍历算法
6.      eWorkList+=newInput; break;}
7.    lBlocksCovered+=TraceBlocks(newInput);
8.    pc=PCCollection(input);
9.    for (i=input.lBound; i<|pc|; i++) {
10.     if (pc[0...(i~1)] and not(pc[i])) has a solution I {
11.       newInput=SeedInput+I; newInput.lBound=i;
12.       newInput.lScore=lScore();                    //计算新测试用例对基本块覆盖的贡献
13.       if ((SByte(dop)∩Use(pc_i)!=∅) && newInput.lScore!=0)   //若为未检测相关约束,则转至遍历
14.         eWorkList+=newInput;
15.       else if newInput.lScore!=0
16.         lWorkList+=newInput}}
17.   if eWorkList!=∅ break;}
```

3 实现和评估

3.1 原型工具实现

为了评估面向危险操作的动态符号执行方法的有效性和效率,实现了原型工具 CrashFinder,包含初始输入选择模块和动态符号执行模块两部分.该工具基于动态插装平台 Valgrind[30],将程序执行中的二进制指令翻译为中间语言 VEX.在此基础上,实现部分控制流敏感的细粒度污点分析,识别危险操作和可影响危险操作的输入字节,为初始输入选择模块提供支持.动态符号执行模块同样分析被执行指令的 VEX 语言,维护程序执行的真实上下文与符号上下文环境,将执行路径中与符号化污点数据相关的指令转换成对输入数据的约束条件.CrashFinder 采用 STP[31]约束求解器对翻转的路径约束进行求解,构造新的测试用例.

为了方便与相关工作比较,CrashFinder 的目标缺陷设置为内存读写错误、整型溢出与除数零错误.相应的关注安全敏感操作集合 T_{op} 包含内存读写操作、内存分配函数(*malloc* 等)以及除操作.为了在测试过程中通过符号化的方式有针对性地检测错误,需要对关注的缺陷定义相应的缺陷模型.表 1 为上述危险操作对应的缺陷触发条件,其中,*base* 代表缓冲区基地址,*len*(*buf*)为缓冲区长度.这些信息是由 CrashFinder 通过插装内存分配函数以及分析函数调用栈获得.当待测程序执行至危险操作时,CrashFinder 在已收集的路径约束基础上,按照表 1

规则添加缺陷触发约束并进行求解.若可解,则生成满足危险语句执行条件和缺陷触发条件的测试用例.CrashFinder 监控所有新测试用例的执行,如果产生关注缺陷类型相关的信号中断,则定位并输出相关数据以辅助程序员重现缺陷.

Table 1 Constraints of defects

表 1 缺陷触发条件

危险操作类型	缺陷触发条件
r=dividend/divisor	*divisor*==0
**p*	*p*==*Null*; *p*<*base*∨*p*>*base*+*len*(*buf*)
malloc(*arg*)等内存分配函数	*arg*>max(*int*) (32 位系统为 0xffffffff)

3.2 性能评估

我们选择一系列开源软件,对 CrashFinder 的初始输入评分算法的有效性、面向危险操作的路径选择策略的有效性和危险路径定位算法中测试用例评估方法的效率等方面进行测评.实验运行的环境为:CPU Intel Core i5-750,主频 2.67GHz,内存 4G,系统 Unbuntu12.10.实验中如没有特殊要求,每个测试任务均被赋予足够的运行时间,直至工作列表中没有能覆盖新基本块的测试用例存在.为了分辨不同用例是否触发同一缺陷,对于测试过程中生成的可触发缺陷或致使待测程序崩溃的测试用例,均通过人工分析来确定缺陷根源.

3.2.1 初始输入评分算法的有效性

为了验证 CrashFinder 初始输入评分算法的有效性,我们选择一组基准程序,并为每个程序提供一组备选初始输入.首先利用 CrashFinder 的初始输入选择模块对各个备选输入进行评分,然后选取各个分数段内的备选输入作为动态符号执行模块的初始输入,对基准程序进行测试,并记录每组实验结果,用来验证不同备选初始输入的评估得分与它们的缺陷检测表现之间的联系.实验采用的基准程序一部分选自相关工作中的开源软件,包括文献[8]中的 readelf 2.23.52、文献[9]中 libjpeg7 软件包内 cjpeg 程序与 swftool 0.9.0 以及文献[23]中的 nginx 0.6.32.这些基准程序被研究人员用于评价各自的方法,具有一定的代表性.基准程序还包括 Linux 平台常用软件 ImageMagic 6.7.7-10 软件包内的 convert 程序.针对每个基准程序,实验所使用的备选初始输入集合中,包含相关工作实验所使用的初始输入文件以及程序自带测试用例集合中部分合法格式输入.需要说明的是,由于实现平台限制,实验中 nginx 经过修改,其接收数据方式为从文件读入.

实验结果见表 2:第 1 列和第 2 列分别描述了基准程序名称和合法输入的格式;第 3 列和第 4 列描述了备选初始输入的编号和大小;第 5 列和第 6 列描述了初始输入选择模块的分析时间和评分结果;第 7 列~第 10 列分别描述了使用备选输入测试基准程序初始所覆盖的基本块数量、完成测试所用时间、产生的测试用例数量和发现的缺陷数.

Table 2 Effectiveness of initial input scoring algorithm in CrashFinder

表 2 CrashFinder 初始输入评分算法的有效性

Program	Format	备选初始输入							
		ID	Size	EvalTime (s)	Scores	BlocksAtStart	TestTime (s)	TestCases	Defects
cjpeg	BMP	ABmp	712Bytes	65	5 720	3 073	12 243	1 321	1
		Bmp1	10KB	121	10 720	3 293	16 224	1 732	3
		Bmp2	9.05KB	145	15 112	3 334	15 235	1 654	3
nginx	HTTP	Http1	4Bytes (uri)	29	4 529	5 423	7 492	1 832	0
		Http2	10Bytes (uri)	36	6 744	5 346	7 982	1 932	1
		Http3	20Bytes (uri)	39	9 843	5 543	8 321	2 324	1
readelf	ELF	CElf	2.93KB	17	9 023	4 386	3 492	3 596	0
		Elf1	56.6KB	42	12 031	4 425	4 735	4 214	3
		Elf2	29.4KB	32	15 202	4 391	4 243	4 712	3
convert	JPG	Jpg1	49KB	132	6 953	13 834	4 134	376	1
		Jpg2	45.0KB	131	13 812	13 390	4 120	335	1
		Jpg3	69KB	143	15 224	13 206	4 430	415	1
swftool	SWF	ASwf	712Bytes	20	934	2 770	9 620	2 987	2
		Swf1	3.46KB	32	1 120	3 427	13 849	3 408	2
		Swf2	11.2KB	45	2 231	3 424	15 393	4 126	2

1) 由 EvalTime (s)列可知,对备选初始输入进行评分所引入的时间开销相对较小,实验中,评分工作多在一分钟左右即可完成.
2) 由 Defects 列和 Scores 列可知,对于每个基准程序,评分高的备选初始输入触发的缺陷数量不低于评分低的备选初始输入.也就是说,我们的初始输入评分算法能够很好地刻画备选初始输入的缺陷发掘能力.例如,在对 cjpeg 进行动态符号执行测试时,选自文献[9]的输入文件 ABmp 与其他备选初始输入均能触发位于 jmemmgr.c:406 的除数零错误.然而出于测试开销考量,文献[9]删除了 *ABmp* 中大量图片内容相关信息而仅保留满足 BMP 格式要求的数据.因此,使用 *ABmp* 为初始输入进行的动态符号执行过程中,由于图片内容缺失,测试流程不能覆盖与图片内容相关的内存分配函数,因此不能触发该函数由于参数发生整型溢出而引起的内存访问越界缺陷.相对而言,在采用高评分初始输入 *Bmp*2 后,我们在 cjpeg 中额外发现了两个内存访问越界错误(rdbmp.c:212,jmemmgr.c:428).类似地,初始输入选择模块不仅可以对 nginx 的备选初始输入按照其 *uri* 长度进行排序评分,而且相对于文献[8]采用文件 *CElf* 为初始输入测试 readelf 时没有发现缺陷,我们采用高评分的初始输入 *Elf*2 在 readelf 中发现了 3 个内存读写错误.
3) 对比 Size 列、BlocksAtStart 列以及 Scores 列可见,部分实例中(例如 *Elf*1 与 *Elf*2、*Bmp*1 与 *Bmp*2),备选初始输入的缺陷触发能力与其文件大小或初始所能覆盖的基本块数量并没有直接联系.即相对于传统黑盒测试中普遍采用的通过简单计算备选输入文件大小或者它们所覆盖的基本块数量来选定初始输入的方法[32],初始输入评分算法在协助动态符号执行技术选择初始输入方面更加精确.

3.2.2 面向危险操作的路径选择策略的有效性

Avalanche[9]为 Linux 平台针对内存访问错误,空指针解引用以及除数零错误进行检测的动态符号执行工具,它依据新用例产生的新增基本块数量对其进行调度.现将 CrashFinder 与 Avalanche 的缺陷检测能力进行比较,用以反映面向危险操作的路径选择策略和传统基于新增基本块的路径选择策略的缺陷检测能力的差异.为了方便比较,这里选取 Avalanche 工作实验中采用的 swftool 0.9.0,libjpeg7 软件包内的 cjpeg 程序、libquicktime-1.1.2 内的 qtdump 程序、libmpeg3-1.8 内的 mpeg3dump 程序以及 speex-1.2rc 作为分析对象.

表 3 为采用相同初始输入分别运行 CrashFinder 和 Avalanche 的实验结果.第 1 列为测试对象,第 2 列~第 5 列和第 7 列~第 10 列分别描述两个工具产生的测试用例数、缺陷检测数量、触发第 1 个缺陷所用时间以及触发所有不同缺陷所用时间.Switch 列为 CrashFinder 测试过程中危险路径遍历与危险路径定位两个阶段之间的切换次数.

Table 3 Comparison of defect detection ability between CrashFinder and Avalanche

表 3 CrashFinder 与 Avalanche 缺陷检测能力比较

Program	CrashFinder					Avalanche			
	TestCases	Defects	T_{first} (s)	T_{total} (s)	Switch	TestCases	Defects	T_{first} (s)	T_{total} (s)
qtdump	230	1	124	124	1	172	1	369	369
speexenc	76	1	293	293	1	44	1	454	454
swftool	3 408	2	180	408	2	3 278	2	616	871
mpeg3dump	23 569	2	2	66	4	20 433	2	2	135
cjpeg	1 654	3	214	434	2	1 365	3	211	723

1) 对比两个工具的 Defects 列和 T_{total}(s)列可知,在保证发现所有缺陷的前提下,CrashFinder 比 Avalanche 花费更少的时间,这反映了面向危险操作的路径选择策略在加速触发缺陷方面的优势.
2) 对比两个工具的 T_{first}(s)列可知,对于多数程序,CrashFinder 比 Avalanche 能够更快地触发第 1 个缺陷.两个例外情况是,在 cjpeg 与 mpeg3dump 内分别存在一个除数零错误(jmemmgr.c:406)与空指针解引用错误(mpeg3ifo.c:509).由于两个工具均能结合初始输入产生的路径约束与上文缺陷触发模型中对应的缺陷触发条件,然后借助约束求解器直接生成可以触发这两个缺陷的测试用例,因此这两例程序对应的 T_{first} (s)相差不大.
3) 由 Switch 列可知,对于多数程序,仅依据初始输入识别的危险操作以及相关路径信息,并不能覆盖待

测程序内所有危险操作.因此,相对于传统的结合细粒度污点分析与危险操作的测试方法[21−23],面向危险操作的路径选择策略可以对待测程序内危险操作进行更全面的检测.

3.2.3 危险路径定位算法的效率

如第 2.4.2 节所描述,危险路径定位算法中,测试用例评估方法在完全覆盖评估方法的注重精度而消耗资源与部分覆盖评估方法的节约资源而牺牲精度之间取折中,加速动态符号执行对待测程序状态空间的探索和对新危险操作的定位.表 4 描述了分别采用危险路径定位算法中测试用例评估方法、部分覆盖评估方法与完全覆盖评估方法来评估和调度测试用例,对待测程序进行动态符号执行的实验结果.其中,第 1 列描述了待测程序名称,第 2 列~第 5 列、第 6 列~第 9 列和第 10 列~第 13 列分别详细描述了使用 3 种评估调度策略的测试时间、测试过程中不同阶段所占时间百分比(测试用例评估/路径约束收集/约束求解以及错误检查)、基本块覆盖总数以及测试用例数.

Table 4 Efficiency of critical paths locating method

表 4 危险路径定位算法的效率

Program	危险路径定位中测试用例评估方法				部分覆盖评估方法				完全覆盖评估方法			
	Total	PCT	Blocks	Case	Total	PCT	Blocks	Case	Total	PCT	Blocks	Case
qtdump	3 552	6/53/41	6 698	211	5 029	5/50/45	6 678	337	4 427	11/48/41	6 687	162
speex	2 048	2/3/95	4 487	44	2 533	1/6/93	4 432	74	2 171	8/3/89	4 468	44
swftool	12 943	21/30/49	4 329	3 415	14 082	18/42/40	4 301	4 608	13 786	46/41/13	4 231	3 158
mpeg3	32 837	30/2/68	5 138	25 573	38 340	24/4/72	5 086	29 365	37 403	54/1/45	5 132	19 033
cjpeg	13 223	2/22/76	3 795	1 475	15 618	2/20/78	3 789	2 004	14 216	7/21/72	3 839	1 252
readelf	4 082	18/9/73	5 165	4 023	5 177	18/6/76	4 993	5 087	5 766	38/2/60	4 884	2 563
convert	3 930	8/37/55	17 997	287	4 477	6/48/46	18 019	632	4 394	19/32/49	17 973	156

1) 由不同策略的 Total 列可得,相对于其他两种评估方法,危险路径定位算法中的测试用例评估方法可以更快地完成对待测程序的测试任务.
2) 对比不同策略的 PCT 列和 Case 列可得,相对于部分覆盖评估方法,危险路径定位中测试用例评估方法可以剪除大量冗余测试用例;相对于完全覆盖评估方法,该评估方法虽然由于损失部分精度,导致完成测试所需测试用例数量增加,但会大幅减少评估测试用例的过程所消耗的时间比重,从而加速对待测程序的探索.

3.2.4 缺陷统计

如表 5 所示,CrashFinder 在上述实验程序以及常见开源软件中总共发现了内存访问越界、空指针解引用以及除数零错误这 3 种类型总共 20 个缺陷.正如第 3.2.1 节所分析,相对于其他针对同样程序进行实验的相关动态符号执行工作,CrashFinder 利用初始输入选择模块评估和选择高质量的初始输入,因此在 cjpeg,readelf 以及 convert 等程序中发现了更多缺陷.经过人工核查后,其中包含 10 个未知缺陷,我们已将相关信息以及对应测试用例发送给相关软件作者.

Table 5 Defects detected by CrashFinder

表 5 CrashFinder 发现的缺陷

Programs	cjpeg	readelf	convert	swftool	swftool	qtdump	speexenc	mpeg3dump	mupdf
Version	7	2.23.52	6.7.7	0.9.0	0.9.2	1.1.2	1.2rc	1.8	0.1
#Defects	3	3	1	2	5	1	1	2	2

4 总 结

提出了面向危险操作的动态符号执行方法,基于缺陷发生与危险操作密切相关的思想,分析和识别危险操作以及与危险操作相关的路径约束项.一方面将这些信息用于协助动态符号执行选择高效的初始输入;另一方面用于指导动态符号执行测试用例的生成和调度,对待测程序中易发生缺陷的路径进行优先系统地探测.对原型系统 CrashFinder 的实验结果表明,面向危险操作的动态符号执行方法能够快速、有效地检测更多的缺陷.

References:

[1] 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 (PLDI). New York: ACM Press, 2005. 213−223. [doi: 10.1145/1065010.1065036]

[2] Kim Y, Kim Y, Kim T, Lee G, Jang Y, Kim M. Automated unit testing of large industrial embedded software using concnlic testing. In: Proc. of the 28th ACM/IEEE Int'l Conf. on Automated Software Engineering (ASE). New York: ACM Press, 2013. 518−528. [doi: 10.1109/ASE.2013.6693109]

[3] Godefroid P, Levin MY, Molnar D. Automated whitebox fuzz testing. In: Proc. of the 2008 Network and Distributed Systems Security (NDSS). San Diego: The Internet Society, 2008. 151−166.

[4] Cadar C, Sen K. Symbolic execution for software testing: three decades later. Communications of the ACM, 2013,56(2):82−90. [doi: 10.1145/2408776.2408795]

[5] Lu XC, Li G, Lu K, Zhang Y. High-Trusted-Software-Oriented automatic testing for integer overflow bugs. Ruan Jian Xue Bao/Journal of Software, 2010,21(2):179−193 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/3785.htm [doi: 10.3724/SP.J.1001.2010.03785]

[6] Godefroid P, Levin MY, Molnar DA. Active property checking. In: Proc. of the 8th ACM Int'l Conf. on Embedded Software (EMSOFT). New York: ACM Press, 2008. 207−216. [doi: 10.1145/1450058.1450087]

[7] Molnar D, Li XC, Wagner DA. Dynamic test generation to find integer bugs in x86 binary linux programs. In: Proc. of the 18th USENIX Security Symp. San Diego: USENIX Association, 2009. 67−82.

[8] Chen B, Zeng QK, Wang WG. Crashmaker: An improved binary concolic testing tool for vulnerability detection. In: Proc. of the 29th Annual ACM Symp. on Applied Computing (SAC). New York: ACM Press, 2014. 1257−1263. [doi: 10.1145/2554850.2554875]

[9] Isaev IK, Sidorov DV. The use of dynamic analysis for generation of input data that demonstrates critical bugs and vulnerabilities in programs. ACM Journal of Programming and Computing Software, 2010,36(4):225−236 [doi: 10.1134/S0361768810040055]

[10] Chen K, Feng DG, Su PR. Dynamic overflow vulnerability detection method based on finite CSP. Chinese Journal of Computers, 2012,35(5):898−909 (in Chinese with English abstract). [doi: 10.3724/SP.J.1016.2012.00898]

[11] Majumdar R, Sen K. Hybrid concolic testing. In: Proc. of the 29th Int'l Conf. on Software Engineering (ICSE). Washington: IEEE Computer Society, 2007. 416−426. [doi: 10.1109/ICSE.2007.41]

[12] Bounimova E, Godefroid P, Molnar D. Billions and billions of constraints: Whitebox fuzz testing in production. In: Proc. of the 2013 Int'l Conf. on Software Engineering (ICSE). Piscataway: IEEE Press, 2013. 122−131. [doi: 10.1109/ICSE.2013.6606558]

[13] Inozemtseva L, Holmes R. Coverage is not strongly correlated with test suite effectiveness. In: Proc. of the 36th Int'l Conf. on Software Engineering (ICSE). New York: ACM Press, 2014. 435−445. [doi: 10.1145/2568225.2568271]

[14] Cadar C, Ganesh V, Pawlowski PM, Dill DL, Engler DR. Exe: Automatically generating inputs of death. In: Proc. of the 13th ACM Conf. on Computer and Communications Security (CCS). New York: ACM Press, 2006. 322−335. [doi: 10.1145/1180405.1180445]

[15] 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 (OSDI). Berkeley: USENIX Association, 2008. 209−224.

[16] Marinescu PD, Cadar C. Make test-zesti: A symbolic execution solution for improving regression testing. In: Proc. of the 34th Int'l Conf. on Software Engineering (ICSE). Piscataway: IEEE Press, 2012. 716−726. [doi: 10.1109/ICSE.2012.6227146]

[17] Burnim J, Sen K. Heuristics for scalable dynamic test generation. In: Proc. of the 23rd IEEE/ACM Int'l Conf. on Automated Software Engineering (ASE). Washington: IEEE Computer Society, 2008. 443−446. [doi: 10.1109/ASE.2008.69]

[18] Cui ZQ, Wang LZ, Li XD. Target-Directed concolic testing. Chinese Journal of Computers, 2011,34(6):953−964 (in Chinese with English abstract). [doi: 10.3724/SP.J.1016.2011.00953]

[19] Su T, Fu ZL, Pu GG, He JF, Su ZD. Combining symbolic execution and model checking for data flow testing. In: Proc. of the 2015 Int'l Conf. on Software Engineering (ICSE). Piscataway: IEEE Press, 2015. 654−665. [doi: 10.1109/ICSE.2015.81]

[20] Babić 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 (ISSTA). New York: ACM Press, 2011. 12−22. [doi: 10.1145/2001420.2001423]

[21] Ganesh V, Leek T, Rinard M. Taint-Based directed whitebox fuzzing. In: Proc. of the 31st Int'l Conf. on Software Engineering (ICSE). Washington: IEEE Computer Society, 2009. 474−484. [doi: 10.1109/ICSE.2009.5070546]

[22] Wang TL, Wei T, Gu GF, Zou W. TaintScope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection. In: Proc. of the 2010 IEEE Symp. on Security and Privacy (S&P). Washington: IEEE Computer Society, 2010. 497−512. [doi: 10.1109/SP.2010.37]

[23] Haller I, Slowinska A, Neugschwandtner M, Bos H. Dowsing for overflows: A guided fuzzer to find buffer boundary violations. In: Proc. of the 22nd USENIX Conf. on Security (SEC). Berkeley: USENIX Association, 2013. 49−64.

[24] CVE statistics for integer vulnerabilities. http://www.cve.mitre.org/cgi-bin/cvekey.cgi?keyword=integer

[25] Wang WG, Zeng QK. Evaluating initial inputs for concolic testing. In: Proc. of the 2015 Int'l Symp. on Theoretical Aspects of Software Engineering (TASE). Piscataway: IEEE Computer Society, 2015. 47−54. [doi: 10.1109/TASE.2015.14]

[26] Chen S, Kalbarczyk Z, Xu J, Iyer RK. A data-driven finite state machine model for analyzing security vulnerabilities. In: Proc. of the 2003 IEEE Int'l Conf. on Dependable Systems and Networks (DSN). Piscataway: IEEE Computer Society, 2003. 605−614. [doi: 10.1109/DSN.2003.1209970]

[27] Bao T, Zheng YH, Lin ZQ, Zhang XY, Xu DY. Strict control dependence and its effect on dynamic information flow analyses. In: Proc. of the 19th Int'l Symp. on Software Testing and Analysis (ISSTA). New York: ACM Press, 2010. 13−24. [doi: 10.1145/1831708.1831711]

[28] CVE-2009-2629: Buffer underflow vulnerability in nginx. 2009. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2009-2629

[29] Marinescu PD, Cadar C. KATCH: High-Coverage testing of software patches. In: Proc. of the 9th Joint Meeting on Foundations of Software Engineering (ESEC/FSE). New York: ACM Press, 2013. 235−245. [doi: 10.1145/2491411.2491438]

[30] Nethercote N, Seward J. Valgrind: A framework for heavyweight dynamic binary instrumentation. In: Proc. of the 2007 ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI). New York: ACM Press, 2007. 89−100. [doi: 10.1145/1250734.1250746]

[31] Ganesh V, Dill DL. A decision procedure for bit-vectors and arrays. In: Proc. of the 19th Int'l Conf. on Computer Aided Verification (CAV). Berlin: Springer-Verlag, 2007. 519−531. [doi: 10.1007/978-3-540-73368-3_52]

[32] Rebert A, Cha SK, Avgerinos T, Foote J, Warren D, Grieco G, Brumley D. Optimizing seed selection for fuzzing. In: Proc. of the 23rd USENIX Conf. on Security Symposium (SEC). Berkeley: USENIX Association, 2014. 861−875.

附中文参考文献:

[5] 卢锡城,李根,卢凯,张英.面向高可信软件的整数溢出错误的自动化测试.软件学报,2010,21(2):179−193. http://www.jos.org.cn/1000-9825/3785.htm [doi: 10.3724/SP.J.1001.2010.03785]

[10] 陈恺,冯登国,苏璞睿.基于有限约束满足问题的溢出漏洞动态检测方法.计算机学报,2012,35(5):898−909. [doi: 10.3724/SP.J.1016.2012.00898]

[18] 崔展齐,王林章,李宣东.一种目标制导的混合执行测试方法.计算机学报,2011,34(6):953−964. [doi: 10.3724/SP.J.1016.2011.00953]

王伟光(1984－),男,河北衡水人,博士生,主要研究领域为信息安全,软件安全测试.

孙浩(1987－),男,博士生,主要研究领域为信息安全,程序分析.

曾庆凯(1963－),男,博士,教授,博士生导师,CCF 高级会员,主要研究领域为信息安全,分布计算.