一种面向程序动态分析的循环摘要生成方法

聂楚江[1],刘海峰[2],苏璞睿[1],冯登国[1]
(1.中国科学院软件研究所可信计算与信息保障实验室,北京 100190;2.北京信息安全测评中心,北京 100101)

摘　要： 动态测试数据生成方法相对于传统 Fuzz 测试方法能有效的提高软件测试与漏洞分析的效率.本文针对动态测试数据生成过程中对循环进行处理时的路径覆盖效率较低与约束求解困难的问题,提出了一种使用归纳变量构建循环摘要,并通过符号计算提取循环摘要的方法.本文通过将循环摘要应用于软件动态分析过程中,验证了使用循环摘要能有效的提高约束求解与循环路径遍历的效率.

关键词： 循环；软件测试；漏洞分析；符号计算
中图分类号： TP309　**文献标识码：** A　**文章编号：** 0372-2112 (2014) 06-1110-08
电子学报 URL： http://www.ejournal.org.cn　**DOI：** 10.3969/j.issn.0372-2112.2014.06.012

A Loop Summarization Method for Dynamic Binary Program Analysis

NIE Chu-jiang[1], LIU Hai-feng[2], SU Pu-rui[1], FENG Deng-guo[1]
(1. *Laboratory of Trusted Computing and Information Assurance, Institute of Software Chinese Academy of Science, Beijing* 100190, *China*;
2. *Beijing Information Technology Security Evaluation Center, Beijing* 100101, *China*)

Abstract: Compared to the traditional Fuzz testing, dynamic test generation can improve the efficiency of software testing and vulnerability analysis. This paper focuses on the performance of path covering and constraint solving in dynamic test generation, and proposes a method which constructs loop summary with induce variables and abstracts loop summary by symbolic computation. This paper uses the loop summary in software dynamic analysis, and proves that the loop summary can improve the efficiency of loop constraint solving and loop path traversal.

Key words: loop; software testing; vulnerability analysis; symbol computation

1 引言

动态测试数据生成与传统的 Fuzz 方法[1]比较,采用有针对性的方法生成测试数据,提高了测试数据生成的效率[2].动态测试数据生成方法采用符号计算在执行过程中遇到的状态条件与输入相关的约束表达式,并通过适当改变约束条件得到其它路径的约束条件;然后利用可满足性求解器求解这些约束条件,得到进入其它执行路径的测试数据输入.动态测试数据生成的方法结合了符号执行的全面性与 Fuzz 测试的适应性,能够以较少的测试数据覆盖较多的路径[3].尽管动态测试数据生成提高了软件测试与程序分析的效率,但还难以解决动态分析中路径求解困难与路径爆炸的问题.

引起动态测试数据生成方法中路径求解困难以及路径爆炸问题的一个重要原因是动态分析中程序的循环将被展开,即在动态分析中程序执行路径因为循环执行次数较多而长度急剧增大,并且将出现大量只是某个循环执行的次数不同的路径.针对循环引起的路径求解困难与路径爆炸问题,Dawn Song[4],Godefroid[5]等人分别针对软件动态分析过程中的循环处理从动态分析的角度提出各自的解决方案,但是都存在不能对循环进行全面分析的局限性.为提高动态测试数据生成的效率,论文提出了一种在静态分析过程中将程序中的循环单独提取出来,利用符号计算进行处理,并形成循环摘要的方法.论文提出的循环摘要应用于动态分析过程中可以有效的简化动态测试数据生成的过程.与本研究相似的有函数摘要(function summary)[2,6]方法,相关的研究有组合符号执行[3,6]、RWset 方法[7]、动静态程序分析方法[8]、DART[9]、CUTE[10]工具等.

收稿日期:2013-03-10;修回日期:2013-07-15;责任编辑:覃怀银
基金项目:国家自然科学基金项目“基于动态污点传播的恶意软件逆向分析研究”(No.61073179);国家重点基础研究发展计划(973计划)课题“未来互联网安全与可信机理”(No.2012CB315804);北京市自然科学基金项目“基于虚拟化技术的智能手机恶意软件检测方法研究”(No.4122086)

本文在对循环进行分析后发现可以使用在循环过程中发生变化的变量描述循环执行对程序状态的改变，这些变量被称为归纳变量；同时可以根据条件分支语句与这些变量的关系确定循环输入对执行过程的影响．因此，本文提出了一种通过归纳变量描述循环行为特征的方法，并将此特征称为循环摘要．我们建立循环摘要的基本思路是通过分析循环执行过程中的状态改变，得到循环中归纳变量与循环次数的函数关系，并结合循环执行过程的约束条件得到循环次数与输入的依赖关系．当这种函数关系是线性或是近似线性的情况时，可以用这种函数关系代替循环，从而简化程序分析过程[4,5]．

2　基于循环摘要的程序分析方法

针对程序动态分析过程中对循环处理困难的问题，我们提出了一种结合循环静态分析的扩展程序动态分析方法．其主要步骤包含把可执行程序转换为中间语言、生成程序控制流图、循环识别、循环摘要提取、循环摘要使用等，其主要的功能模块以及功能流程如图 1 所示．我们在本章中主要介绍基于循环摘要的程序分析方法的总体框架，各个过程的实施细节将在第 3 章中展开．

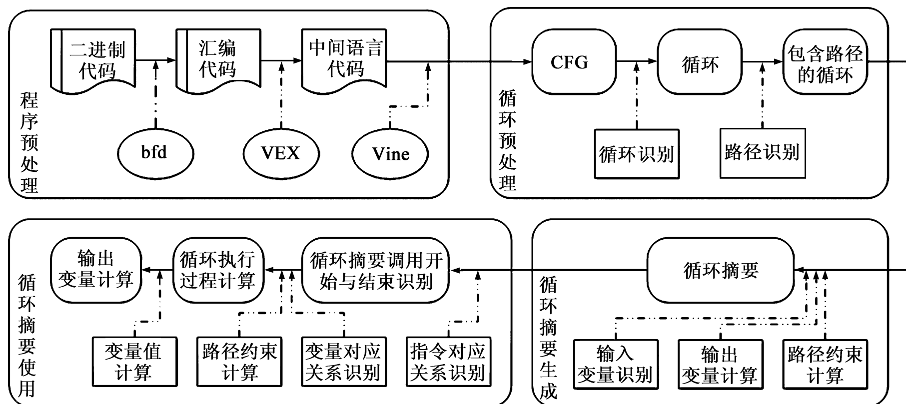


图1　基于循环摘要的程序分析方法

2.1　预处理过程

我们针对程序的二进制代码进行分析，所以首先要对程序进行反汇编，得到汇编语言表示的程序．考虑到汇编语言和计算机架构相关，并且某些汇编指令会隐式的改变一些特定寄存器的值，我们需要使用一种中间语言对其进行统一表示，使得在此基础上对程序进行的分析过程与指令集无关．本文的其它研究工作都在中间语言的基础上进行．

我们把对程序状态空间或程序执行流产生影响的过程称为语句，记为 statement，而把抽象的运算称为表示，记为 expression．程序中的 statement 一般包含赋值、跳转、调用与返回 4 种情况，而 expression 包含的情况则较多，主要可以分为运算、常数、变量取值、地址取值以及类型转换等种类．我们采用 statement block 表示一条汇编指令对应的多条语句．此外，我们使用的中间语言使用静态单一赋值 SSA(Static Single Assignment)[11]的方法对中间过程变量进行区分，并采用显式的跳转替代隐式的执行流．

我们用 CFG(Control Flow Graph)的节点表示 statement block，用两个节点间的有向边连接表示两个节点之间存在程序执行时的直接先后顺序．我们把不包含函数调用与循环的一段路径称之为简单路径，简单路径可以表示为一个指令序列．我们把类型为 a 的序列表示为 a list，于是 statement block 可以表示为 statement list，而简单路径 spath 可以表示为 statement block list．

对于循环与函数，可以将其视作一个整体，作为路径中的一个元素．我们定义自然循环的类型 loop，定义函数为类型 func，于是我们可以得到复合路径 cpath = spath | loop | func list．于是自然循环可以表示为{header, loop paths, exit paths, back edges, exit edges}，其中 header 表示循环的入口，循环路径 loop paths 表示一条从入口到回边 back edges 的 cpath，结束路径 exit paths 表示一条从入口到出边 exit edges 的 cpath．由此我们可以得到一个循环的执行过程是一个 loop paths list 与 exit paths 的组合．

最后，我们使用支配树识别函数与循环，并依据函数与循环之间的包含关系提取函数与循环包含的复合路径．同时为了从函数与循环中提取出复合路径，我们实现了一种多叉树的非递归遍历方法．

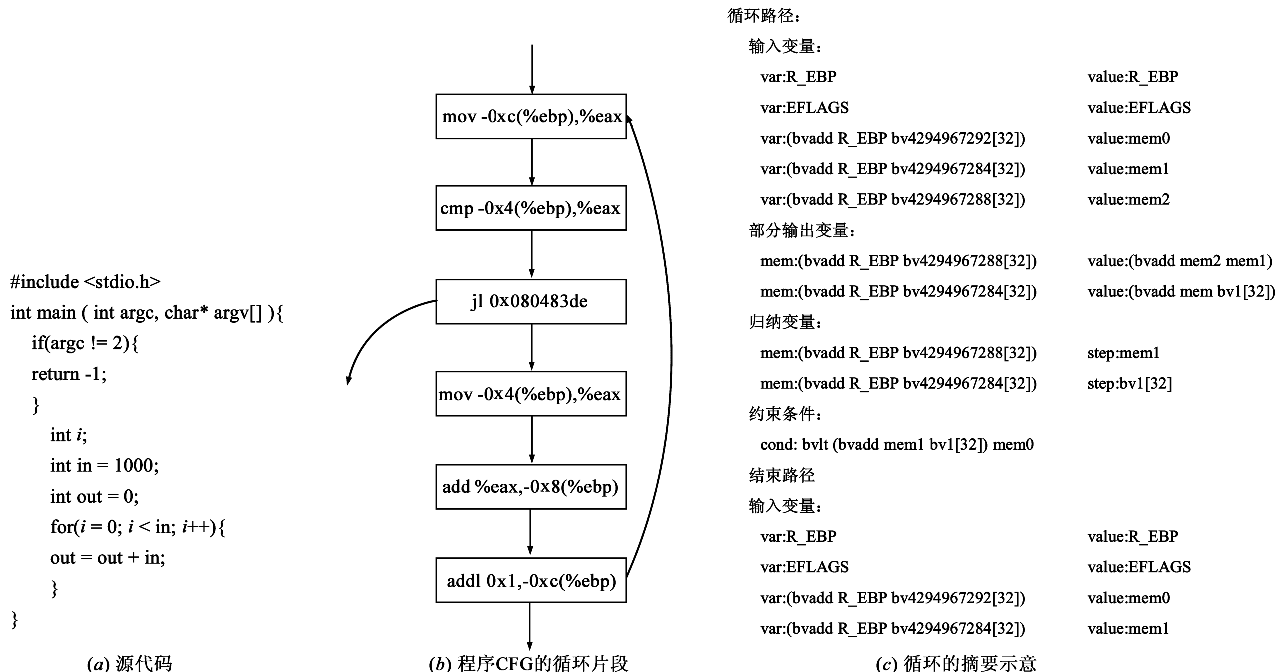
2.2　循环摘要

我们首先描述简单循环摘要，简单循环摘要可以描述循环路径与结束路径中不包含循环或函数的情况下的循环的行为．我们采用循环路径与结束路径摘要的集合表示循环摘要．对于简单路径 spath，我们定义其

摘要 summary = {cfg, $vars_{in}$, $vars_{out}$, $mems_{in}$, $mems_{out}$, constrains},其中 cfg 表示当前的 CFG 数据结构,$vars_{in}$表示寄存器形式的输入变量集合,$vars_{out}$表示寄存器形式的输出变量集合,$mems_{in}$表示内存地址形式的输入变量集合,$mems_{out}$表示内存地址形式的输出变量集合,而 constrains 表示当前路径的约束条件.

为了与程序动态分析相结合,我们采用的循环摘要以约束求解器所支持的变量类型与表达式表示.我们用 Z3 中的抽象语法树节点表示摘要中的变量:首先定义一个一种字节数组类型,然后以这种类型定义 $vars_{in}$中的变量;然后针对内存地址形式表示的变量,定义相应的变量 $mems_{in}$;最后用 $vars_{in}$与 $mems_{in}$的表达式表示 $vars_{out}$与 $mems_{out}$.在只考虑简单路径的情况下,循环路径在程序执行的过程中不断的循环,直到进入一条结束路径或另一条循环路径.在循环执行过程中,往往存在一些变量随着循环次数的增加而逐渐增加或减少,而正是由于这些变量的变化使得循环当前路径约束不再满足,从而进入结束路径或另一条循环路径.由于只有归纳变量才会影响路径分支条件是否被满足,因此我们把归纳变量单独提取出来作为循环摘要的重要组成部分.于是循环摘要可以表示为{loop,($summary_{spath}$, $vars_{induce}$) list}.对于复合路径,其摘要表示为简单路径 spath、循环 loop 以及函数 func 的摘要的集合.在定义了复合路径摘要的基础上,我们可以对一般循环给出循环摘要的定义{loop,($summary_{cpath}$, $vars_{induce}$) list}.

在 $summary_{cpath}$中记录了路径中所有的变量,以及这些变量在路径起始点与结束点的符号值.而归纳变量表示程序在此路径上发生改变的输入变量,通过归纳变量我们可以描述循环在执行完一次后状态的变化,通过归纳变量与路径约束我们可以计算此路径连续执行的次数.以图 2 中的简单循环为例,图 2(*a*)是源代码,图 2(*b*)是由其二进制代码生成的 CFG 的循环部分,图 2(*c*)是此循环的部分输入输出变量、归纳变量以及路径约束的 Z3 表达式.图 2(*c*)中 bv1[32]表示一个 32 位的常数,其值为 1;bvadd 表示一个加法操作,bvlt 表示小于的逻辑运算,mem0 表示一个符号值.考虑到机器指令中的移位等操作,我们把程序中的所有数据都视为比特向量,'bv'就表示运算的作用域是比特向量.图 2(*c*)中输入输出变量可以是内存地址与寄存器,输入变量包含所有在取值前没有被赋值的变量,输出变量包含所有在此路径上被赋值的变量,归纳变量是在此路径执行过程中取值发生变化的输入变量,约束条件是此路径执行时必须满足的条件.



(*a*) 源代码　(*b*) 程序CFG的循环片段　(*c*) 循环的摘要示意

图2 一个简单循环的摘要示意图

2.3 循环摘要的使用

在定义了循环摘要的基础上,我们对其在动态分析中的使用方法进行探讨.首先,需要判断当前执行指令是否是循环的入口.其次,需要判断路径分析中的变量与摘要中的变量的对应关系.然后,我们需要将循环摘要中输出变量的表达式转换为相对于输入变量地址形式的变量表示的表达式.最后,在程序执行路径中找到循环出口的位置,结束此次循环摘要使用过程.

动态测试数据生成方法使用循环摘要的情况主要有两种.第一种方法将循环在某一预期执行过程中的输入输出关系作为符号表达式应用于路径约束求解中.令循环的执行次数为 N,符号执行的消耗为 C_e,路径求解的消耗为 C_r,摘要提取的消耗为 C_s,那么对于线性循环路径,我们得到使用摘要前后动态分析的近似消耗比为$(C_e+C_r+C_s)/N\cdot(C_e+C_r)$.因此,对于循环次数较多的循环,使用循环摘要将有显著优势.并且在第 5 章中我们发现循环中线性循环路径的比例很高.

第二种方法则是通过构建一些有代表性的循环执行预期,对循环所有可能执行路径进行近似遍历.令到达此循环的可能路径数量为 M_p,以循环出口开始的可能路径数量为 M_s,循环的最大可能执行次数为 N_m,我们可以通过循环执行预期的方法采用 $s\cdot M_p\cdot M_s$ 条路径,近似覆盖 $N_m\cdot M_p\cdot M_s$ 条可能的路径,其中 s 是循环执行预期的数量.因此,循环摘要为有效的解决动态测试数据生成过程中的路径遍历问题提出了一种可行的方法.

3　循环摘要产生的方法

在图 2 中,我们对一个简单循环的摘要进行了示意,下面我们将详细描述循环摘要的生成过程.由于图 2 中的示例比较简单,为了使我们的方法有较好的适用性,在循环摘要的生成过程中,需要对循环中包含多条循环路径与结束路径、路径中包含多个条件分支以及循环嵌套的情况进行处理.

3.1　复合路径的处理

在以上的工作中我们已经把循环转换为其包含的复合路径以及路径对应的约束条件,下面将对复合路径进行处理,得到复合路径执行的条件以及执行后对程序状态的改变.我们对循环体的处理仅限于两类指令,一种是对内存或寄存器的赋值语句,一种是条件跳转语句.因为只有对内存或寄存器进行赋值的语句才会在每次循环过程中改变程序的状态,只有条件跳转语句才能改变循环执行过程中的操作.我们生成循环摘要的方法主要是基于静态分析.

我们将循环变量符号值的集合记作 S,$S[m]$表示 m 变量的值,←表示赋值操作,$S+(v,m)$表示在状态 S 中增加了一个表示变量的符号 v,其符号值为 m.下面我们将针对循环中的每一条复合路径进行分析,分别遍历其中的复合节点.如果此节点是简单路径,则采用 treatSpath 函数进行下一步的处理,如果是子循环则对子循环进行处理.对于函数调用的情况,如果函数在分析的静态程序样本中,则将其视作普通跳转.如果是库函数或系统调用的情况,我们将在动态分析部分进行处理.如果复合路径中不含子循环或系统调用,我们可以使用 merge_summary 函数得到复合路径输入输出变量之间的关系 $f_{cpath}:S_{entry}\rightarrow S_{exit}$.图 3 描述了循环处理过程.

```
Input ctx:          Z3 context
Input cfg:          program control flow graph
Input loop:         the loop to be analyze
loopAnalyzer(ctx, cfg, loop){
    for complex_path in cycle pathes and exit pathes
      solver = create_solver(ctx);
      path.condition_jump = {}
      for node in complex_path.nodes
        switch node
                  case   simple_path:
                         (S_entry, S_exit, solver) = treatSpath(ctx, cfg, simplePath, S_entry, S_exit, solver);
                         node_summary.add(S_entry, S_exit, solver)
                  case   loop:
                         loopAnalyzer(cfg, path);
                  case   fun:
                         ignore;
        complex_path.summarys.add(node_summary);
      endfor
      if all node in complex_path.nodes is simple_path then
        merge_summary(complex_path.summarys);
      endif
}
```

图 3　循环处理过程

我们在处理赋值语句时,如果复合路径中的符号变量在使用前没有被赋值,则将其标识为输入变量,最后可以得到复合路径输出变量值与输入变量值的关系.同时,我们在处理条件分支语句时,得到复合路径在当前分支被满足的约束条件,将复合路径全部的约束条件组合在一起就得到在程序进入此复合路径的条件.至此,在对程序进行动态分析时可以根据路径输入变量的真实取值计算出路径执行结束后程序的状态.图 4 描述了简单路径处理过程.

```
Input ctx:                  Z3 context;
Input cfg:                  program control flow graph
Input simplePath:           the loop to be analyze
Input/Output S_entry:       entry status
Input/Output S_exit:        exit status
Input/Output solver:        the constrain in Z3 style
treateSpath (ctx, cfg, simplePath, S_entry, S_exit, solver){
  for inst in path
      for statement in inst
          switch statement:
             case m_t ← exp:
                 if exp contain m′ ∧ m′ ∉ S_exit[m] then
                     S_entry + (m′, m′);
                     S_exit + (m′, m′);
                   endif
                   expZ3 = translateToZ3(exp);
                   S_exit + (m, expZ3);
             case conditionjump(exp, target1, target2):
                   if the next inst is target1 then
                     solver.add(translateToZ3(exp));
                   else
                     solver.add(not translateToZ3(exp));
                   endif
      end for
  endfor
}
```

图 4　简单路径处理过程

3.2　循环摘要的提取

在得到了循环路径的 S_{entry}与 S_{exit}之后,我们可以识别出此路径包含的归纳变量,同时某些归纳变量会出现在循环路径的约束条件中,当这些变量的取值变化时,循环的约束条件可能会变成不再被满足,然后循环将进入结束路径或其它循环路径.我们通过提取归纳变量以及它与条件分支语句之间的关系来对循环中路径的 f_{cpath}进行一定的近似,采用这种近似过程,可以方便的从循环输入判断循环执行的次数,并计算出利用循环执行次数与输入变量表示的循环输出状态.图 5 描述了归纳变量的生成方法以及循环执行次数的计算过程.

在提取路径约束条件时,需要对指针进行针对性的处理,我们在 $S[m]$的基础上进行了扩展,增加了 $S[[m]]$的语义.对于指针变量,我们分析其地址值的变化,同时记录指针所指向地址对应变量的符号值的变化.最后记录通过指针变量访问其它变量时的基址 $S[m]$,以及地址的变化 dp,并通过引入变量增量模式描述这种输入输出变量不确定的情况.通过扩展的 $S[[m]]$,我们可以描述字符串以及数组变量变化的情况,并以此确定此约束条件决定的循环次数.

在对循环路径中的变量进行处理后,我们采用 LEXP◇REXP 表示循环中的条件分支,其中$◇\in\{<,>,\leqslant,\geqslant,=,\neq\}$.我们采用 v_{cond} = LEXP-REXP 表示约束条件的状态,并使用 dv_{cond}表示约束条件的改变,两者相除向上去整就得此约束条件所决定的循环的执行次数,它是一个与输入相关的符号值.对于约束条件中包含指针的情况,我们重点分析当指针指向的字符串或数组是输入数据的情况,并采用 iter_assert 函数表示在当前执行过程中,字符串或数组需要满足的条件,其形式一般是前 $n-1$ 个变量需要满足的约束,以及第 n 个变量需要满足的约束.

```
Input: loop            the loop to be analyzed
Output: summary        the summary of loop
Summary getSummary(Loop loop){
    for all path in loop.loop_pathes with index i
        for all S_entry[m] in loop.S_entry
            if S[m] is not an address
                dS[m] = S_entry[m] - S_exit[m];
                if dS[m] != 0 then
                    path.rv.add(S_entry[m]);
                    path.rv[m].sValue.add(dS[m]);
                    S'[m] = S_entry[m] + dS[m] * satisfiedCount[i];
                end if
            else
                dp = S_entry[m] - S_exit[m];
                dS[m] = S'[[m]] - S[[m]];
                if dp != 0 then
                    path.rv.add(S_entry[m]);
                    path.rv[m]. pAddress.add(dp);
                    S'[m] = S[m] + dp * satisfiedCount[i];
                    path.ra.add (S[m],dp,dS);
                endif
        end for
        for cond in loop.condition_jump and cond in path
            path.conds.add(cond)
            match cond.exp with LEXP ◇ REXP;
            if v_cond does not contain S[m] in path.ra, and dp ≠ 0 then
                v_cond = LEXP - REXP;
                dv_cond = v_cond[satisfiedCount[i] ← 1] - v_cond[satisfiedCount[i] ← 0];
                count[cond.pc] = ⌈v_cond / dv_cond⌉;
            else
                count[cond.pc] = iter_assert(cond.exp);
        end for
        path.count = count; i++;   pathes.remove(path);
    end for
    summary = pathes;
}
```

图 5　从归纳变量与条件分支生成循环摘要

本文生成的循环摘要能够对循环路径约束只包含线性归纳变量的循环进行准确的处理,对于循环路径约束与非线性归纳变量相关的循环,可以以迭代的方式进行精确处理或者以近似线性的方式进行近似处理.我们使用简单路径占全部循环路径的比例评价循环摘要的适用性,以及使用线性归纳变量占全部归纳变量的比例评价循环摘要的准确性.

4　具体实现

我们采用 GNU bfd 对程序进行反汇编,并利用 Valgrind[12]的 VEX 转换为中间语言,在此基础上采用 BitBlaze[13]中的 Vine 模块生成程序 CFG.在获得 CFG 的基础上,我们采用 OCaml 语言进行进一步的分析,并使用 ocamlgraph 对 CFG 进行分析.我们采用 OCaml 构建了整个系统的框架,并在此基础上进行循环的提取、循环路径识别、路径处理等工作.

我们采用的可满足性求解器是 Z3.我们得到的程序的中间语言表示包含所有的变量定义,根据这些定义创建一个上下文,并对所有的中间语言语句表示建立了映射到相应的符号表达式的规则.通过这些规则,我们得到了循环中各条路径上的变量与约束条件的符号表示.我们为循环中每条路径创建了相应的输入变量符号表、输出变量符号表以及分支条件表.此外为了支持对内存地址以及指针的处理,定义了内存地址数据结构,并在上面定义了一种相等关系,即如果通过符

号求解证明两个地址是恒等的,则这两个地址是同一个地址.在实际的程序分析过程中,我们发现这种处理方法能够处理大部分的情况.

5 实验与验证

我们的实验环境是 Ubuntu 12.04.1,Ocaml 版本为 3.12.1,Gcc 版本为 4.6.3.我们分别对 Linux 与 Windows 下各 3 个程序或动态库进行了实验,并对其中的循环计算了相应的摘要.我们对循环的嵌套关系,以及循环中的路径进行了分析,发现程序中的循环存在如下特点:(1)如图 6 所示,程序中的循环主要是简单循环,即对大部分循环进行分析不需要处理循环嵌套的情况;(2)如图 7 所示,程序中循环所包含的路径一般不超过 10 条,说明在大部分情况下,循环摘要的表示不会过于复杂;(3)如图 7 所示,程序中循环包含的路径大部分都是简单路径,不包含子循环或函数调用,说明循环摘要在大部分情况下不需要考虑与其它循环摘要或函数摘要的关联关系.

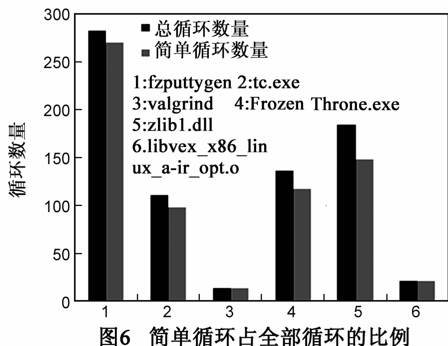


图6 简单循环占全部循环的比例

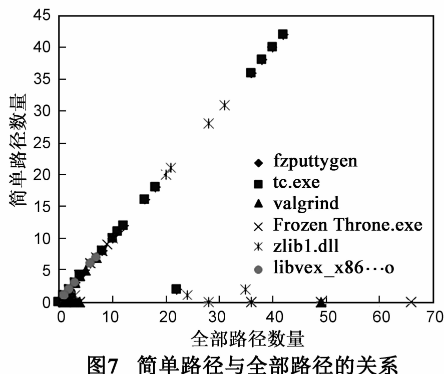


图7 简单路径与全部路径的关系

在对生成的循环摘要进行分析时,我们发现这些变量中存在大量在一次循环路径执行过程中值变化为常数的线性归纳变量与线性归纳地址,这说明我们可以很方便的计算出输入变量或地址与循环执行的次数之间的函数关系.我们从表 1 中发现其中有 4 个程序的线性归纳变量数目接近输入变量与输入地址数量,因此可以认为很大部分的循环路径可以用线性函数表示,结合图 7 的数据,可以认为部分循环有较好的线性特征.

表 1 路径分析数据

程序	输入变量	输出变量	输入地址	输出地址	归纳变量	归纳地址	线性归纳变量	线性归纳地址
fzputtygen	2834	618	7123	4830	1167	1761	830	119
tc.exe	578	103	209	128	311	557	174	16
valgrind	48	11	17	3	17	34	12	1
Frozen Throne.exe	871	144	266	143	526	901	234	15
zlib1.dll	2785	334	3008	956	1715	2346	587	239
ibvex_x86_linux_a-ir_opt.o	154	34	80	28	57	86	40	0

最后我们对循环摘要生成过程中的时间与内存消耗并结合影响循环分析的主要因素进行分析.图 8 表明,我们的方法在平常性能的环境中对于一定规模的程序能在合适的范围内完成循环摘要的提取,说明该方法的时间复杂度与空间复杂度被控制在较好的范围内.

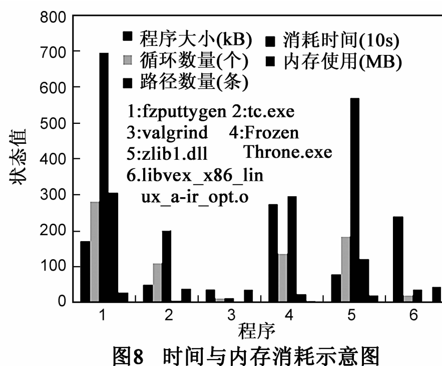


图8 时间与内存消耗示意图

对于循环来说,执行次数越多,其约束求解代价也越高.而采用循环摘要在动态分析过程中对循环进行替换,其主要的开销包括程序反汇编、转换为中间语言、CFG 图的构造、循环识别、路径处理、符号执行等,此外为了和已有的动态测试数据生成方法结合,还需要额外的循环入口识别、循环变量识别等操作带来的时

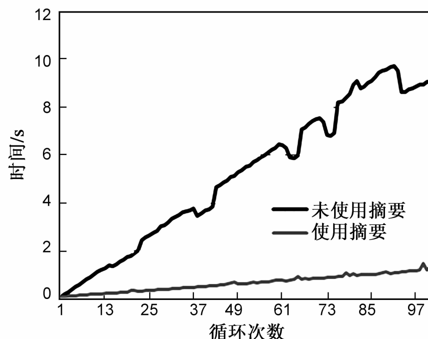


图9 使用循环摘要前后时间消耗的比较

间消耗.我们对图 2 中的程序进行简单修改，让变量 v_{in} 从输入获得，然后分别对 1 到 100 此循环的情况进行分析，对不使用循环摘要的路径求解代价与使用循环摘要的代价进行比较，具体的性能数据如图 9 所示.

6　工作总结

我们的工作主要针对动态测试数据生成过程中由程序执行路径包含循环引起的路径爆炸问题以及路径约束求解困难的问题进行优化.目前动态测试数据生成方法主要采用 flip 的方法遍历路径，这种方法在对程序中普通的分支进行处理时不会出现效率问题，但是它在处理循环这种高度重复的程序结构时也只能逐渐改变循环执行的次数或循环执行过程，因此需要分析大量的程序可能执行路径.同时，当采用 flip 方法遍历到循环执行次数较多的执行路径时，路径约束求解的代价也将变得很高.我们采用的改进方法是首先设计一个循环的执行预期，该执行预期覆盖循环各种典型的执行情况；然后通过循环摘要生成此预期下的循环输入输出变量之间的函数关系，并应用在路径约束求解过程中.这种方法能有效提高循环执行路径约束求解以及循环路径遍历的效率.

参考文献

[1] Takanen A，Demott J D，Miller C.Fuzzing for Software Security Testing and Quality Assurance[M].Norwood：Artech House on Demand，2008.22 – 32.

[2] Godefroid P，Levin M Y，Molnar D.Automated whitebox fuzz testing[A].Network & Distributed System Security Symposium [C].San Diego，California，USA：Internet Society，2008.151 – 166.

[3] Anand S，Godefroid P，Tillmann N.Demand-driven compositional symbolic execution[A].14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems[C].Berlin，German：Springer，2008.367 – 381.

[4] Saxena P，Poosankam P，Mccamant S，et al.Loop-extended symbolic execution on binary programs[A].Proceedings of the Eighteenth International Symposium on Software Testing and Analysis[C].New York，USA：ACM，2009.225 – 236.

[5] Godefroid P，Luchaup D.Automatic partial loop summarization in dynamic test generation[A].Proceedings of the 2011 International Symposium on Software Testing and Analysis[C].New York，USA：ACM，2011.23 – 33.

[6] Godefroid P，Nori A V，Rajamani S K，et al.Compositional may-must program analysis：unleashing the power of alternation [A].ACM Sigplan Notices[C].New York，USA：ACM，2010.43 – 56.

[7] Boonstoppel P，Cadar C，Engler D.RWset：Attacking path explosion in constraint-based test generation[A].Tools and Algorithms for the Construction and Analysis of Systems[C].Berlin，German：Springer and Heidelberg GmbH & Co K，2008.351 – 366.

[8] 陈平，韩浩，沈晓斌，等.基于动静态程序分析的整形漏洞检测工具[J].电子学报，2010，38(8)：1741 – 1747.
Chen Ping，Han Hao，Shen Xiao-bing，et al.Detecting integer bugs based on static and dynamic program analysis[J].Acta Electronica Sinica，2010，38(8)：1741 – 1747.(in Chinese)

[9] Godefroid P，Klarlund N，Sen K.DART：directed automated random testing[A].ACM Sigplan Notices[C].New York，USA：ACM，2005.213 – 223.

[10] Sen K，Agha G.Cute and jCUTE：Concolic unit testing and explicit path model-checking tools[A].Computer Aided Verification[C].Berlin，German：Springer，2006.419 – 423.

[11] Pop S.The SSA Representation Framework：Semantics，Analyses and GCC Implementation[D].Paris：école des Mines de Paris，2006.

[12] Nethercote N，Seward J.Valgrind：a framework for heavyweight dynamic binary instrumentation[J].ACM Sigplan Notices，2007，42(6)：89 – 100.

[13] Song D，Brumley D，Caballero J，et al.BitBlaze：A new approach to computer security via binary analysis[A].Proceedings of the 4th International Conference on Information Systems Security：Information Systems Security[C].Berlin，German：Springer，2008.1 – 25.

作者简介

聂楚江　男，1983 年生于湖南省隆回县，现为中国科学院软件研究所博士研究生，主要研究领域为网络与系统安全.
E-mail：niecj@is.iscas.ac.cn

刘海峰　男，1976 年出生于山东省泰安市，现为北京信息安全测评中心副研究员，主要研究领域为信息安全、电子政务.
E-mail：liuhf@bjeit.gov.cn