一种目标制导的混合执行测试方法

崔展齐 王林章 李宣东
(南京大学软件新技术国家重点实验室 南京 210093)
(南京大学计算机科学与技术系 南京 210093)

摘 要 混合执行测试(concolic testing)是一种将具体执行与符号执行相结合的自动化测试方法.由于混合执行测试从程序本身出发,未将目标缺陷的先验知识作为指导,会导致生成和执行大量不能发现缺陷的测试输入,从发现缺陷的角度看浪费了时间和计算资源开销.这个问题在具有时间、成本及资源约束的实际测试任务中更加突出.为解决这一问题,文中提出了一种结合静态分析和混合执行测试技术的目标制导的混合执行测试方法:使用静态分析工具分析待测程序中可能含有缺陷的可疑语句及其缺陷类型,并将静态分析所报告的可疑语句作为目标指导测试.目标制导的混合执行测试技术分为 3 个步骤:首先,计算从程序各分支到待检测缺陷语句的可达性;其次,对待测试程序进行插装以支持混合执行测试;第三,使用静态分析的结果和可达性信息作为指导,只生成和执行可能会覆盖待检测缺陷语句的测试输入,以避免生成和执行不能发现缺陷的测试输入.基于此方法,作者实现了一个测试缓冲区溢出缺陷的原型工具:TARGET,并在一组 C 语言基准程序上进行了对比实验.实验结果表明与原有的混合执行测试技术相比较,TARGET 能在更短的时间内发现程序中更多的缺陷.

关键词 目标制导测试;缺陷触发;静态分析;混合执行测试
中图法分类号 TP311 **DOI 号**: 10.3724/SP.J.1016.2011.00953

Target-Directed Concolic Testing

CUI Zhan-Qi WANG Lin-Zhang LI Xuan-Dong
(*State Key Laboratory of Novel Software Technology, Nanjing University, Nanjing* 210093)
(*Department of Computer Science and Technology, Nanjing University, Nanjing* 210093)

Abstract Concolic testing is an automatic testing technique which combines concrete execution and symbolic execution together. Concolic testing only focuses on programs under test, and lacks the prior knowledge of target faulty statements. From the perspective of detecting defects, time is wasted in generating and executing test inputs that cannot find defects. Furthermore, the limitation of time, budgets, and resources in practical test tasks make this problem even worse. To address this problem, this paper proposes a target-directed testing approach to combine static analysis with concolic testing techniques: the program under test is statically analyzed before testing to identify types and positions of suspicious defects; then, the program is tested with the guidance of the static analysis results. The target-directed testing technique is consisted by 3 steps: Firstly, calculate the reachability relationships from branches to the suspicious statements; Secondly, instrument the program under test for concolic testing; Thirdly, use static analysis information and reachability relationships to guide concolic testing, only generate test inputs can cover suspicious statements, in order to avoid generating test inputs that cannot detect defects. A prototype tool, TARGET, has been implemented based on the proposed technique to test buffer overflows. The authors have experimentally evaluated TARGET on a set of C bench-

收稿日期:2010-11-20;最终修改稿收到日期:2011-05-10.本课题得到国家自然科学基金(90818022,91018006,61021062)、国家"九七三"重点基础研究发展规划项目基金(2009CB320702)、国家"八六三"高技术研究发展计划项目基金(2011AA010103)和核高基项目(2009z01036-001-001-3)资助.**崔展齐**,男,1984 年生,博士研究生,主要研究方向为软件工程、软件测试. E-mail: zqcui@seg.nju.edu.cn. **王林章**(通信作者),男,1973 年生,博士,副教授,主要研究方向为软件工程、软件测试. E-mail: lzwang@nju.edu.cn. **李宣东**,男,1963 年生,博士,教授,博士生导师,主要研究领域为软件工程、形式化方法、软件建模与分析、软件测试与验证.

marks, and the results show that TARGET can find more defects with less time overhead than original concolic testing techniques.

Keywords target-directed testing; defects triggering; static analysis; concolic testing

1 引 言

为提高软件质量,软件测试和静态分析是两类使用最为广泛的缺陷检测技术[1-2].其中,软件测试是目前工业界使用最多的软件质量保障手段,而静态分析近年来开始逐渐得到工业界的重视和应用.静态分析技术能在软件生命周期的较早阶段发现程序缺陷[3-4].此外,静态分析能在很大程度上实现自动化,具有人力成本开销较小的优势.软件测试技术通过测试用例来触发和确认程序缺陷,能发现程序中的真实缺陷[5].同时,具体的测试用例还能够为软件工程师调试和排错提供有效帮助.但是,软件测试工具通常需要一个较为充分的测试用例集,这在大多数情况下是难以获取的.

混合执行测试[6-7]是一种有效的提高测试自动化程度的技术,其目标是通过自动生成测试输入来执行程序中的所有可行路径,从而发现程序缺陷.混合执行测试技术在待测试程序上执行具体输入的同时进行符号执行.首先通过插装的方式在程序执行过程中收集路径条件的符号表达式,然后通过约束求解器求解所收集到的路径条件,以生成执行新路径的测试输入.当不能求解所收集到的路径条件时,混合执行测试使用具体的值来简化符号表达式.混合执行测试在一定程度上解决了缺乏测试用例集的问题,但由于该方法从程序本身出发,未将目标缺陷的先验知识作为指导,导致生成和执行了大量不能覆盖缺陷语句的测试输入,从发现缺陷的角度看,浪费了时间和计算资源开销.

由于只有能覆盖缺陷所在语句的测试输入才有可能触发缺陷,不能覆盖缺陷语句的测试输入对发现缺陷是毫无贡献的.如能获取程序中潜在缺陷的类型和位置,就能在不损害缺陷发现能力的前提下,通过避免生成和执行不能覆盖缺陷语句的测试输入来降低测试开销.而一些静态分析工具,则能报告程序中潜在缺陷语句的位置和缺陷类型信息,但由于静态分析在分析过程中使用了一些保守的假设,因而具有较高的误报率.从含有大量误报的缺陷警报中识别出真正的缺陷仍需要大量的人力和时间开销.

软件测试和静态分析在很大程度上具有互补性,如果能有效地利用静态分析的结果来指导软件测试,将带来两方面的好处:一方面,静态分析工具所提供的信息能为软件测试提供测试的目标,从而降低软件测试的开销;另一方面,软件测试提供的运行时信息能用于确认和精化静态分析工具所报告的缺陷警报.

基于上述思路,本文提出了一种静动态结合的目标制导的混合执行测试方法:使用静态分析工具分析待测程序,检测程序中可能会含有缺陷的可疑语句及其缺陷类型,并将静态分析所报告的可疑语句作为待检测缺陷语句,指导测试输入的生成和执行.目标制导的混合执行测试方法分为 3 个步骤:首先,使用程序控制流程图分别计算从程序各分支语句到待检测缺陷语句的可达性;其次,对待测试程序进行插装以支持混合执行测试;第三,使用静态分析的结果和可达性信息作为指导,只生成和执行可能会覆盖待检测缺陷语句的测试输入,以避免生成不能发现缺陷的测试输入.我们在一组 C 语言基准程序上就缓冲区溢出缺陷进行了对比实验.实验结果表明,与原有的混合执行测试技术相比较,本方法能在更短的时间内发现更多的程序缺陷.

本文的主要贡献在于:

(1)提出了一种结合静态分析和混合执行测试的缺陷检测方法,使用静态分析报告的潜在缺陷的位置和类型信息指导混合执行测试,从而降低测试开销;

(2)基于该方法实现了一个用于检测 C 语言中缓冲区溢出的原型测试工具,并在一组 C 语言基准程序上进行了实例研究以评估本方法的有效性.

本文第 2 节给出混合执行测试和静态分析的相关背景知识;第 3 节通过一个例子来引出使用静态分析结果作为目标指导测试的动因;第 4 节详细描述目标制导的混合执行测试技术;第 5 节介绍原型工具的实现并进行实验和评估;第 6 节介绍相关的工作;第 7 节总结本文的工作并进行展望.

2 背景介绍

2.1 混合执行测试

混合执行测试[6-7]是一种将具体执行与符号执

行相结合的自动化测试方法，其目标是通过生成测试输入来执行程序中的所有可行路径，以发现程序缺陷. 其中，concolic 是将具体(CONCrete)和符号(symbOLIC)两个词结合在一起所构成的一个新词. 具体执行是以具体的输入值执行程序；符号执行使用变量符号代替具体的输入值，模拟执行程序，得到程序各路径条件的符号化表达式，然后对符号表达式进行约束求解，以得到能够执行指定程序路径的具体输入值. 而混合执行测试则是将具体执行与符号执行相结合的测试技术，其思想是在具体执行过程中收集路径条件的符号表达式，并按照深度优先的策略将路径条件逐一取反，然后通过约束求解的方式求解取反后的路径条件，以获得执行新路径的测试输入. 当不能求解路径条件时，使用具体的值来代替符号表达式，以简化路径条件.

在混合执行测试技术的基础上，有一些相关的扩展工作. Sen 等提出的 CUTE[7] 将混合执行测试技术扩展到支持含有指针和数组等复杂数据结构的程序，JCUTE[8] 将混合执行测试技术用于自动化测试多线程程序的竞争问题. 为解决混合执行测试中分支覆盖率较低和可扩展性不强的问题，CREST[9] 提出了几种基于程序静态控制流图的路径搜索策略，尽量生成执行尚未覆盖分支的测试输入. SPLAT[10] 则将缓冲区的一段前缀和缓冲区的长度进行符号化建模，用于检测程序中的缓冲区溢出缺陷.

2.2　静态分析

静态分析是指在不运行软件的前提下对软件的各种性质进行分析的过程，其对象可以是设计模型、源程序、字节码等软件生命周期中的各种产物[2]. 静态代码分析技术最常见的用途是检查源程序中的缺陷，可以按是否流敏感、是否路径敏感、是否支持过程间分析等维度进行分类[4].

静态分析技术的优势在于自动化程度高、可扩展性好，能处理规模较大的程序，其缺点在于精度不高，可能会出现误报(false positive)和漏报(false negative)的情况. 误报是指分析工具所报告的缺陷实际上并不存在，而漏报则是指分析工具未能检测出程序中存在的某些缺陷. 另外，如何从静态分析的结果中识别出真实的缺陷，并设计测试用例以确认该缺陷仍然具有很大的挑战性. Splint[11]、ARCHER[12]、BOON[13] 等静态分析工具能够报告程序中缺陷语句或变量的位置. Prefix[14]、PREfast①、Marple[15] 等静态分析工具对识别出的缺陷还提供了相应的路径信息.

3　示　例

图 1 是将 WuFTP-1[16] 中的一个缺陷进行简化后的示例程序，其中 *path* 是一个输入字符串，Example 函数的功能是将 *path* 中字符串拷贝到数组 *mapped_path* 中，若 *path* 的第 1 位不为‘/’，则需要在前面加上‘/’后再进行拷贝. 如该程序的控制流程图所示，该程序共包含 4 条路径，第 8、9 行修改了 *mapped_path* 中的内容，是可能引发缓冲区溢出错误的语句，其中当 *path* 的第一位不为‘/’且长度为 9 时，将会在第 9 行触发缓冲区溢出错误.

```
1  void Example (char *path){
2     char mapped_path [10];
3     if(strlen(path)==0)
4        return;
5     if(strlen(path)>=10)
6        return;
7     if(path[0] != '/')
8        strcpy(mapped_path, '/');
9     strcat(mapped_path, path);
10 }
```

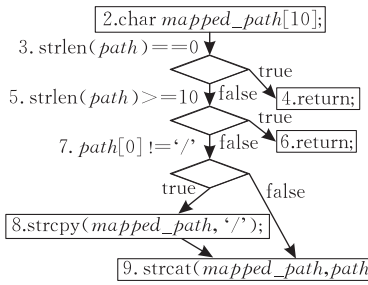


图 1　示例程序及对应的控制流程图

使用支持缓冲区建模的混合执行测试工具，如 SPLAT[10]，将会尝试覆盖程序中的所有路径，并在每次修改缓冲区时检查是否可能溢出. 第一，将空字符串作为 *path* 的默认输入并运行程序，覆盖语句 2，3(T)，4(T，F 分别表示条件判断语句的 true 和 false 分支)；第二，转置 3(T)所对应的路径条件以生成非空字符串作为下一次程序运行的输入，假设为“a”，该测试输入将覆盖语句 2，3(F)，5(F)，7(T)，8，9，并分别在第 8、9 行检查缓冲区是否可能会溢出，结果为第 8 行在当前路径条件下不能溢出，第 9 行可能会溢出，生成长度为 9 且第 1 位不为‘/’的字符串作为测试输入并执行，以确认该缺陷；第三，转置 7(T)所对应的路径条件，生成第 1 位内容为‘/’的测试输入，假设为‘/’，覆盖语句 2，3(F)，

① https://www.microsoft.com/china/whdc/devtools/tools/prefast.mspx

5(F),7(F),9；第四，转置 5(F)所对应的路径条件，生成长度为 10 的字符串，覆盖语句 2,3(F),5(T),6，测试结束. 在测试过程中共生成了 5 个测试用例以覆盖所有程序路径和确认缺陷，并对可疑语句检查了 2 次.

若使用 Marple 进行静态分析，将报告第 8 行是安全的缓冲区修改操作，而第 9 行则是存在缺陷的. 但第 9 行的缺陷是否为真实的缺陷以及将会如何被触发仍是未知的.

观察上述结果发现，程序中可能存在缺陷语句的可疑程度是不同的，使用静态分析对待测试程序进行分析可排除部分不需要动态检测的安全语句. 而程序中也只有部分分支能够到达需要动态检查的可疑语句，如分支 3(T)、5(T)均不能到达第 9 行. 运行不能覆盖待检测语句的测试输入，如路径 2,3(T),4，不能对检测程序缺陷做出贡献. 因此，若能有效地使用静态分析的结果指导测试，并不需要覆盖程序的所有路径来检测程序中所包含的缺陷. 在测试过程中，若能避免生成和执行不能覆盖待检测缺陷语句的测试输入，将会降低测试开销.

4 目标制导的混合执行测试

如图 2 所示，本文提出的目标制导的混合执行测试方法首先使用静态分析工具对待测试程序进行分析，获取程序中可能存在缺陷的可疑语句的类型及其所在位置，并将可疑语句作为待检测缺陷语句指导混合执行测试. 目标制导的混合执行测试分为 3 个步骤：可达性分析、程序插装、动态测试. 首先，计算程序中各分支语句与静态分析所报告的待检测语句间的可达性；其次，插装待测试程序，以支持在具体执行程序时收集路径条件和各变量的符号化表达式；第三，根据静态分析和可达性分析的信息判断需要覆盖的路径，求解路径条件生成测试输入，驱动程序执行测试输入，同时进行符号执行并对待检测语句进行符号化检测.

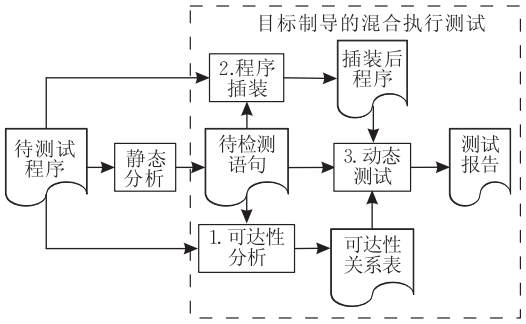


图 2 目标制导的混合执行测试流程图

4.1 程序模型

在详细介绍各步骤之前，我们先给出待测试程序的模型. 基于 CREST[9] 所使用的程序模型，我们定义了如下与 C 语言类似的程序模型，用于描述目标制导的混合执行测试技术. 待测试程序是由 n 个函数所构成的一个集合：$\{f_1, f_2, \cdots, f_n\}$，其中包含一个程序执行的入口函数 main. 一个函数 f_i 由 m_i 条带标签的语句序列 $l_{i,1}:s_{i,1}, l_{i,2}:s_{i,2}, \cdots, l_{i,m_i}:s_{i,m_i}$ 所组成. 组成函数的语句可分为 5 类：(1) 输入语句，其形式为 Input(var,t)，为变量 var 输入 t 字节的值；(2) 函数调用语句，其形式为 $f_i(var_1, var_2, \cdots)$，以 $var_1, var_2, \cdots$ 为参数调用函数 f_i；(3) 赋值语句，其形式为 Assign(var_d,var_s,t)，将 var_s 中 t 字节的值拷贝到 var_d 中；(4) 条件判断语句，其形式为 if c goto l；(5) 终止语句，包含正常终止 halt 和异常终止 abort 两种形式.

对于一条条件判断语句 $l_{i,j}$：if c goto $l_{i,k}$，$l_{i,k}$ 和 $l_{i,j+1}$ 分别为条件 c 为 true 和 false 时执行的下一条语句. 我们称这样的两条语句为分支语句，且互为对应分支，将其表示为 Pair($l_{i,k}$)=$l_{i,j+1}$ 和 $l_{i,k}$=Pair($l_{i,j+1}$). 程序的一条路径可表示为一个语句序列，其形式为(entry,B,exit)，其中 entry 是程序入口语句，exit 是该次执行的终止语句，B 是该次执行过程中顺序经过的分支语句所组成的一个序列：$b_1, b_2, \cdots, b_w$.

4.2 静态分析

为使用静态分析的结果来指导动态测试，降低测试开销，我们首先使用静态分析工具对待测试程序进行分析，并获取程序中可疑语句的缺陷类型及其所在位置. 将可疑语句作为待检测缺陷语句指导测试，其中缺陷的类型将用于构造缺陷模型，用于动态运行时进行相应检查，位置信息则用于计算程序各分支语句到待检测缺陷语句的可达性.

静态分析的结果需要具有尽可能高的精确度，因为误报会给动态测试带来额外的检测开销，而漏报则会导致动态测试时不对漏报的语句进行检测，从而将漏报传递给动态测试. 一些静态分析工具支持将所报告的缺陷根据重要性进行排序，我们可按照测试预算从重要性较高的缺陷开始，依次选取需要动态检测的缺陷，并作为目标指导混合执行测试.

4.3 可达性分析

并非所有的程序路径都能够覆盖待检测缺陷语句，触发静态分析所识别出的待检测缺陷更需要满足特定的条件. 使用程序的结构信息能够部分避免

生成和执行不可达待检测缺陷语句的测试输入，从而将测试的重点尽量集中在能覆盖待检测缺陷语句的测试输入上，降低测试开销.

本文中我们所使用的程序结构信息是基于过程间控制流图(Inter-procedural Control Flow Graph，ICFG)所计算出的程序中各分支语句到可疑语句的可达性关系. ICFG 是一个由节点和边所组成的有向图，一个节点对应于程序中的一条语句，而一条边则从一个节点指向该节点对应程序语句的直接后续语句所对应的节点. 一个由 n 个函数$\{f_1, f_2, \cdots, f_n\}$所构成的程序，语句 $l_{i,j}$：$s_{i,j}$的直接后继语句为如下的一个集合：

- 空集$\varnothing$，若 $l_{i,j}$：$s_{i,j}$为一条终止语句；
- $\{l_{i,j+1}$：$s_{i,j+1}, l_{i,k}$：$s_{i,k}\}$，若 $l_{i,j}$：$s_{i,j}$为一条条件判断语句：if c goto $l_{i,k}$；
- $\{l_{r,t+1}$：$s_{r,t+1} \mid$函数f_r中的语句$l_{r,t}$为$f_i(\cdots)\}$，若 $l_{i,j}$：$s_{i,j}$为函数 f_i的返回语句；
- $\{l_{r,1}$：$s_{r,1}\}$，若 $l_{i,j}$：$s_{i,j}$为一条函数调用语句：$f_r(\cdots)$；
- $\{l_{i,j+1}$：$s_{i,j+1}\}$，其它情况.

基于 ICFG，在转置分支 b 所对应的路径条件前，先判断 b 对应分支 Pair(b)的可达性关系，若 Pair(b)能达到某待检测缺陷语句，则进行约束求解并生成相应测试输入，否则，跳过该分支，继续转置其它路径条件.

算法 1 描述了计算程序分支语句与可疑语句间可达性关系的算法. 该算法将程序的 ICFG 和静态分析所提供的待检测缺陷语句集合 V 作为输入. 输出 map *reachability* 存放从各分支语句到该分支可达待检测缺陷语句集合的映射. 第 1～5 行调用 Reach 过程计算分支语句到各待检测缺陷语句的可达性. Reach 过程(8～15 行)中第 9 行获取当前语句 l 的直接后继语句集合 *successors*，第 10 行逐一检查 *successors* 中的语句 l_i是否为待检测缺陷语句，若是，将 l_i加入当前遍历分支的可达待检测缺陷语句集合 *reachable* 中. 第 13 行迭代调用 Reach 过程遍历 l_i的可达语句.

算法 1.　分支可达性计算算法.

输入：ICFG：*icfg*//程序 ICFG 图
　　set：*V*//待检测缺陷语句集合
输出：map *reachability*//从分支到可达待检测缺陷语句集合的映射

```
1.  for each branch statement b in icfg {
2.      set reachable
3.      Reach(b, &reachable)
4.      reachability[b] := reachable
5.  }
6.  return reachability
7.
8.  Procedure: Reach(statement l, set *reachable){
                                    //递归遍历 l 可达语句
9.      set successors := Successors(l)
                                    //l 的直接后继语句
10.     for each statement l_i in successors{
11.         if l_i ∈ V
12.             reachable.push(l_i)
13.         Reach(l_i, &reachable)
14.     }
15. }
```

4.4　程序插装

为了在执行具体测试输入的同时进行符号执行，需要对源程序进行插装以收集执行过程中对各个基本类型变量、指针变量和缓冲区的赋值、修改等操作及路径条件的符号表达式. 我们将符号执行的功能封装为一个函数库，该函数库包含了 4 种类型的函数：输入声明、赋值修改跟踪、路径条件跟踪和缺陷检测.

算法 2 描述了程序插装的算法. 该算法将源程序和静态分析所提供的待检测语句集合 V 作为输入，输出插装后的程序. 第 1 行逐一遍历程序中的函数，第 2 行遍历函数中各条语句 l_i：若 l_i为输入语句，则在 l_i前插装输入声明库函数，用于声明输入变量的符号表达式(3～4 行)；若 l_i为赋值修改语句，则在 l_i前插装赋值修改跟踪库函数，用于符号化表示该操作，以对程序中的所有变量建立一个由输入变量和常量所构成的符号表示式(5～6 行)；若 l_i为条件判断语句，则在条件判断语句及其所对应的分支语句处插装路径条件跟踪库函数，用于收集路径条件的符号表达式(7～13 行)；若 l_i为静态分析所报告的待检测缺陷语句，则在 l_i前插装缺陷检测库函数，用于符号化检查在当前执行路径条件下，该缺陷被触发的可能性(14～15 行).

算法 2.　程序插装算法.

输入：set：*F*//源程序
　　set：*V*//待检测缺陷语句集合
输出：set F'//插装后程序

```
1.  for each function f_i in F{
2.      for each statement l_i in f_i{
3.          if l_i is a input statement
```

```
4.        instrument lib function AddInput() before
          l_i in F
5.      else if l_i is a assign or update statement
6.        instrument lib function Operation() before
          l_i in F
7.      else if l_i is a conditional statement{
8.        instrument lib function Conditional() before
          l_i in F
9.        b_t := true branch of l_i
10.       instrument lib function TrueBranch() after
          b_t in F
11.       b_f := false branch of l_i
12.       instrument lib function FalseBranch() after
          b_f in F
13.     }
14.     else if l_i ∈ V
15.       instrument lib function Checking() before
          l_i in F
16.   }
17. }
```

4.5 动态测试

动态测试包括测试生成和测试执行两个部分.测试输入的生成过程中,在按照深度优先规则转置各分支的路径条件前,首先检查对应分支是否可达至少一条待检测缺陷语句,若可达则对转置后的路径条件进行约束求解,生成覆盖新路径的测试输入,否则跳过该分支,继续遍历.测试执行过程则运行所生成的测试输入,并在运行时收集路径条件的符号化表达式.在测试执行过程中,检查当前分支对各待检测缺陷语句的可达性,若均不可达,则终止测试执行.此外,在测试执行过程中还需要检查各待检测缺陷是否会被触发.

算法 3 描述了目标制导的混合执行测试技术的测试输入生成算法.第 2 行为待测试程序准备初始输入值,基本类型输入变量的初始值默认为 0,字符串输入变量的初始值默认为空字符串.若输入变量的范围可知,基本类型输入变量可初始化为最大值和最小值,字符串输入变量可初始化为所允许的最大长度和最小长度.

算法 3. 目标制导的混合执行测试输入生成算法.

```
输入: set: V//待检测缺陷语句集合
      map reachability//从分支到可达待检测缺陷集
                       合的映射
1.  list: B /*存放分支语句*/,
        list C /*存放分支对应的路径条件*/
2.  list I=InitInput()//初始化输入
3.  Run_Program(I,&B,&C)//以 I 为输入运行程序
4.  Target_Guided_Search(1,B,C)
5.
6.  Procedure: Target_Guided_Search(int depth,
                          list B,list C){
7.    bool pairedReachable := false
                  //对应分支是否可达待确认缺陷
8.    for(int i := sizeOf(B); i>=depth; i--) {
9.      branch pairedBranch := Pair(B[i])
                  //B[i]对应分支
10.     if reachability[pairedBranch]⋂V≠null{
11.       pairedReachable := true
12.     }
13.     if (pairedReachable=true){
14.       list I' := solve (C[1]⋂…⋂C[i-1]⋂
                          ¬(C[i]))
15.       list: B'/*存放分支语句*/,
              list C' /*存放路径条件*/
16.       Run_Program(I',&B',&C')
                          //以 I'为输入运行程序
17.       Target_Guided_Search(++depth,B',C')
18.     }
19.   }
20. }
```

第 3 行使用初始输入值驱动程序运行,并在链表 B,C 中分别顺序记录所覆盖的分支语句及所对应路径条件的符号表达式.在程序的执行过程中,若执行到静态分析所报告的待检测缺陷语句,则检测该语句的缺陷是否会发生.若会发生,确认该缺陷并从 V 中删除;若不会发生,结合路径条件对该语句的缺陷触发条件进行约束求解,若可解,生成并执行满足上述条件的测试输入以确认该缺陷.当检测到缺陷将会发生时,修改程序状态,避免错误实际发生(如:检测到缓冲区溢出缺陷会发生时,为该缓冲区分配新的地址空间,以避免实际发生缓冲区溢出),并继续运行程序以发现隐藏在程序路径中更深位置的缺陷.程序的可达性信息还被用于降低执行测试输入的开销,当执行到的分支语句对所有待检测缺陷语句均不可达时,提前终止程序运行.

Target_Guided_Search 过程(6~20 行)描述了目标制导的搜索算法.第 8 行对所收集到的分支链表 B 进行深度优先遍历,对分支 $B[i]$,第 10 行根据程序的可达性信息判断该分支的对应分支 Pair($B[i]$)是否可达至少一条待检测缺陷语句.若可达,第 14 行求解路径条件 $C[1]\bigcap\cdots\bigcap C[i-1]\bigcap\neg(C[i])$,

生成的测试输入将沿路径($entry, B[1], B[2], \cdots,$ Pair($B[i]$), $\cdots, exit$)运行. 第 16 行使用生成的输入值驱动程序运行，并在链表 B'，C' 中分别顺序记录所覆盖的分支语句及所对应路径条件的符号表达式. 当所有待检测缺陷都已得到确认，或所有可达待检测缺陷语句的路径都已被覆盖后，终止测试.

5　实现和评估

5.1　原型工具实现

为了使本文所提出的方法不失一般性，也具有针对性，基于上述目标制导的混合执行测试技术，我们面向一类典型的程序缺陷：缓冲区溢出，实现了一项原型测试工具：TARGET(TARget-GuidEd Testing)①，以检测 C 语言程序中的缓冲区溢出缺陷. TARGET 的主页上提供了详细的使用说明及下载.

为了检测缓冲区溢出缺陷，除了对缓冲区的内容进行符号化建模外，我们还对缓冲区的大小(size)和已使用长度(length)提供了符号化表示. 在混合执行测试技术的基础上，我们为指针变量和缓冲区也建立了映射关系. 在 TARGET 的符号映射关系 δ 中，将一个指针变量从其具体地址 $addr$ 映射到一个三元组($size, len, C$)表示的缓冲区，其中 $size$ 是一个符号表达式，表示该缓冲区变量分配空间的大小，len 是一个符号表达式，表示该缓冲区变量已使用的长度，即该缓冲区变量中终结符'\0'所在位置，$C=c_1, c_2, \cdots, c_v$ 为一个序列，是该缓冲区所存放内容的符号表达式. 我们借鉴了 SPLAT[10] 的思想，仅为缓冲区变量前 v 个位置进行符号化建模以提高系统的可扩展性. 我们将一个指针变量 p 表示为一个二元组($addr, off$)，其中 $addr$ 是一个无符号长整形数，是该指针变量的地址，off 则是 p 所指向缓冲区中的位置距离该缓冲区起始位置偏移量的符号表达式，$\delta(addr)$ 返回 p 所指向的缓冲区 $b(size, len, C)$. 图 3 描述了 TARGET 中缓冲区的符号化表示方式. 为了在测试过程中通过符号化的方式来检测缓冲区溢出缺陷，需要对该类缺陷定义相应的缺陷模型. 表 1 描述了缓冲区溢出缺陷的缺陷模型.

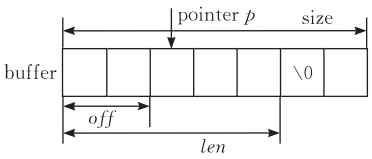


图 3　缓冲区符号化表示

表 1　缓冲区溢出缺陷模型

缓冲区操作	溢出条件
strcpy(p_d, p_s)	$len_s - off_s >= size_d - off_d$
strcat(p_d, p_s)	$len_s - off_s + len_d >= size_d$
$*p_d := var$	$off_d > size_d$

注：表格中 $p_d=(addr_d, off_d)$，$b_d=(size_d, len_d, C_d)=\delta(addr_d)\neq$ null，$p_s=(addr_s, off_s)$，$b_s=(size_s, len_s, C_s)=\delta(addr_s)\neq$ null.

TARGET 的动态测试部分是基于混合执行测试工具 CREST② 实现的，采用的约束求解器为 Yices③. TARGET 的开发环境是 Window 7 Ultimate、Visual Studio 2008 及 Microsoft Phoenix SDK 2008④. 其中 TARGET 的插装器是作为 Phoenix 的一个插件来实现的. 待测程序仅需在 Phoenix 平台下使用该插件进行编译即可生成包含符号执行所需库函数的可执行程序，该程序可在执行具体输入的同时进行符号执行. 表 2 描述了部分函数的符号化执行语义.

表 2　部分函数符号化执行语义

函数	符号化执行语义
$p_d :=$ input($size$)	创建 $size$ 大小的缓冲区 b，其中 $len := size$，$off_d := 0$，增加映射关系 $\delta(addr_d) := b$
$p_d :=$ malloc($size$)	创建 $size$ 大小的缓冲区 b，其中 $off_d := 0$，$C :=$ null，增加映射关系 $\delta(addr_d) := b$
$p_d := p_s \pm v$	$\delta(add_d) := \delta(add_s)$，$off_d := off_s \pm v$
$*p_d :=$ '\0'	$b := \delta(addr_d)$，$len := off_d$

注：表格中 $p_d=(addr_d, off_d)$，$p_s=(addr_s, off_s)$，$b=(size, len, C)$.

5.2　实验设计

为评估目标制导的混合执行测试技术的有效性，我们在同样的开发平台上实现了 SPLAT 技术[10]，并使用上述两个工具分别对一组基准程序进行了测试，从缺陷检测能力、测试输入生成开销和动态检测开销 3 个方面进行了度量. 测试的基准程序包括 WuFTP-1、Sendmail-2 和 gzip-1.2.4，其中 WuFTP-1 和 Sendmail-2 取自文献[16]中的一组含有代表性缓冲区溢出缺陷的基准程序，gzip-1.2.4⑤ 是一个开源的文件压缩程序. 由于原有的 SPLAT 技术发现一个缺陷后就会报告错误并停止测试，为便于比较，我们采取的方式是立即修正所发现的缺陷并继续进行测试，直到不再发现新的缺陷为止. 实验运行的硬件环境为 Intel Duo Core 2.26 GHz 处理器和 2GB 内存的计算机.

① http://seg.nju.edu.cn/TARGET/
② http://code.google.com/p/crest/
③ http://yices.csl.sri.com/
④ http://connect.microsoft.com/Phoenix
⑤ http://www.gzip.org/

在本实验中，TARGET 工具所需的待检测缺陷信息由静态分析工具 Marple[15] 提供. Marple 将可能存在缺陷的语句分为安全的(safe)、有缺陷的(vulnerable)、不确定的(don't know)几类. 安全的语句在任何输入条件下均不会被触发缺陷；有缺陷的语句在一定条件下可能会被触发缺陷；不确定类型的语句则是 Marple 不能判断缺陷是否能被触发的语句. 库函数调用、非线性操作及复杂的指针运算都可能导致缺陷是否会被触发不能确定. 在本实验中，我们将 Marple 报告为有缺陷的语句作为待检测缺陷语句指导动态测试. Marple 对 3 个基准程序的静态分析结果如表 3 所示.

表 3 Marple 分析结果统计

基准程序	报告缺陷数[15]	漏报缺陷数	误报缺陷数*
WuFTP-1	4	1	0
Sendmail-2	4	0	0
gzip-1.2.4	9**	0	4

注：* Marple 分析过程中所考虑的是可能的最坏情况，但由于操作系统等环境因素的限制，如系统所支持的最长文件长度有限，部分缺陷在实际的系统中不能被触发，本文将这类缺陷也暂归为误报.

** 原文中 gzip-1.2.4 程序所报告的缺陷数为 10. 在论文发表后，原文作者对 Marple 进行了改进，减少了一处误报.

5.3 实验结果分析

5.3.1 缺陷检测能力

表 4 中比较了两个工具检测缺陷的能力，其中第 1、2 列给出了基准程序的名称及该程序的代码行数(不含注释和空行)，第 3、4 列分别给出了两个工具检测到的缺陷数、漏报的缺陷数及消耗的时间. 其中 TARGET 所消耗的时间由使用 Marple 进行静态分析的时间和测试的时间两部分所组成.

从表 4 中可以看出，TARGET 在 WuFTP-1 和 Sendmail-2 中均检测到了更多的缺陷. 其中，TARGET 在基准程序 WuFTP-1 中所漏报的缺陷是由于静态分析工具未将该缺陷作为有缺陷的语句报告，因而未在测试时进行检测. 总体上看，TARGET 所检测到的缺陷数比 SPLAT 技术多 5 个，其原因主要有如下两个方面：一方面，当发现会发生缓冲区溢出时，TARGET 为缓冲区分配新的空间，继续安全地运行程序，从而能覆盖更深的程序路径，识别出更多的缺陷，而 SPLAT 技术则立即报告错误并停止测试，为继续测试而修正所发现的缺陷可能会隐藏其它的潜在缺陷；另一方面，除了将默认输入作为测试的初始输入外，TARGET 还支持将变量的边界值作为测试的初始输入，从而能更覆盖更多的程序路径. 从表中还可以看出，TARGET 所花费的总时间大为降低，所消耗的总时间约为 SPLAT 技术消耗总时间的 1/6. 其原因是通过使用静态分析的结果和可达性信息，程序中的部分路径不再需要进行测试，从而节省了测试时间.

表 4 测试时间和缺陷检测能力比较

基准程序	规模/k	SPLAT 技术			TARGET			
		检测缺陷数	漏报缺陷数	时间/s	检测缺陷数	漏报缺陷数	时间/s	
							分析[15]	测试
WuFTP-1	0.2	2	3	1020	4	1	2.1	3
Sendmail-2	0.3	1	3	600	4	0	1.1	167
gzip-1.2.4	4.5	3	2	485	3	2	26.2	176

5.3.2 测试输入生成开销

表 5 中比较了两种工具在生成测试输入上的开销. 尝试生成次数列给出了两个工具尝试生成测试输入的次数，即调用约束求解器求解路径条件的次数；成功生成次数列给出了两个工具成功生成测试输入的次数，即约束求解器成功解出路径条件的次数，也即是运行待测试程序的次数；时间列给出了测试过程中在生成测试输入上所消耗的时间.

表 5 测试输入生成的开销

基准程序	SPLAT 技术			TARGET		
	尝试生成次数	成功生成次数	时间/ms	尝试生成次数	成功生成次数	时间/ms
WuFTP-1	13995	1747	25651	3	3	105
Sendmail-2	1028	774	52877	5362	252	643
gzip-1.2.4	4252	482	8711	1271	233	1635

从表 5 中可以看出，TARGET 降低了生成测试用例的次数和所耗费的时间，TARGET 尝试生成测试输入的总次数约为 SPLAT 技术的 1/3，成功生成测试输入的次数不到 SPLAT 技术的 1/6，所消耗的时间约为 SPLAT 技术的 1/40. 其原因是通过使用静态分析的结果和程序的结构信息，程序中的部

分路径不再需要动态检查，也就不需要生成相应的测试输入，从而极大地节省了生成测试输入的时间.

值得注意的是在 Sendmail-2 基准程序中，TARGET 尝试生成测试输入的次数更多. 这是由于错误的关联性所引起的，对一个错误的修正可能会导致程序路径的减少、其它错误被隐藏等情况发生. 该实例中正是出现了这样的情况，对其中一个错误的修改导致路径大量缩减，而 TARGET 是在原来未经修改的程序上进行测试的，所需要覆盖的路径数更多，所以尝试生成覆盖这些路径的测试输入的次数更多. 从表 5 中我们还可以看出，测试输入生成开销的主要因素是成功生成测试输入的次数，而不是尝试生成测试输入的次数.

5.3.3　缺陷检测开销

表 6 中比较了动态检测待确认缺陷的开销. 可疑语句数列给出了动态测试过程中需要进行检测的语句数；动态检查次数列给出了在测试过程中检查可疑语句中缺陷是否可能会被触发的次数；时间列则给出了测试过程中在检查可疑语句中缺陷是否会被触发所消耗的时间.

表 6　动态缺陷检测开销

基准程序	SPLAT 技术			TARGET		
	可疑语句数	动态检查次数	时间/ms	可疑语句数	动态检查次数	时间/ms
WuFTP-1	13	19472	89932	4	7	35
Sendmail-2	8	3921	9519	4	565	467
gzip-1.2.4	29	2726	13653	9	709	2040

从表 6 中可以看出，TARGET 测试过程中需要进行检测的可疑语句数约为 SPLAT 技术的 1/3，动态检查的次数不到 SPLAT 技术的 1/20，所消耗的时间不到 SPLAT 技术的 1/40. 其原因是由于 SPLAT 技术未使用静态分析的信息，因此需要检查程序中所有涉及缓冲区修改的语句，而 TARGET 仅需检查静态分析工具所报告的待检测缺陷所在语句，因而 SPLAT 技术动态检测缺陷的次数和时间开销均远高于 TARGET.

5.4　讨　论

从上面 3 个表格中可以看出，与原有的混合执行测试技术相比较，TARGET 有效降低了测试生成、缺陷检测及总测试时间的开销，并能发现程序中的更多缺陷.

目前 TARGET 仅支持检测 C 语言中的缓冲区溢出缺陷，但本方法能通过增加缺陷模型和扩充插装函数库的方式，方便地扩展到支持检测多种程序缺陷，如整形数溢出、内存溢出等. 另外，静态分析结果的精确度将会对本方法产生影响：误报会增加动态测试的开销，而漏报则会导致动态测试时不对漏报的可疑语句进行检测，从而不能检测到缺陷. 因此，高精度的静态分析结果能进一步提高本方法的效果.

6　相关工作讨论

软件缺陷的检测技术一般可分为静态和动态两类，部分动态缺陷检测技术，如 ProPolice① 和 CRED[17] 采用的方法不是生成触发软件缺陷的测试用例，而是致力于如何有效地插装源程序以通过运行时监控的方式检测错误的发生. 在本节中，我们主要讨论缓冲区溢出的测试生成技术和静动态结合的软件缺陷检测技术.

首先讨论缓冲区溢出的测试生成技术. Grosso 等[18] 使用遗传算法为函数调用生成输入来发现缓冲区溢出. Shahriar 等[19] 针对缓冲区溢出这类缺陷提出了一组变异操作符，其主要用途是评估测试集的充分度. 基于 DART[6]，Xu 等[10] 提出了一种缓冲区溢出测试技术：SPLAT，其主要贡献是提出仅为缓冲区的一段前缀而不是整个缓冲区进行符号化建模，从而提高该方法的可扩展性. 该方法与其它的混合执行测试技术一样，无差别地对待程序中的所有路径，并尽可能多地遍历程序路径. 与之相比，我们的方法利用静态分析的结果来减少测试过程中需要遍历的路径，并减少需要在运行时检查的可疑语句，从而降低了测试的开销，提高了可扩展性. 同时，我们的方法也借鉴了 SPLAT 仅为缓冲区的一段前缀进行符号化建模的思想，进一步提高了可扩展性.

另一个相关的领域是静动态结合的测试技术. Aggarwal 等[20] 使用 BOON[13] 的静态分析结果，减少了需要测试的缓冲区溢出的数量，从而减少了生成的测试用例. 与我们的方法最为相似的是 Christoph 等提出的 Check 'n' Crash[21] 和 DSD-Crasher[22].

① Gcc extension for protecting applications from stack smashing attacks, http://www.trl.ibm.com/projects/security/ssp/

Check 'n' Crash 使用 ESC/JAVA[23] 分析 JAVA 程序中函数的前置条件,并在所计算出的前置条件的基础上生成测试用例.由于该技术中所使用的静态分析是过程内的,因此所检测到的程序缺陷很有可能为误报.为解决这一问题,DSD-Crasher 提出首先执行测试用例并通过 Daikon[24] 收集程序中的不变式,获取过程间的约束条件,从而在一定程度上降低误报率.然而,Daikon 所收集到的不变式所描述的是系统在正常使用的情况下可能满足的不变式,而安全缺陷往往是系统在异常的使用方式下才会被触发的,另外,Daikon 所生成的不变式是不精确的,所以该方法仍然会有一定的误报率.同时,该方法还需要一组较为充分的测试用例集,这在大多数情况下是难以获取的.与之相比,我们的方法使用静态分析结果和程序结构信息指导自动地生成测试输入,触发并确认真正的缺陷,从而避免产生误报.

7 结 论

针对现有的混合执行测试技术在遍历程序路径和输入空间时缺乏指导,导致生成和执行不能覆盖潜在缺陷语句的测试输入耗费了大量测试开销的问题,本文提出了一种基于静态分析结果的目标制导的混合执行测试方法:使用静态分析工具分析待测程序中可能含有缺陷的可疑语句及其缺陷类型,并作为目标指导测试.混合执行测试方法由 3 个步骤组成:可达性分析、程序插装和动态测试.基于该方法我们实现了一项针对 C 语言缓冲区溢出缺陷的原型测试工具:TARGET,并在一组 C 语言基准程序上进行了实例研究.实验结果表明与原有的混合执行测试方法相比较,使用静态分析信息能有效地降低测试开销.

在本文研究的基础上,我们计划将该方法用于测试更多类型的程序缺陷,同时,我们还计划使用更多的静态分析信息来进一步降低测试开销.

致 谢 本文的部分工作是第一作者在 Computer Science Department, University of Virginia, USA 进行学术访问期间完成的.Mary Lou Soffa 教授和 Wei Le 博士为本项工作的完成提供了宝贵的意见,特别是对静态分析工具 Marple 的使用提供了大量的帮助,在此表示衷心的感谢!

参 考 文 献

[1] Pezzè M, Young M. Software Testing and Analysis: Process, Principles and Techniques. Hoboken, NJ: John Wiley & Sons, 2007

[2] Mei Hong, Wang Qian-Xiang, Zhang Lu, Wang Ji. Software analysis: A road map. Chinese Journal of Computers, 2009, 32(9): 1697-1710(in Chinese)
(梅宏,王千祥,张路,王戟. 软件分析技术进展. 计算机学报, 2009, 32(9): 1697-1710)

[3] Emanuelsson P, Nilsson U. A comparative study of industrial static analysis tools. Electronic Notes in Theoretical Computer Science, 2008, 217: 5-21

[4] Zhang Jian. Sharp static analysis of programs. Chinese Journal of Computers, 2008, 31(9): 1549-1553(in Chinese)
(张健. 精确的程序静态分析. 计算机学报, 2008, 31(9): 1549-1553)

[5] Bertolino A. Software testing research: Achievements, challenges, dreams//Proceedings of the Future of Software Engineering (FOSE'07). Washington, DC, USA: IEEE Computer Society, 2007: 85-103

[6] Godefroid P, Klarlund N, Sen K. DART: Directed automated random testing//Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'05). New York, NY, USA: ACM, 2005: 213-223

[7] Sen K, Marinov D, Agha G. CUTE: A concolic unit testing engine for C//Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE-13). New York, NY, USA: ACM, 2005: 263-272

[8] Sen K, Agha G. CUTE and Jcute: Concolic unit testing and explicit path model-checking tools//Proceedings of the 18th International Conference on Computer Aided Verification (CAV'06). Lecture Notes in Computer Science 4144. Berlin, Heidelberg: Springer, 2006: 419-423

[9] Burnim J, Sen K. Heuristics for scalable dynamic test generation//Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering (ASE'08). Washington, DC, USA: IEEE Computer Society, 2008: 443-446

[10] Xu R-G, Godefroid P, Majumdar R. Testing for buffer overflows with length abstraction//Proceedings of the 2008 International Symposium on Software Testing and Analysis (ISSTA'08). New York, NY, USA: ACM, 2008: 27-38

[11] Evans D, Larochelle D. Improving security using extensible lightweight static analysis. IEEE Software, 2002, 19(1): 42-51

[12] Xie Y, Chou A, Engler D. ARCHER: Using symbolic, path-sensitive analysis to detect memory access errors//Proceedings of the 9th European Software Engineering Conference Held Jointly with 11th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE-11). New York, NY, USA: ACM, 2003: 327-336

[13] Wagner D, Foster J S, Brewer E A, Aiken A. A first step towards automated detection of buffer overrun vulnerabilities//Proceedings of the Network and Distributed System Security Symposium (NDSS'00). San Diego, California, 2000: 3-17

[14] Bush W R, Pincus J D, Sielaff D J. A static analyzer for finding dynamic programming errors. Software Practice & Experience, 2000, 30(7): 775-802

[15] Le W, Soffa M L. Marple: A demand-driven path-sensitive buffer overflow detector//Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering (SIGSOFT'08/FSE-16). New York, NY, USA: ACM, 2008: 272-282

[16] Zitser M, Lippmann R, Leek T. Testing static analysis tools using exploitable buffer overflows from open source code. ACM SIGSOFT Software Engineering Notes, 2004, 29(6): 97-106

[17] Ruwase O, Lam M S. A practical dynamic buffer overflow detector//Proceedings of the 11th Annual Network and Distributed System Security Symposium(NDSS'04). San Diego, California, 2004: 159-169

[18] Grosso C D, Antoniol G, Merlo E, Galinier P. Detecting buffer overflow via automatic test input data generation. Computers and Operations Research, 2008, 35(10): 3125-3143

[19] Shahriar H, Zulkernine M. Mutation-based testing of buffer overflow vulnerabilities//Proceedings of the 32nd Annual IEEE International Computer Software and Applications Conference(COMPSAC'08). Washington, DC, USA: IEEE Computer Society, 2008: 979-984

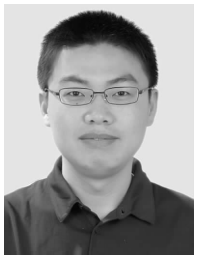
[20] Aggarwal A, Jalote P. Integrating static and dynamic analysis for detecting vulnerabilities//Proceedings of the 30th Annual International Computer Software and Applications Conference (COMPSAC'06). Washington, DC, USA: IEEE Computer Society, 2006: 343-350

[21] Csallner C, Smaragdakis Y. Check '*n*' crash: Combining static checking and testing//Proceedings of the 27th International Conference on Software Engineering(ICSE'05). New York, NY, USA: ACM, 2005: 422-431

[22] Csallner C, Smaragdakis Y, Xie T. DSD-Crasher: A hybrid analysis tool for bug finding. ACM Transactions on Software Engineering and Methodology, 2008, 17(2): 1-37

[23] Flanagan C, Rustan K, Leino M, Lillibridge M, Nelson G, Saxe J B, Stata R. Extended static checking for Java//Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'02). New York, NY, USA: ACM, 2002: 234-245

[24] Ernst M D, Cockrell J, Griswold W G, Notkin D. Dynamically discovering likely program invariants to support program evolution//Proceedings of the 21st International Conference on Software Engineering (ICSE'99). New York, NY, USA: ACM, 1999: 213-224



CUI Zhan-Qi, born in 1984, Ph. D. candidate. His research interests include software engineering and software testing.

WANG Lin-Zhang, born in 1973, Ph. D., associate professor. His research interests include software engineering and software testing.

LI Xuan-Dong, born in 1963, Ph. D., professor, Ph. D. supervisor. His research interests include software engineering, formal method, software modeling and analysis, software testing and verification.

Background

In order to improve the quality of software before releasing, dynamic testing and static analysis are the most widely used techniques for finding defects. Static analysis can find defects at the implementation stage, which is earlier than dynamic testing. In addition, static analysis costs less human efforts, as it can be highly automated. Dynamic testing, on the other hand, is effective in avoiding false positives and confirming real defects with concrete test inputs. Moreover, the concrete test cases, which trigger the defects, are also very helpful for debuggers to fix the defects. However, most dynamic testing techniques need an adequate test suite and cost a large amount of runtime overhead, which are not available in most cases.

To address the problem of generating an adequate test suite, concolic testing has been proposed for testing software automatically, with the goal of exploring as many paths of a program as possible. In concolic testing, the program under test is concretely executed and symbolically evaluated simultaneously. Instrumentations are inserted to the program to collect the symbolic path constraints and value updates during program execution. The symbolic constraints are solved to generate test inputs which execute new paths. When symbolic expressions cannot be solved, they are simplified by replacing with corresponding concrete values. However, concolic

testing lacks the prior knowledge of target faulty statements. From the perspective of detecting defects, time is wasted in generating and executing test inputs that cannot find defects.

Since only test inputs that can cover suspicious statements may exploit real defects, test inputs cannot reach suspicious statements cannot make any contribution for finding defects. If the information of where a suspicious statement is located and how the defect could be exploited are available, one way to reduce testing costs without compromising defects detecting capability is to avoid generating and executing test inputs that cannot reach the suspicious statements. Some static analysis tools can report the position of suspicious statements and how they can be exploited. But static analysis tools suffer high false positive rates due to conservative assumptions used during the analysis process. Identifying real defects from a large amount of false alarms is still a labor intensive task.

The static analysis and dynamic testing approaches are complementary. The benefit of using static information to guide dynamic testing is twofold, first, the potential vulnerable information provided by static analysis tools can be used to guide dynamic testing process; second, the runtime information provided by dynamic testing can be used to confirm and refine the results of static analysis tools. Our research problem is how to leverage the report of static analysis tools to guide dynamic testing and thus reduce the costs of dynamic testing.