软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2017,28(4):827−844 [doi: 10.13328/j.cnki.jos.005189]

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

# C/C++程序静态内存泄漏警报自动确认方法*

李 筱[1,2], 周 严[1,2], 李孟宸[1,2], 陈园军[1,2], XU Guo-Qing[3], 王林章[1,2], 李宣东[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 210023)

[2](南京大学 计算机科学与技术系,江苏 南京 210023)

[3](Department of Computer Science, University of California, Irvine, USA)

通讯作者: 王林章, Email: lzwang@nju.edu.cn

**摘 要**: 内存泄漏是C/C++程序的一种常见的、难以发现的缺陷,一直困扰着软件开发者,尤其是针对长时间运行的程序或者系统软件,内存泄漏的后果十分严重.针对内存泄漏的检测,目前主要有静态分析和动态测试两种方法.动态测试实际运行程序具有较大开销,同时依赖测试用例的质量;静态分析技术及自动化工具已被学术界和工业界广泛运用于内存泄漏缺陷检测中,然而由于静态分析采取了保守的策略,其结果往往包含数量巨大的误报,需要通过进一步的人工确认来甄别误报.但人工确认静态分析的结果耗时且容易出错,严重限制了静态分析技术的实用性.提出一种基于混合执行测试的静态内存泄漏警报的自动化确认方法:首先,针对静态分析报告的目标程序中内存泄漏的静态警报,对目标程序进行控制流分析,并计算警报的可达性,形成制导信息;其次,基于警报制导信息对目标程序进行混合执行测试;最后,在混合执行测试过程中,监控追踪内存对象的状态,判定内存泄漏是否发生,对静态警报进行动态确认并分类.实验结果表明:该方法可对静态内存泄漏警报进行有效的分类,显著降低了人工确认的工作量.

**关键词**: 内存泄漏;静态分析;警报;混合执行测试;确认;分类

**中图法分类号**: TP311

中文引用格式: 李筱,周严,李孟宸,陈园军,Xu Guo-Qiang,王林章,李宣东.C/C++程序静态内存泄漏警报自动确认方法.软件学报, 2017,28(4):827−844. http://www.jos.org.cn/1000-9825/5189.htm

英文引用格式: Li X, Zhou Y, Li MC, Chen YJ, Xu GQ, Wang LZ, Li XD. Automatically validating static memory leak warnings for C/C++ programs. Ruan Jian Xue Bao/Journal of Software, 2017,28(4):827−844 (in Chinese). http://www.jos.org.cn/1000-9825/5189.htm

## Automatically Validating Static Memory Leak Warnings for C/C++ Programs

LI Xiao[1,2], ZHOU Yan[1,2], LI Meng-Chen[1,2], CHEN Yuan-Jun[1,2], XU Guo-Qing[3], WANG Lin-Zhang[1,2], LI Xuan-Dong[1,2]

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

[2](Department of Computer Science and Technology, Nanjing University, Nanjing 210023, China)

[3](Department of Computer Science, University of California, Irvine, USA)

**Abstract**: Memory leak, which has perplexed software developers for a long time because of imperceptibility, is a very common bug for C/C++ programs and can do serious harm especially for long-running program or system software. Aiming at this problem, both static and dynamic program analysis techniques have been attempted. Dynamic program analysis technique detects memory leak by running the program, which has huge overhead and depends on the quality of test cases. Static analysis technology and automatic tools are widely used in the work of detecting memory leaks among academic community and industrial community. Since it uses conservative algorithm,

* 基金项目: 国家自然科学基金(61202148, 61332015, 61373078, 61272245); 国家重点研发计划(2016YFB1000802)

Foundation item: National Natural Science Foundation of China (61472179, 61572249, 61632015, 61561146394); State Key Research and Development Program (2016YFB1000802)

收稿时间: 2016-01-15; 修改时间: 2016-05-06; 采用时间: 2016-09-08; jos 在线出版时间: 2017-01-24

CNKI 网络优先出版: 2017-02-20 13:51:06, http://www.cnki.net/kcms/detail/11.2560.TP.20170220.1351.006.html

Static analysis is able to detect a lot of defects but at the sometime increases the false positives, which needs manual confirmation. As manual confirmation is time-consuming and error prone, it limits the practicability of the technology. In this paper, a novel method to automatically validate static memory leak warnings is proposed based on concolic testing. First, drawing on the memory leak warnings given by static analysis report, the control flow of the target program is analyzed and the reachability of the target path is calculated. Then the path information is used to guide the concolic testing and execute program in the particular path. Finally, the static warnings is validated by tracking memory object during execution. Experimental results show that this method can effectively classify static warnings and significantly reduce the workload of manual validation.

**Key words**: memory leak; static analysis; warning; concolic testing; validation; classification

C/C++语言中,内存的分配与回收都是由程序员在编写代码时主动完成的,好处是内存管理的开销较小,程序拥有更高的执行效率;弊端是依赖于程序员的水平,随着代码规模的扩大,极容易遗漏释放空间的步骤,或者一些不规范的编程可能会使程序具有安全隐患.如果对内存管理不当,可能导致程序中存在内存缺陷,甚至会在运行时产生内存故障错误.内存泄漏则是各类缺陷中十分棘手的一种,对系统的稳定运行威胁较大.当动态分配的内存在程序结束之前没有被回收时,则发生了内存泄漏.由于系统软件,如操作系统、编译器、开发环境等都是由 C/C++语言实现的,不可避免地存在内存泄漏缺陷,特别是一些在服务器上长期运行的软件,若存在内存泄漏则会造成严重后果,例如性能下降、程序终止、系统崩溃、无法提供服务等,因而,内存泄漏也成为软件安全的一个重要漏洞.因此,工业界和学术界都在研究如何检查和消除内存泄漏问题,当前使用较多的主要是静态分析和动态测试手段.

动态测试技术[1−5]经常用于检测内存泄漏.动态测试需要真实的运行程序并监测内存资源的分配和释放情况,从而判断是否发生了内存泄漏.动态测试能较为准确地发现错误以及找到隐蔽的内存泄漏问题,但是动态测试需要较大的运行开销,包括时间和空间等,同时依赖测试用例的覆盖度,一个较高的测试覆盖度可以得到较好的检测结果.

静态分析也是检测 C/C++程序中内存泄漏的常用手段.静态分析通常是寻找内存分配位置以及相应的释放点的配对,即:验证是否所有的路径都会存在正确的内存释放,一旦某条路径中未含有内存释放操作,则被认为疑似泄漏.因此,静态分析的手段是对路径敏感的.由于静态分析技术不会运行代码,自动化程度较高,深受使用者喜爱.当前,有许多静态分析的工作[6−10],也有一些商用的静态分析工具,比如 Klocwork[11],Coverity[12]以及 HP Fortify[13],这些工具在商业的软件开发中应用广泛.

但是,静态分析工具的查全率依赖于检测规则,对于较大规模的软件而言,程序中含有较为复杂的指针运算以及路径信息,为了分析这类较大数量的代码,会降低查全率,比如路径信息不全等,从而造成漏报.但是由于采用了保守的策略,可能存在误报.目前,工业界能够采用的办法是人工对警报进行逐个确认,确认为缺陷的警报才交由开发人员修复.即使是成熟的商用静态分析工具,例如 HP Fortify[13],也会得到大量的不确定警报,需要人工再次确认.人工确认静态分析警报需要确认者具有专家级知识,确认过程费时、费力,效率低下,随着系统规模的不断扩大,静态分析工具报告的警报数量通常会显著上升,见表 1,从而导致人工确认不可行,降低了静态分析工具的可用性.

**Table 1** Result of applications scanned by Fortify

**表 1** 采用 Fortify 扫描实际程序的结果

| 程序 | 代码规模 | 总警报数 | 内存泄漏警报数 |
|---|---|---|---|
| Gzip-1.3.9 | 7 622 | 52 | 2 |
| Latd-1.30 | 9 194 | 273 | 6 |
| Libcgroup-0.37.1 | 16 104 | 351 | 46 |
| cgminer-4.3.4 | 37 859 | 1 005 | 69 |
| Flac-1.3.0 | 56 239 | 819 | 361 |
| Sendmail-8.12.4 | 99 196 | 5 246 | 115 |
| Git-1.7.2 | 117 607 | 4 621 | 328 |

混合执行测试[14,15]是一种将具体执行与符号执行结合起来的自动化测试方法,它能有效地提高测试自动

化程度.混合执行测试的主要思想是:在具体执行的过程中,同时进行符号执行和符号化路径约束条件的收集,利用约束求解产生测试输入,从而探索程序路径空间,查找代码缺陷.但是,混合执行测试没有将目标缺陷的先验知识作为指导,导致生成和执行大量无用的测试输入,浪费了时间和资源开销,影响了测试效率.

本文基于目标程序的源代码和静态分析工具报告的内存泄漏警报,提出了基于混合执行测试方法的内存泄漏静态分析警报自动确认方法,对静态分析得到的警报进行分类.本文方法根据静态内存泄漏警报:首先,在被测程序的控制流图上进行警报的可达性分析,计算程序中分支语句到路径片段中节点的可达性,得到路径制导信息;其次,以控制流图上静态内存泄漏警报的可能路径为覆盖目标,基于混合执行测试方法产生测试用例并执行;最后,追踪和监控运行时内存对象的状态,在测试执行结束时判断内存泄漏是否存在,从而完成对静态内存泄漏警报的确认和分类.

本文提出的方法一方面利用了静态分析技术得到较为全面的警报,静态分析警报提供的信息用于提高动态测试方法中测试用例的质量,从而降低动态执行的开销;另一方面,有效地减少了静态分析中的误报,得到更准确的警报.通过该方法可以对警报进行确认和分类,分为已经有证据可以确认的内存泄漏的误报和目前无法确认的警报,程序开发人员和测试人员可以对不同类别的静态警报采取不同的、有针对性的处理,比如仅对目前无法确认的警报进行人工确认、优先修复已确认的内存泄漏等.

本文第 1 节建立针对内存泄漏静态警报进行确认的分类系统,对警报的不同类别进行形式化的定义并予以说明.第 2 节主要介绍通过我们的内存泄漏静态分析结果进行分类或验证的核心工作,给出对静态分析的内存泄漏缺陷结果进行验证的基本框架和算法,并提出基于路径制导的混合执行的方法.第 3 节展示实验结果.第 4 节主要介绍相关工作,包括内存泄漏缺陷方面、测试生成方法以及对静态分析结果中消除误报方面的现有研究工作.最后,第 5 节进行总结与展望.

## 1 静态内存泄漏警报

在本节中,我们建立了对 C/C++语言程序内存泄漏静态警报进行确认的分类系统,对警报的不同类别进行形式化的定义并予以说明.

### 1.1 形式化定义

为便于描述本文方法,我们对 C/C++语言程序的抽象模型描述如下:

| | |
|---|---|
| Variables | $u,v\in V$ |
| Labels | $l\in L$ |
| Integers | $n\in I$ |
| Predicates | $\phi\in \Phi$ |
| Statements | $e\in E$ |

$e ::= u = {}^{l}v \mid u = {}^{l}malloc(n) \mid u = {}^{l}*v \mid *u = {}^{l}v \mid {}^{l}free(u) \mid e;e \mid \text{if } (\phi) \text{ then } {}^{l^T}e \text{ else } {}^{l^F}e$

该模型代表了 C/C++语言中内存操作的最简语句,其中,语句 $e$ 包含了赋值语句($u={}^{l}v$,$u={}^{l}*v$ 或$*u={}^{l}v$)、内存分配语句($u={}^{l}malloc(n)$)、内存释放语句(${}^{l}free(u)$)等.而其中,内存分配语句只是使用 $malloc(n)$代表一般情况(事实上还包含 $calloc(n)$,$realloc(n)$等多种情况).每个语句 $e$ 均有一个标签 $l\in L$ 用于标记和辨别该语句.在分支语句中,使用标签 $l^T$ 和 $l^F$ 来区分 True 分支和 False 分支.这也是静态分析工具报告内存泄漏警报中提供的有用输出信息.

利用以上模型,我们首先给出内存泄漏的静态分析警报的形式化描述.

**定义 1(内存泄漏警报)**. 内存泄漏的静态分析警报 $w$ 由三元组 $(a, p=\langle l_1^{b_1},l_2^{b_2},...,l_n^{b_n}\rangle, e)$ 构成,其中,$a,b,l_1,l_2,\ldots,l_n\in L$,且 $b_i\in\{T,F\}$.$a$ 和 $e$ 为语句标签,$p$ 为语句标签组成的路径片段.其中,$a$ 为内存分配语句($malloc$)的标签,而静态分析认定,由 $a$ 位置创建的内存对象在语句 $e$ 位置失去其所有引用.标签 $\langle l_1^{b_1},l_2^{b_2},...,l_n^{b_n}\rangle$ 为一系列程序分支的标签,它们与位置 $a$ 和 $e$ 一起构成了一条路径片段 $p$.

在验证内存泄漏时,我们使用了测试用例集,形式化定义如下.

**定义 2(验证内存泄漏的测试用例集)**. 给定静态分析警报 $w=(a,p=\langle l_1^{b_1},l_2^{b_2},...,l_n^{b_n}\rangle,e)$, 一个用于验证 $w$ 的测试用例集合 $S_w$ 包含了一系列的测试用例$(t_1,t_2,...,t_n)$,而其中每个测试用例 $t_i$ 对应的测试执行过程应满足一组路径约束条件$\varphi_1\wedge\varphi_2\wedge\ldots\wedge\varphi_n$,这里,若 $b_i=T$,则$\varphi_i=\phi_i$;反之,$\varphi_i=\neg\phi_i$.其中,$\phi_i$ 为 $l_i$ 处表示分支条件的谓词.

在这里,只要存在用例 $t\in S$ 能够在测试执行时覆盖到该程序路径,则称该用例集合是完备的.但是我们并不能保证 $S$ 的完备性,因为有可能无法找到合适的路径.

### 1.2 警报分类

利用我们提出的静态内存泄漏警报自动确认框架,可以将静态内存泄漏警报分为 4 个类别:MUST-LEAK, LIKELY-NOT-LEAK,BLOAT 和 MAY-LEAK.不同的类别对应不同的优先级,每种类别对应的内容如下.

- MUST-LEAK.我们有较高的把握认为它泄漏了,即,在动态执行的过程中存在测试用例使其达到 $e$ 点后确实没有释放申请的空间;
- LIKELY-NOT-LEAK.我们有较高的把握认为它并没有发生泄漏,即,在动态执行的过程中不存在测试用例能够造成内存泄漏.但是我们无法保证遍历了从 $a$ 到 $e$ 的所有路径,因此只能认为有较大可能性未发生泄漏;
- BLOAT.如果在动态执行的过程中证实:达到 $e$ 之前对内存进行了释放和处理,但在释放前很长一段时间内没有使用到该空间,则认为这里可以将释放步骤提前,对程序进一步优化;
- MAY-LEAK.当警报不属于以上 3 类情况时,我们将其划分到该类中,即没有证据来表明其是否泄漏,例如无法产生能够达到测试点的测试用例等.

在此基础上,我们对所有的分类进行一个形式化的定义.

假设,我们使用符号 $O_{a,t}$ 表示在测试用例 $t$ 中,在内存分配位置 $a$ 位置创建的内存集合.此处我们使用 3 个布尔函数来进行后续分类的定义:设布尔函数 $hasIR(o,e)$返回在疑似泄漏位置 $e$ 之后内存对象 $o$ 是否还有引用存在;布尔函数 $isUsed(o,e)$返回内存对象 $o$ 是否在位置 $e$ 之后被使用;布尔函数 $isFreed(o)$返回内存对象 $o$ 最终是否被释放.

根据内存泄漏的定义,满足如下条件($C_1$)则称为警报正确.

$$\exists t\in S_w:\exists o\in O_{a,t}:\neg hasIR(o,e) \quad (C_1)$$

$C_1$条件是最为理想的结果,它可以将所有的警报划分为两类:True 或者 False.但是 C/C++语言中的灵活性和复杂指针的使用,使得这样精确的划分十分困难,同时也加重了判断的开销.因此,我们使用了近似的条件($C_w$),即:判定在执行结束前,$a$ 所创建的内存对象是否被释放.

$$\exists t\in S_w:\exists o\in O_{a,t}:\neg isFreed(o) \quad (C_w)$$

这一条件比 $C_1$ 的判定更加容易,我们只需跟踪所有内存对象的状态即可,但是限定较之更弱.当警报满足 $C_w$ 时,可以判定发生了内存泄漏.但是内存对象有可能在该位置之后还有引用,并非在位置 $e$ 丢失了全部引用,而对于此种情况,也需要优先被修复.

根据以上分析,对 MUST-LEAK 的定义如下.

**定义 3(MUST-LEAK)**. 给定静态内存泄漏警报 $w$,若为其生成的测试用例集合 $S_w$ 满足条件 $C_w$,则将 $w$ 分类为 MUST-LEAK.

由于我们使用了较弱的限定和非完备的测试集合 $S$,当不满足条件 $C_w$ 时,并不代表误报,或许是由于我们未能生成有效的测试用例来覆盖内存泄漏的路径.毕竟在考虑别名的情况下,针对一般程序的全路径测试生成实际上是一个 NP 困难(NP-hard)问题[16],并不能保证测试用例可以覆盖全部情况.

因此,我们定义了类别 LIKELY-NOT-LEAK,当多次覆盖都未能产生泄漏时,则认为这个位置很可能是误报.同时,为了提高准确性,我们加入了条件 $C_s$.

$$\forall t\in S_w:\forall o\in O_{a,t}:isUsed(o,e)\wedge isFreed(o) \quad (C_s)$$

即,在位置 $e$ 之后是否进行了内存访问:如果满足该条件,说明静态警报对泄漏点的判断有误,这里极有可能是误报.

根据以上分析,对 LIKELY-NOT-LEAK 定义如下(其中,当测试集合的路径覆盖度越高时,该分类的可信度越高).

**定义 4(LIKELY-NOT-LEAK)**. 给定静态内存泄漏警报 $w$,若:(1) 路径约束条件$\varphi_1\wedge\varphi_2\wedge\ldots\wedge\varphi_n\equiv$false;或(2) 测试用例集合 $S_w$ 满足条件 $C_s$,则 $w$ 为 LIKELY-NOT-LEAK 警报.

但是有一些警报并不满足 LIKELY-NOT-LEAK 定义的后半部分,即:内存空间虽然在执行结束前得到了释放,但在位置 $e$ 之后再也没有被访问过.这些内存事实上并没有发生泄漏,但是程序员可以有选择地更早地释放它们.我们将这样的条件定义为条件 $C_b$.

$$\forall t\in S_w:(\forall o\in O_{a,t}:isFreed(o))\wedge(\exists o\in O_{a,t}:\neg isUsed(o,e)) \quad (C_b)$$

在此基础上,我们定义了 BLOAT 分类.

**定义 5(BLOAT)**. 给定静态内存泄漏警报 $w$,若其对应的测试用例集合 $S_w$ 满足条件 $C_b$,则 $w$ 为 BLOAT 类警报.

当警报不属于以上 3 类时,往往是由于程序无法生成需要的测试用例,即,找不到一条路径 $p$.这种情况下,程序无法判定是否发生了内存泄漏,需要人工进行确认.我们把这一分类定义为 MAY-LEAK.

**定义 6(MAY-LEAK)**. 给定静态内存泄漏警报 $w$,若满足:(1) 路径约束条件$\varphi_1\wedge\varphi_2\wedge\ldots\wedge\varphi_n\not\equiv$false;且(2) $S_w=\varnothing$,则 $w$ 为 MAY-LEAK 类警报.

表 2 显示了上述 4 类警报的验证优先级和修复优先级,其中,1 为最高,4 为最低.对于验证而言,MAY-LEAK 是程序获得信息最少的一类,最需要人工进行验证,而其他几类或多或少都有一定的证据表明泄漏/未泄漏.根据程序的确信程度,MUST-LEAK 拥有最高的可信度,其次是 LIKELY-NOT-LEAK.对于修复而言,只要产生了内存泄漏都是需要修复的,而 MUST-LEAK 的部分通常确实发生了内存泄漏,是首先需要进行修复的,LIKELY-NOT-LEAK 和 BLOAT 分类在动态执行的过程中实际发现了内存释放点,因此有较大可能并没有发生内存泄漏,对程序影响不大,因此修复优先级较低.

**Table 2** Priority comparisons among the four categories

**表 2** 4 种分类的优先级比较

| 分类 | MUST-LEAK | MAY-LEAK | BLOAT | LIKELY-NOT-LEAK |
|---|---|---|---|---|
| 验证优先级 | 4 | 1 | 2 | 3 |
| 修复优先级 | 1 | 2 | 3 | 4 |

## 2 C/C++程序静态内存泄漏警报自动确认方法

本节我们提出 C/C++程序的静态内存泄漏警报自动确认方法的基本架构,详细描述了对内存泄漏静态警报进行动态确认以及分类的具体方法.在此基础上,我们提出路径制导的混合执行测试方法,用于动态地检测内存泄漏情况.

### 2.1 方法架构

图 1 显示了该方法的整体架构,虚框内为本文工作.我们的方法的输入是静态分析工具报告的警报及目标程序源代码.根据定义 1,程序需要规范静态警报信息,并在源代码中进行标记.利用警报 $w=(a,p,e)$在源代码中标记分配位置 $a$、片段 $p$ 以及泄漏位置 $e$.本文方法主要分为 3 步:第一,静态内存泄漏警报的可达性分析,利用目标代码得到程序的控制流图,结合警报信息对所有需要的路径进行可达性分析,得到制导信息用于指导后续的混合执行测试;第二,进行路径制导的混合执行测试,自动覆盖可达路径;第三,在混合执行测试过程中动态追踪监测内存块的状态,例如申请和释放等状态,依据内存状态机判定运行时内存泄漏是否发生,从而对警报进行确认和分类.

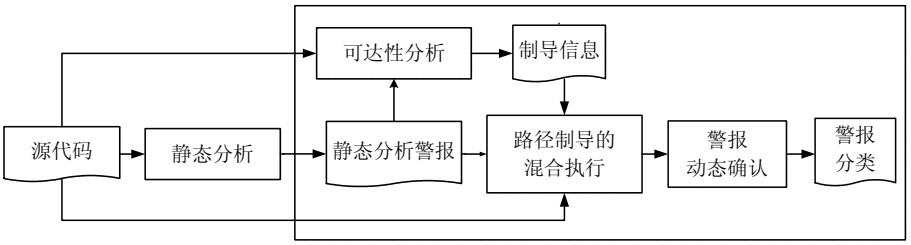



Fig.1 Framework of our work

图 1 方法架构

### 2.2 警报可达性分析

静态分析警报由目标程序中内存分配位置、疑似泄漏点以及从分配位置到疑似泄漏点的路径片段组成.要想确认警报的正确性,需要在运行时找到内存泄漏发生的证据.由于程序中可能的循环、递归与分支的组合可能会产生路径爆炸,若能够只执行与警报相关的所有可执行路径,则会大大提高效率.我们将静态警报路径片段中的语句标签映射到控制流图的基本块中,路径片段所标注的控制流片段即为我们需要验证的目标路径.这里并非所有的程序路径都能覆盖目标路径,因此我们使用了可达性分析,忽略掉与目标路径无关的分支,尽可能地生成能够覆盖目标路径的测试用例,以降低测试的开销.

对于给定的静态警报的路径片段 $p$:$\langle l_0,l_1,...,l_{k1}\rangle$,可达性分析的目的是建立程序中的路径分支到一组布尔值之间的映射关系 $m$:$\langle l,\langle b_0,b_1,...,b_{k1}\rangle\rangle$.$l$ 为程序中的某一分支语句的 True 或 False 分支的标签,$\langle b_0,b_1,...,b_{k1}\rangle$为一组布尔值.其中,布尔值 $b_i$ 对应了程序分支 $l$ 到某一静态警报的路径片段 $p$ 的节点 $l_i$ 是否可达.

为了建立该可达性映射,可达性分析的过程需要如下步骤:首先建立控制流图,标记警报位置,初始化所有分支对应路径片段节点的布尔值为 False;之后进行反向遍历,对控制流图建立反向图,以静态警报路径片段的节点位置 $l_i$ 为起点,在反向图中进行遍历;对于遍历中遇到的程序分支 $l$,将该 $l$ 所对应映射关系中的布尔值 $b_i$ 赋值为 True,更新可达性映射结果.

这里使用了保守的判定方法,可达性分析只是表明路径存在的可能性,并不代表每次执行必然可达,凡具有路径存在的可能性,则可达性结果为 True.可达性分析的结果用于加快混合执行测试的空间探索过程,当可达性结果为 False 时,混合执行测试中对路径分支 $l_i$ 的约束条件的取反求解操作将不会被执行,从而忽略无用分支,使用可达性分析可以有效地加快混合执行测试的速度.

### 2.3 路径制导的混合执行测试

我们通过静态内存泄漏警报在控制流图上的可达性分析,获取目标路径.在测试执行过程中,仅执行目标路径能够有效地确认静态分析的警报.我们采用了路径制导的混合执行测试方法来遍历程序并覆盖目标路径,动态地记录和追踪警报中的内存对象的状态,以验证警报.在混合执行测试之前,需要先对目标程序进行插装,用于追踪和监测内存对象的信息和状态.插装需要遍历目标代码,在内存操作的函数后插入内存追踪相关的函数,这里涉及到的内存仅包括静态分析中标记的内存对象,例如在分配内存(*malloc*/*calloc*/*realloc* 等)函数后面插入新函数用于新建该内存对象的维护信息,在疑似内存泄漏点后面插入泄漏标记函数等.同时,混合执行需要在初始时指出输入相关的变量,并手动地将变量符号化,以方便后续产生混合执行的测试输入.

算法 1 阐述了路径制导的混合执行测试方法.该算法的输入是静态警报的路径片段 $p$,即上文所述的内存申请位置到疑似泄漏位置的路径,作为制导的条件;另一个输入是通过可达性分析得到的映射关系 $m$,$m$ 为程序中的分支 $l$ 到布尔值向量$\langle b_0,b_1,...,b_{k1}\rangle$之间建立的映射关系.布尔值 $b_i$ 表示的是分支 $l$ 是否可能在程序执行时到达路径片段 $p$ 的第 $i$ 个分支标签 $l_i$.其中,静态警报的路径片段应该默认为可达的.因此在这样的条件下,若 $l$ 到 $l_i$ 可达,则到其后续分支标签 $l_{i+1},l_{i+2},...,l_{k1}$ 也应可达.

该算法的基本思想是:利用 $m$ 在混合执行测试时优化路径空间探索,达到路径制导的目的,同时减少运行开

销.在一般的混合执行测试过程中,通常会对所有路径的所有分支都进行取反的操作和约束求解的过程,而在我们的算法中,通过 $m$ 来削减与所需要的路径无关的操作.这一过程在最坏的情况下将会不采取任何削减操作,而直接使用一般的混合执行测试方法.

首先初始化两个空的集合 $B$ 和 $C$(第 1 行、第 2 行),用于记录执行过程中覆盖的路径分支和分支对应的约束条件;算法将会以一个初始的输入 $I$(第 3 行)来动态地执行目标程序(第 4 行),例如字符串初始为空、基本类型变量初始为 0,或者在已知变量范围时取最大值或最小值.在此过程中,将会维护 $B$,$C$ 两个集合,利用集合 $C$ 进行约束求解,可以得到该路径对应的条件和输入,即该路径对应的测试用例.对集合 $C$ 的分支约束进行取反,可以得到新的路径和对应的输入.原混合执行测试算法[14,15]通过依次对每一分支的约束条件进行取反,从而与其他分支约束条件组成了一个新的路径约束条件.在算法中,我们希望利用 $m$ 对其进行一定程度的优化.

通过 *PathGuidedSearch* 生成并求解新的路径约束(第 5 行).在每次循环中(第 10 行~第 23 行),检查分支标签 $B[i]$的相反分支 $\overline{B[i]}$ (例如分支语句的 True 分支与 False 分支互为相反分支)在映射 $m$ 中记录的可达性(第 12 行),以此来忽略与 $p$ 无关的分支约束条件.

变量 *count* 记录了路径片段 $p$ 目前被覆盖到的分支标签(第 20 行~第 22 行).表达式 $m(\overline{B[i]})[count]$ 返回 1,表明分支 $\overline{B[i]}$ 是否到 $p$ 上的第 *count* 个分支标签(即 $l_{count}$)可达.这里,只要分支与路径片段 $p$ 有可能存在可达性关系(该值为 True),都会对相应的符号约束条件 $C[i]$取反,并形成新的路径约束条件(第 15 行).若 $B[i]$与 $l_i$ 为相同的分支标签,*count* 自增 1.在下一次迭代中,我们将检查分支 $\overline{B[i+1]}$ 到分支标签 $l_{i+1}$ 的可达性.若某一次迭代中路径片段 $p$ 的所有分支 $l_0,l_1,...,l_{k1}$ 已经达到覆盖(即 *count*=$k$),则此后的迭代过程不再检查可达性关系.当所有警报都已得到确认或者所有可达的路径都已被覆盖后,终止测试.

**算法 1**. 路径制导的混合执行测试方法.

输入:路径片段 $p$:$\langle l_0,l_1,...,l_{k1}\rangle$,

可达性映射 $m$:$\langle l,\langle b_0,b_1,...,b_{k1}\rangle\rangle$;

```
Begin
1: 集合 B 用于存储程序分支标签,初始化为空
2: 集合 C 用于存储分支的符号约束,初始化为空
3: 初始化程序输入 I
4: 以初始输入 I 驱动混合执行测试,Run(I,&B,&C)            执行过程中记录 B,C
5: 递归混合执行测试及测试生成,PathGuidedSearch(B,C)
6: PathGuidedSearch(Array B,Array C)
7:    int count=0
8:    bool reachable=TRUE
9:    int n=length(B)1
10:   for (i=0;i<n;i++)
11:      if (count<k)
12:         reachable= m(B[i])[count]
13:      end if
14:      if (reachable==k||count==k)
15:         I'=solve(C[1]∧C[2]∧…∧C[i−1]∧¬C[i])          //利用约束条件集合 C 求解输入 I'
16:         初始化新的集合 B',C'
17:         Run(I',&B',&C')重新执行程序
18:         PathGuidedSearch(B',C')递归
19:      end if
20:      if (B[i]==l_count)
```

```
21:          count++
22:       end if
23:    end for
End
```

### 2.4 警报动态确认

本文通过追踪和更新目标内存的状态作为最终警报确认和分类的依据.当目标内存分配时,则为其新建一块空间,用于存储和维护该内存对象的状态信息.目前,通常使用两种技术来维护这一状态信息:一种是创建影子堆[17−19],完全投影堆空间,对于堆上的每一块内存空间,都有一段大小相同且具有相同偏移位置的影子空间来维护状态信息,但是影子空间的大小完全等于堆空间的大小,使其具有较大开销.另一种方法是利用哈希表来建立映射关系[20].这里,我们采用了后一种方法.

图 2 是我们在进行内存追踪操作时所使用的状态机模型.内存被分配时的初始状态为 Created,如果执行过程中到达了内存泄漏点,则状态转换为 LP(即,达到 LEAK POINT),在此之后,如果对该内存进行了访问,状态即变为 UseAfterLP.在该状态机中存在 3 种释放状态:第 1 种释放状态(Freed1)代表并没有到达泄漏点就释放了,这种情况基本不存在;第 2 种(Freed2)表示在泄漏点后没有进行访问,但是最终释放了内存,对应于 BLOAT 分类;第 3 种(Freed3)在泄漏点后进行了访问,最终也对内存进行了释放操作,对应于 LIKELY-NOT-LEAK 分类.

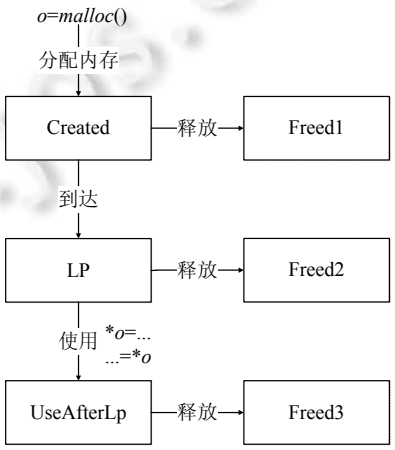



Fig.2 State machine of a tracked object

图 2 内存追踪的状态机图

通过该状态机,可以得到第 1.2 节中两个布尔函数的返回值.若最终达到了任意一个释放状态,则 *isFreed*(*o*)返回 TRUE;否则返回 FALSE.如果经过了 UseAfterLP,则 *isUsed*(*o*,*e*)返回 TRUE;否则,返回 FALSE.得到布尔函数的返回值后,利用第 1.2 节的定义对警报进行分类,即可完成警报的动态确认过程,并对警报进行分类.

## 3 工具与评估

为了检验本文提出的方法,我们实现了基于混合执行测试的静态内存泄漏警报自动确认工具.在此基础上,我们针对若干基准程序进行了实验研究,并对实验数据进行分析.本文涉及到的工具实现、实验对象、实验结果的详情,参见 http://ssthappy.github.io/memleak/.

### 3.1 工具实现

在实验中,我们使用 HP 的 Fortify[13]作为静态分析的工具.Fortify 在业界使用较广,同时,Fortify 给出的静态分析结果使用了 XML 格式,便于对警报的格式化和处理.我们基于 CIL 环境、测试生成和混合执行测试的工具

CREST[21]、约束求解器 Yices[22],实现了基于混合执行测试的静态内存泄漏警报自动确认工具,支持控制流图分析、可达分析、内存跟踪、插装、路径制导和约束求解、警报确认、警报分类等.

### 3.2 实验对象与设计

针对静态内存泄漏警报的动态确认,我们期望通过实验来考察和回答如下几个研究问题.

- Q1:该静态内存泄漏警报自动确认方法的准确性如何?
- Q2:该自动确认方法能够节省多少人工确认的工作量?
- Q3:该自动确认方法应用于具有一定规模的实际程序时效果如何?
- Q4:该方法的执行效率如何?对比动态测试技术开销如何?

针对上述问题,我们设计了 3 组实验:第 1 组用于测试该方法的准确性,第 2 组用于验证在一定规模的程序中的实验效果,第 3 组用于评估该方法对比动态测试技术的效率.

第 1 组和第 3 组实验来自基准程序集 Siemens[23]和 coreutils[24],共有 10 个程序,见表 3.其中,上标 1 代表来自 Siemens,上标 2 代表来自 coreutils.

**Table 3** Experiment 1 & Experiment 3: Subjects

**表 3** 实验 1 和实验 3:对象

| 基准程序 | 代码行数 | 说明 |
|---|---|---|
| replace[1] | 563 | 替换 |
| print_tokens[1] | 726 | 词法分析 |
| print_tokens[1] | 569 | 词法分析 |
| tcas[1] | 173 | 防冲突 |
| wc[2] | 802 | 打印,统计字节数 |
| cat[2] | 785 | 拼接、读写文件 |
| head[2] | 1 063 | 输出文件的第1部分 |
| tr[2] | 1 949 | 替换或者删除字符 |
| expand[2] | 433 | 将制表符转换为空格 |
| unexpand[2] | 535 | 将空格转换为制表符 |

这些实验对象的规模都比较小,以用于验证本文提出的方法的准确性和效率.另外,为了增加静态分析警报的数目,我们对这些实验对象进行了内存泄漏植入,比如建立全局的数据结构存储动态分配的内存指针,或者使用二级指针操作.对于这两种情况,Fortify 通常会误报.我们同时植入了真实内存泄漏和可能导致静态分析误报两种情况,对于误报,我们确保在程序结束之前释放内存,错误植入的位置是随机决定的.在植入的时候,已经知道了这些警报的真实分类,这有助于我们更快地判断动态验证的结果是否正确.

第 2 组实验对象是 texinfo-4.13,其规模较大.在这一组实验中,没有植入内存泄漏,所有的警报均为原始结果.

我们所使用的实验环境为 Linux 3.2.0 系统,配置了英特尔酷睿 i5-3360M 2.8GHz 处理器的机器,Fortify 的版本为 3.2.

### 3.3 实验数据及分析

#### 3.3.1 实验 1

表 4 展示了利用本文方法对第 1 组基准程序进行分析的实验数据.

表 4 中,第 1 部分(a)显示了每个基准程序的代码行数(#*LOC*)和 Fortify 报告的内存泄漏警报数目(#*W*),而#*S* 是覆盖到目标路径的测试执行次数.第 2 部分(b)记录了分类结果:MUST-LEAK(#*Must*),LIKELY-NOT-LEAK (#*LNL*),BLOAT(#*B*)和 MAY-LEAK(#*May*).第 3 部分(c)中,集合 *T* 代表了真正的内存泄漏的数目(包括原程序中的泄漏和植入的泄漏),#(*LNL*∩*T*)和#(*B*∩*T*)则分别表示了正确内存泄漏警报被错误划分到类别 LIKELY-NOT-LEAK 和 BLOAT 中的数目,即验证错误的部分.第 4 部分(d)记录了测试执行中的时空开销,$T_0$ 和 $Sp_0$ 代表目标程序正常执行时(即未进行插装)的时空开销,而 $T_1$ 和 $Sp_1$ 代表混合执行测试中插装后的时空开销.

**Table 4** Experiment 1: Categorized warnings

**表 4** 实验 1:警报分类结果

| 基准程序 | (a) 基本信息 | | | (b) 静态警报分类情况 | | | | (c) 错误的分类 | | (d) 性能 | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | *#LOC* | *#W* | *#S* | *#Must* | *#LNL* | *#B* | *#May* | *#(LNL∩T)* | *#(B∩T)* | $T_0(s)$ | $T_1(s)$ | $Sp_0$(MB) | $Sp_1$(MB) |
| replace | 563 | 18 | 3 444 | 5 | 3 | 4 | 6 | 0 | 0 | 3.82 | 3.95 | 16.4 | 20.4 |
| print_tokens | 726 | 22 | 17 383 | 8 | 4 | 6 | 4 | 0 | 0 | 3.7 | 5.0 | 16.9 | 20.6 |
| print_tokens2 | 569 | 29 | 16 943 | 8 | 7 | 9 | 5 | 0 | 0 | 4.2 | 4.7 | 22.1 | 22.2 |
| tcas | 173 | 8 | 54 | 1 | 4 | 1 | 2 | 0 | 0 | 0.06 | 0.07 | 15.6 | 19.8 |
| wc | 802 | 8 | 6 000 | 2 | 2 | 2 | 2 | 0 | 0 | 10.5 | 15.5 | 17.1 | 22.8 |
| cat | 785 | 8 | 4 002 | 2 | 1 | 2 | 3 | 0 | 0 | 16.9 | 26.2 | 15.7 | 19.9 |
| head | 1 063 | 18 | 5 007 | 4 | 6 | 2 | 6 | 0 | 0 | 12.2 | 17.4 | 16.3 | 20.5 |
| tr | 1 949 | 32 | 37 281 | 11 | 8 | 8 | 5 | 0 | 0 | 21.2 | 26.5 | 16.8 | 21.2 |
| expand | 433 | 6 | 3 854 | 1 | 1 | 2 | 2 | 0 | 0 | 22.3 | 26.9 | 21.8 | 25.9 |
| unexpand | 535 | 6 | 3 996 | 1 | 1 | 2 | 2 | 0 | 0 | 25.7 | 26.2 | 23.1 | 27.2 |

根据以上结果我们可以得到如下几个结论.

**结论 1**. 表 4 中,(c)部分均为 0.这表明:在我们的实验中,并没有将确实发生内存泄漏的警报划分到误报的分类(LIKELY-NOT-LEAK 和 BLOAT)中.说明 LIKELY-NOT-LEAK 和 BLOAT 分类的警报有极大可能是误报,开发人员可以降低它们的修复和验证优先级.

**结论 2**. 基准程序中一共含有 155 个静态内存泄漏警报,表 4 中 MAY-LEAK 类别的警报数仅为 37.通过本文方法的分类,需要人工确认的警报数目下降了 76.1%,大大减轻了人工验证的负担.

**结论 3**. 表 4 中,(d)部分所示插装/不插装性能差距不大,表明我们的方法并没有引入过多的时空开销,大约引入了 24.8%的时间开销和 21.4%的空间开销.当然,这主要得益于使用了近似条件 $C_w$,使用 $C_1$ 条件往往额外开销很大,甚至是百倍的差距.

3.3.2 实验 2

第 2 组实验测试规模较大的程序,以评估本文方法的可扩展性.Texinfo 是一个文本格式转换程序,可以将一定格式的源文件转换为其他输出格式,比如 html 以及 pdf 等.Texinfo 一共有 46 493 行代码.该程序的输入为特定格式的输入文本和转换需要的命令行参数,而我们无法让混合执行测试技术自动地产生如此复杂的输入数据,因此提前创建了 texinfo 可接受的输入文件作为测试输入的一部分,而参数部分采用 CREST 自动生成.创建这些输入文件大约花费了一周半的时间,这主要是混合执行测试工具能力的不足造成的,并非本文方法的限制.

Fortify 共报告了 91 个静态内存泄漏警报,其中有 21 个划分为 MAY-LEAK,而其余 70 个都进行了有效的分类,包含 69 个 MUST-LEAK、1 个 LIKELY-NOT-LEAK 和 0 个 BLOAT,分类结果主要受限于依赖工具的能力和约束求解的能力.我们对这些警报进行了逐一排查,59 个警报符合了条件 $C_1$,确实在泄漏点后丢失了所有的引用;而另外 10 个则是全局指针或者数据结构,在泄漏点后还有引用.我们也进一步确认了分类结果中的 LIKELY-NOT-LEAK 确实是一个误报.尽管有诸多限制,本文的方法依然使得人工验证的警报数目降低了 76.9%,内存追踪所引入的时间和空间开销分别为 77.5%和 26.4%.

3.3.3 实验 3

第 3 组实验对比了本文方法与动态测试技术的效率和效果,这里,动态测试工具选用 Valgrind[25].Valgrind 在业界应用十分广泛,尤其是在内存泄漏方面的检测.表 5 展示了对比实验的结果.

在表 5 中,(a)部分所示基本信息包含了每个基准程序的行数(*#LOC*)以及产生泄漏的内存对象的个数(*#LP*,简称为泄漏点),这里,*#LP* 与实验 1 中的*#W* 不同,不同的警报可能是由相同的内存对象所造成的,只是其路径片段不同.(b)部分中展示了本文方法能找出的泄漏点个数($\#LP_0$,即 MUST-LEAK 分类)以及动态测试能够找到的泄漏点个数($\#LP_1$).(c)部分测试用例记录了本文方法覆盖到目标路径的测试执行次数(*#S*)以及动态测试时使用的测试用例数量(*#TC*).(d)部分中分别记录了两种方法的时空开销,$T_0$ 和 $SP_0$ 为本文方法的时空开销,这里包含了静态分析阶段的开销,时间为静态分析和动态确认过程的总和,内存以/隔开;$T_1$ 和 $SP_1$ 为 Valgrind 的时空开销(这里未包含测试用例生成的时间).

动态测试实验中,需要额外给定测试用例.这里,Siemens[23]中的程序使用了其自带的测试用例,其余程序使

用 KLEE[26]自动生成测试用例.这使得 coreutils[24]中基准程序的空间开销包含了 KLEE 的部分,实际内存开销与 Siemens 中基准程序近似.

**Table 5** Experiment 3: Contrast experiment with Valgrind[25]

**表 5** 实验 3:Valgrind[25]对比实验

| 基准程序 | (a) 基本信息 | | (b) 检测结果 | | (c) 测试用例 | | (d) 性能 | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | *#LOC* | *#LP* | $\#LP_0$ | $\#LP_1$ | *#S* | *#TC* | $T_0(s)$ | $T_1(s)$ | $SP_0$(MB) | $SP_1$(MB) |
| replace | 563 | 7 | 5 | 7 | 3 444 | 5 542 | 30.77 | 4 968.12 | 549.9/20.4 | 28.9 |
| print_tokens | 726 | 4 | 4 | 4 | 17 383 | 4 130 | 32.16 | 3 737.74 | 545/20.6 | 29.3 |
| print_tokens2 | 569 | 2 | 2 | 2 | 16 943 | 4 115 | 30 | 3 735.83 | 551.1/22.2 | 27.9 |
| tcas | 173 | 2 | 1 | 2 | 54 | 1 608 | 24.53 | 1 424.58 | 552.6/19.8 | 27.9 |
| wc | 802 | 1 | 1 | 1 | 6 000 | 835 | 48.44 | 1 807 | 565.7/22.8 | 80.9 |
| cat | 785 | 2 | 2 | 2 | 4 002 | 123 | 57.99 | 262 | 567.6/19.9 | 79.8 |
| head | 1 063 | 4 | 4 | 4 | 5 007 | 909 | 50.97 | 2 002 | 540.9/20.5 | 80.9 |
| tr | 1 949 | 6 | 6 | 5 | 37 281 | 1 593 | 64.4 | 3 430 | 565.5/21.2 | 80.7 |
| expand | 433 | 2 | 1 | 2 | 3 854 | 2 578 | 55.95 | 5 965 | 544.4/25.9 | 83.5 |
| unexpand | 535 | 2 | 1 | 2 | 3 996 | 1 560 | 56.09 | 3 435 | 544.8/27.2 | 83.1 |

根据表中的结果,我们可以得到如下结论:首先,两种方法都具有较高的查全率,但是动态测试技术的查全率依赖测试用例集的质量和覆盖度,且自动生成测试用例需要花费较多时间,甚至超过 48 小时,并且随着程序规模的扩大,生成测试用例时间呈指数上升,例如 tr 中有 6 个内存泄漏点,但是 valgrind 只能找到其中的 5 个,说明生成的测试用例不足以覆盖剩余的一个泄漏点对应的路径;其次,本文方法的时空开销远远小于动态测试技术,由于动态测试技术常在执行时动态插装并且追踪,会造成较大的时空开销,在运行测试用例时,时间开销可能会是普通运行的百倍甚至千倍;最后,事实上,我们在性能和分类的准确性之间找到了一个平衡,较大地提升了性能而稍微降低了分类的准确性,依然可以得到较好的检测效果.

利用 Siemens 程序自带的测试用例集合,我们进行了附加实验,将混合执行产生的测试用例替换为自带的高覆盖度测试用例,得到表 6 所示的结果.

表格中,第 1 部分(a)显示了每个基准程序的代码行数(*#LOC*)和 Fortify 报告的内存泄漏警报数目(*#W*),而 *#TC* 是测试用例的个数.第 2 部分(b)记录了分类结果:MUST-LEAK(*#Must*),LIKELY-NOT-LEAK(*#LNL*),BLOAT(*#B*)和 MAY-LEAK(*#May*).第 3 部分(c)中,集合 *T* 代表真正的内存泄漏的数目(包括原程序中的泄漏和植入的泄漏),$\#(LNL\cap T)$和$\#(B\cap T)$则分别表示正确的内存泄漏警报被错误划分到类别 LIKELY-NOT-LEAK 和 BLOAT 中的数目,即验证错误的部分.第 4 部分(d)记录了测试执行中的时空开销,*T* 和 *Sp* 代表执行测试用例的总时空开销.

**Table 6** Experiment 3: Executing results of test cases

**表 6** 实验 3:测试用例执行结果

| 基准程序 | (a) 基本信息 | | | (b) 静态警报分类情况 | | | | (c) 错误的分类 | | (d) 性能 | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | *#LOC* | *#W* | *#TC* | *#Must* | *#LNL* | *#B* | *#May* | $\#(LNL\cap T)$ | $\#(B\cap T)$ | *T*(*s*) | *Sp*(MB) |
| replace | 563 | 18 | 5 542 | 5 | 4 | 4 | 5 | 0 | 0 | 1 142.9 | 10.8 |
| print_tokens | 726 | 22 | 4 130 | 8 | 4 | 6 | 4 | 0 | 0 | 754.63 | 11.1 |
| print_tokens2 | 569 | 29 | 4 115 | 9 | 0 | 16 | 4 | 0 | 0 | 593.43 | 9.9 |

在该实验中,根据静态分析的结果对程序进行插装,利用给定的高覆盖度测试用例集合执行插装后的程序,对警报进行分类.根据表中的结果可以看出:首先,不存在分类错误的情况,print_tokens2 中虽然将部分 LIKELY-NOT-LEAK 分到了 BLOAT 分类,但事实上两者都是未泄露,并不是错误分类;其次,在测试用例集合覆盖度较高的情况下,对比实验 1 能够得到更好的测试结果,即降低了 MAY-LEAK 的比例,我们可以认为,在混合执行技术不断完善的情况下,本文所提出的方法可以得到更好的效果;最后,该实验与 Valgrind 使用了相同的测试集,但是时空开销远小于 Valgrind,说明本方法具有更高的效率.该实验进一步佐证了本文中动静态结合的方式具有较高的实用性和效率.

### 3.4 讨 论

本节将对实验的结果作进一步的讨论.

首先,本文方法确实有助于减少人工确认警报的时间和精力.对于具有有效分类的警报,准确率较高.能够在混合执行测试中覆盖到的警报,通常都能给出正确的分类,说明本文方法具有较高的准确率和实用性.

其次,本文方法所引入的时空开销并不显著.对比于现有的一些工作,比如 LeakPoint[1]所采用的指针追踪技术引入了 100~300 倍开销,以及上述实验中的 Valgrind,本文方法已经具有较高的性能和效率.事实上,我们是在性能和准确性之间寻找到一个平衡,采用近似的条件显著提高了应用时的效率但是稍微降低了准确率,当然,我们认为这个平衡是值得的.

最后,在扩展性方面,我们通过实验 2 进行了评估.在该实验中,并没有发生将正确的警报划分为误报分类的情况,我们认为,其可以扩展到具有一定规模的程序中.而在时间开销上,由于执行过程中需要追踪内存的状态,则方法需要在目标内存相关的地方插装,因此程序规模越大,插装的内容越多,额外的时间开销越大.同时,随着程序规模的扩大,路径的求解变得更加复杂,求解的时间也会随之上升.程序规模对本方法的限制主要在于混合执行工具的能力和求解器的能力以及路径的覆盖率.随着相关技术的发展,比如约束求解器和混合执行测试工具能力的提升,其扩展能力和准确率也会有一定的提升.

综上所述,我们的实验结果表明:利用动态测试和运行时分析的技术对静态内存泄漏警报进行分类,提供了一种有效的验证和确认静态分析警报的方法,从而使得在实际应用中,静态内存泄漏检测工具变得更加实用和有效.

除此之外,实验中还具有一些缺陷值得注意.首先,我们仅在有限数目的基准程序上评估了该分类系统的准确性和有效性,因此,我们并不能保证该方法对所有实用程序都具有绝对的准确性,但是我们依然相信,实验的结果确实表明了采用本文方法的可行性以及该方法对各类实用 C/C++程序的适用性.另外一个问题是,实验中我们采用了 MAY-LEAK 的数据来衡量减少的人力成本,但是实验并未说明警报数目的减少如何影响开发人员消耗的时间以及验证警报的难度.因此在后续的工作中,我们也将寻找更好的方法来衡量所减少的人力成本.

## 4 相关工作

在本节中,我们主要通过以下 3 个方面来介绍和讨论相关的研究工作.第一,内存泄漏的静态分析和动态检测方法;第二,测试生成方法;第三,目前已有的针对静态分析结果中的消除误报的相关工作.

### 4.1 内存泄漏检测

内存泄漏在内存的缺陷中十分常见,通常是由于程序实现中对动态内存的管理不当造成的.当程序中动态分配的内存没有得到及时的释放时将会产生泄漏,会使得内存空间被消耗且无法被回收和重新利用,导致内存资源减少,最终降低了程序性能,尤其是对于长期运行的程序来说,影响十分显著.内存泄漏具有一定的隐蔽性,极易被忽略,尤其是对于 C/C++这类显式内存管理的语言,随着规模的扩大和程序员的不规范编程,内存泄漏的情况更加常见.

对于内存泄漏的检测方法,目前常用两种方式:一种是静态分析,另一种是动态测试.

静态分析被广泛地应用于 C/C++程序中内存管理的相关问题,比如内存泄漏和重复释放等.静态分析工具的原理通常是查找和分析特定的错误模式,例如缺少释放空间的步骤,或者建立描述内存状态的模型来进行缺陷的判定.由于软件十分复杂,严格的匹配会造成漏报,而模糊的匹配会造成误报.学术界一直在这之中寻找平衡,以期在降低误报率的情况下找到更多的缺陷.Cherem 等人[6]将其转化为一个可达性问题,利用分析数据流图上的数值传播来检查是否发生了内存泄露.内存泄漏检测工具 Clouseau[7]使用指针关系模型查找应该释放内存的变量,形成一个约束系统.Heine 和 Lam[8]同样是基于内存对象的隶属模型,描述容器中的内存对象间的隶属关系(支持多态),进而通过类型检查是否有违反约束的内存泄漏或重复释放等问题.Orlovich 和 Rugina 等人[9]使用反证法,首先假定缺陷存在,然后采用反向数据流分析来否定该假设,从而判定是否泄漏.Saturn[10]框架将其简化为一个布尔可满足性问题,并使用 SAT 求解器来判定潜在的错误.文献[27]的算法是一种基于三值(3-valued)逻辑的形状分析(shape analysis),这种分析可以证明链表操作中不存在内存泄漏缺陷.Hackett 和 Rugina 等人[28]提出的形状分析算法可以追踪单个堆单元中的内存位置,从而判别内存是否泄漏.Saber[29,30]通过

分析稀疏值流图(SVFG)中指针引用的传递和值传递来判定内存泄漏.Melton[31]提出了一种过程间分析的算法,利用符号执行生成内存状态转换图(MSTG),在分析过程中监测内存泄漏.除了学术界,许多商业工具,如Klocwork[11],Coverity[12]和 Fortify[13]等在实际的软件开发中使用广泛.然而,静态分析工具依旧受限于规模,虽然已有大量针对全局、可扩展的内存分析的研究工作,如何将静态分析应用到大规模的软件上依然是一个受到广泛关注的问题.本文提出的方法在现有的静态分析方法的基础上对静态分析的警报进行验证和分类,以期提高静态分析工具的实用性和查准率.

内存泄漏的动态测试工具则是在程序执行的过程中,根据内存的实际变化来判断泄漏是否发生.动态测试能够准确地检测某一运行过程中是否发生了内存泄漏,同时能够准确地找出泄漏点,但是它也存在一些问题.例如,动态检测需要通过插装或者外部监测的方法对内存情况进行监测和控制,具有较大的开销,对设备硬件要求较高.除此之外,动态检测工具只能检测输入路径的内存情况,对输入的测试用例的质量和覆盖度要求很高,对于覆盖不到的路径很可能产生漏报,因此,动态测试技术依赖较高质量的测试用例.目前,学术界也有许多针对C/C++的内存泄漏动态检测工作.LeakPoint[1]基于污点传播的方法来监测内存,追踪内存最后使用的位置以及失去引用的位置.Purify[2]采用插装的方式来跟踪内存的使用情况,在目标代码中插入进行内存追踪的语句,是动态测试技术最早的思想基础.Omega[3]和 Maebe[6]主要采用指针计数的思想记录内存对象的引用计数.SafeMem[5]利用纠错内存(ECC memory)开发了内存使用情况分析技术,可以降低监测的开销.Valgrind[32]采用了动态插装技术,即在执行时进行插装.Sniper[33]主要利用处理器中的性能监测单元(performance monitoring unit,简称 PMU)取样,在运行时监测堆的状态,提出了基于 staleness 的内存泄漏检测技术.在此基础上,文献[34]对堆进行抽象建模,并使用机器学习来检测内存泄漏.另外,动态测试技术在 Java 等托管语言中的内存泄漏检测方面也得到了很多应用[35−46].

本文的工作主要有两个方面与上述相关工作不同:第一,本文工作的目标在于确认静态分析的警报,与任意静态分析技术互补;第二,本文进行警报确认的混合执行测试,利用了可达警报路径制导,减少了内存泄漏无关路径的测试执行,与其他动态测试技术相比,降低了开销,提高了效率.

文献[47]对内存泄漏进行了自动修复,通过对程序中指针、控制流、数据流等的分析,在检测内存泄漏的同时尝试安全地修复泄漏点.在这类工作中,能够成功修复的部分往往代表正确的检测.但是这类工作修复过程较为保守,能够修复的部分并不全面,可能造成漏报或无法修复的情况.其中,无法修复的情况无法确认其正确性.而我们的工作基于已有静态分析工具给出的警报,对所有情况进行分析,不仅针对正确的警报,同时也针对误报的警报以及 BLOAT 类型,给出准确的分类.

### 4.2 测试生成方法

测试生成在软件测试中是十分重要的组成部分,生成高质量的测试用例一直是学术界和业界的关注热点.

将代码抽象为模型,则运行程序的过程就是对模型的具体化的过程.

对程序的测试,可以使用不同的具体数值对模型进行实例化,以检验在任何输入情况下是否都可以得到符合预期的结果.这样的方法可称为具体执行(concrete execution).

区别于具体执行,符号执行(symbolic execution)是将程序执行过程中的变量抽象为符号表达式,经过符号化过程,每一条路径都有了相对应的符号表示的约束条件,可以利用约束求解器解出不同程序路径所对应的具体值范围[48],形成输入.

混合执行测试(concolic testing)[14,15]则是对具体执行和符号执行[48,49]两种方法的结合.混合执行测试在具体执行的过程中,同时进行符号执行和符号化路径约束条件的收集.通过具体的执行,程序可以得到本次执行的路径中所有的约束条件,比如分支约束条件.对其中某一个分支约束条件取反,就会得到同一分支的不同路径的输入值,即得到一条新的路径.通过这样的方法,不断地探索程序路径空间,以尽可能地覆盖更多的代码.混合执行测试的优势在于:在符号执行过程中,一旦我们想得到一条路径的输入,就需要先得到这条路径中的所有条件约束在进行约束求解,而混合执行测试是边执行边生成的,同时,具体的数值可以被用于简化符号执行的约束条件.然而,混合执行测试也面临着挑战,比如无法处理规模较大的程序以及建模准确度不够.因此,混合执行测试

工具通常采用一定时间或迭代次数作为测试生成过程的限制.但是这也会带来一些问题,比如路径空间不全、遍历受限不完整等,依旧有亟需优化的地方.混合执行测试的方法被应用于越来越多的领域[50,51],实现了各种混合执行测试工具,例如,Conpy[52]是混合执行测试在 Python 上的实现.

测试生成与其他技术相结合的例子也有很多,比如:协同算法[53]将模型检验(model-checking)与动态符号执行工作 DART[14]相结合,从而尝试遍历程序中的所有状态;文献[54]同样利用了混合执行测试生成测试用例.这些测试生成的目的都是得到更高的覆盖度,尽可能地遍历程序的每一个角落.而我们的工作并不是针对于此,而是希望通过测试生成得到我们需要的某一条或者某几条路径.

与我们的思路有一定相似之处,一些学者也将动态和静态的技术结合起来使用,例如,DyTa[55]工具结合了静态程序验证与动态测试生成工作,以期达到降低误报和提高效率的目的;DSD-Crasher[56]工具依次采用动态-静态-动态方法来发现程序错误;Zhang 和 Saff 等人[57]则针对单元测试提出了一种静动态相结合的方法,用于自动地产生有效和多样的方法调用序列;Babić 等人[58]利用静态分析来指导二进制程序的自动测试生成;Taneja 等人[59]利用基于路径的测试生成来进行更加高效的回归测试;文献[60]提出了一种规则制导(rule-directed)的符号执行技术来高效地检查系统规则;李游等人[61]利用一种 $n$-变长子路径程序频谱(length-$n$ subpath program spectra)来系统地引导符号执行过程覆盖更“少”被覆盖到的程序路径空间.本文的混合执行测试方法是以崔展齐等人[62]提出的一种目标制导的混合执行测试方法为基础的、高效的测试生成和执行方法,能够使得本文的静态警报确认更加高效.

### 4.3 静态分析结果的验证和误报消除

静态分析工具在实际中的应用十分广泛,上文也进行了分析.静态分析工具极容易出现误报和漏报的情况,在这个方向也有一定的相关工作积累.

Ruthruff 等人[63]通过挖掘现有警报中潜在的信息以及相关联的代码,建立基于统计的回归模型来预测之后静态警报的类型.Heckman 等人[64]提出一种减少误报的技术,并用一个基准程序集 FAULTBENCH 来验证该技术.Kim 和 Ernst[65]通过分析软件的多个版本来对警报进行分类.FEEDBACK-RANK[66]利用已知的对静态分析警报的修正,开发出一种基于概率的排序方法来降低误报的数目.Z-Ranking 利用人工验证静态分析结果的统计数据和频次来对警报进行优先级排序[67].Boogerd 和 Moonen 通过采用似然分析来排序静态分析警报[68].Kim 和 Ernst[69]则通过软件变更历史,找到静态分析警报和实际错误修复之间的关系,进而提出一种基于历史的警报优先级(history-based warning prioritization,简称 HWP)处理算法,从而利用修复的历史来对警报进行排序.Dillig 等人提出一种基于诱发推测的算法,该算法被用于判定静态分析中丢失的信息,从而诊断静态分析的警报[70].Clarify[71]工具通过对软件执行情况的总结,利用机器学习的方法,以期改善静态分析的错误报告.ALETHEIA[72]工具同样使用了机器学习的方法,用户只需要人工确认一部分警报,其余的警报利用机器学习的方法进行分类.Chimdyalwar 等人[73]提出了循环抽象的有界模型检验的方法来检测静态分析中的误报,用已知的小边界的抽象循环替换程序中的循环.

在以上这些相关工作中,大多数都是通过静态分析的结果进行二次分析,并没有结合动态执行.文献[70]虽然类似于我们的方法,进行了动态测试,但其目的主要在于采集更多的信息,帮助开发人员理解和修复缺陷.因此,当前并没有与我们完全相同的工作.我们在前期工作[74]中提出了采用混合执行测试方法对静态内存泄漏警报进行分类的思想和方法,这是本文之前的工作,也是本文的基础.本文增加了与动态测试技术的对比实验,在该实验中,一方面对比了内存泄漏的动态测试工具,另一方面,利用高覆盖度的测试用例集合作为测试输入,替代混合执行产生的测试用例,执行插装后的程序对警报进行分类,进一步证明了本文思想和方法的有效性和效率.利用动静态结合的方法,能够对静态内存泄漏警报进行准确的分类,同时也大大降低了时空开销.

## 5 总结与展望

本文提出了一种基于混合执行测试的 C/C++程序静态内存泄漏警报自动确认方法,对静态分析工具报告的内存泄漏警报进行动态的自动化确认和分类.首先,在被测程序的控制流图上进行警报的可达性分析,计算程

序中分支语句到路径片段中节点的可达性,得到路径制导信息;其次,以控制流图上静态内存泄漏警报的可能路径为覆盖目标,基于混合执行测试方法产生测试用例并执行;最后,追踪和监控运行时内存对象的状态,在测试执行结束时判断内存泄漏是否存在,从而完成对静态内存泄漏警报的确认和分类.

在本文方法的实现过程和实验中,我们也遇到了一些问题和挑战,这也是我们未来的研究方向:(1) 本文对内存状态的判断是通过对其的追踪和更新完成的,可以在状态中加入更多内容以得到更多的信息,例如在MUST- LEAK 中,利用内存泄漏的大小对警报进行一个更细的排序;(2) 在内存追踪时加入内存对象的指针或者引用计数,从而更准确地验证警报;(3) 将当前的工作扩展到其他类型的内存缺陷中;(4) 研究更高效的测试策略,提升本文方法的规模化能力.

**致谢** 在此,向对本文工作给予建议的老师、参与本文实验的所有同学、给我们提出建议的评审专家表示感谢.

**References**:

[1] Clause J, Orso A. LEAKPOINT: Pinpointing the causes of memory leaks. In: Proc. of the 32nd ACM/IEEE Int'l Conf. on Software Engineering. New York: ACM Press, 2010. 515−524. [doi: 10.1145/1806799.1806874]

[2] Hastings R, Joyce B. Purify: Fast detection of memory leaks and access errors. In: Proc. of the USENIX Conf. Berkeley: Usenix Association, 1992. 125−138.

[3] Omega: An instant leak detector tool for valgrind. 2016. http://www.brainmurders.eclipse.co.uk/omega.html

[4] Novark G, Berger ED, Zorn BG. Efficiently and precisely locating memory leaks and bloat. ACM SIGPLAN Notices, 2009,44(6): 397−407. [doi: 10.1145/1543135.1542521]

[5] Qin F, Lu S, Zhou Y. SafeMem: Exploiting ECC-memory for detecting memory leaks and memory corruption during production runs. In: Proc. of the 11th IEEE Int'l Symp. on High-Performance Computer Architecture. Piscataway: IEEE, 2005. 291−302. [doi: 10.1109/HPCA.2005.29]

[6] Cherem S, Princehouse L, Rugina R. Practical memory leak detection using guarded value-flow analysis. ACM SIGPLAN Notices, 2007,42(6):480−491. [doi: 10.1145/1273442.1250789]

[7] Heine DL, Lam MS. A practical flow-sensitive and context-sensitive C and C++ memory leak detector. ACM SIGPLAN Notices, 2003,38(5):168−181. [doi: 10.1145/780822.781150]

[8] Heine DL, Lam MS. Static detection of leaks in polymorphic containers. In: Proc. of the 28th Int'l Conf. on Software Engineering. New York: ACM Press, 2006. 252−261. [doi: 10.1145/1134285.1134321]

[9] Orlovich M, Rugina R. Memory leak analysis by contradiction. In: Proc. of the Static Analysis. Berlin, Heidelberg: Springer-Verlag, 2006. 405−424. [doi: 10.1007/11823230_26]

[10] Xie Y, Aiken A. Context-And path-sensitive memory leak detection. ACM SIGSOFT Software Engineering Notes, 2005,30(5): 115−125. [doi: 10.1145/1095430.1081728]

[11] Klocwork. The Klocwork static analysis tool. 2001~2016. http://www.klocwork.com/

[12] Coverity. The coverity static analysis tools. 2016. http://www.coverity.com/

[13] HP Fortify. 2016. http://www8.hp.com/us/en/software-solutions/application-security/index.html

[14] Godefroid P, Klarlund N, Sen K. DART: Directed automated random testing. ACM SIGPLAN Notices, 2005,40(6):213−223. [doi: 10.1145/1064978.1065036]

[15] Sen K, Marinov D, Agha G. CUTE: A concolic unit testing engine for C. ACM SIGSOFT Software Engineering Notes, 2005,30(5): 263−272. [doi: 10.1145/1095430.1081750]

[16] Kosmatov N. All-Paths test generation for programs with internal aliases. In: Proc. of the 19th IEEE Int'l Symp. on Software Reliability Engineering. Piscataway: IEEE, 2008. 147−156. [doi: 10.1109/ISSRE.2008.25]

[17] Xu G, Mitchell N, Arnold M, Rountev A, Schonberg E, Sevitsky G. Finding low-utility data structures. ACM SIGPLAN Notices, 2010,45(6):174−186. [doi: 10.1145/1809028.1806617]

[18] Xu G, Arnold M, Mitchell N, Rountev A, Sevitsky G. Go with the flow: Profiling copies to find runtime bloat. ACM SIGPLAN Notices, 2009,44(6):419−430. [doi: 10.1145/1543135.1542523]

[19] Heckman S, Williams L. On establishing a benchmark for evaluating static analysis alert prioritization and classification techniques. In: Proc. of the 2nd ACM-IEEE Int'l Symp. on Empirical Software Engineering and Measurement. New York: ACM Press, 2008. 41−50. [doi: 10.1145/1414004.1414013]

[20] Arnold M, Vechev M, Yahav E. QVM: An efficient runtime for detecting defects in deployed systems. ACM SIGPLAN Notices, 2008,43(10):143−162. [doi: 10.1145/1449955.1449776]

[21] CREST: An automatic test generation tool for C. 2016. https://github.com/jburnim/crest

[22] Yices: An SMT solver. 2016. http://yices.csl.sri.com/

[23] Siemens. 2016. http://sir.unl.edu/

[24] GNU core utilities. 2001~2016. http://www.gnu.org/software/coreutils/coreutils.html

[25] Valgrind. 2016. http://valgrind.org/

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

[27] Dor N, Rodeh M, Sagiv M. Checking cleanness in linked lists. In: Proc. of the Static Analysis. Berlin, Heidelberg: Springer-Verlag, 2000. 115−134. [doi: 10.1007/978-3-540-45099-3_7]

[28] Hackett B, Rugina R. Region-Based shape analysis with tracked locations. ACM SIGPLAN Notices, 2005,40(1):310−323. [doi: 10.1145/1047659.1040331]

[29] Sui Y, Ye D, Xue J. Static memory leak detection using full-sparse value-flow analysis. In: Proc. of the 2012 Int'l Symp. on Software Testing and Analysis. New York: ACM Press, 2012. 254−264. [doi: 10.1145/2338965.2336784]

[30] Sui Y, Ye D, Xue J. Detecting memory leaks statically with full-sparse value-flow analysis. IEEE Trans. on Software Engineering, 2014,40(2):107−122. [doi: 10.1109/TSE.2014.2302311]

[31] Xu Z, Zhang J, Xu Z. Melton: A practical and precise memory leak detection tool for C programs. Frontiers of Computer Science, 2015,9(1):34−54. [doi: 10.1007/s11704-014-3460-8]

[32] Nethercote N, Seward J. Valgrind: A framework for heavyweight dynamic binary instrumentation. ACM SIGPLAN Notices, 2007, 42(6):89−100. [doi: 10.1145/1273442.1250746]

[33] Jung C, Lee S, Raman E, Pande S. Automated memory leak detection for production use. In: Proc. of the 36th Int'l Conf. on Software Engineering. New York: ACM Press, 2014. 825−836. [doi: 10.1145/2568225.2568311]

[34] Lee S, Jung C, Pande S. Detecting memory leaks through introspective dynamic behavior modelling using machine learning. In: Proc. of the 36th Int'l Conf. on Software Engineering. New York: ACM Press, 2014. 814−824. [doi: 10.1145/2568225.2568307]

[35] Bond MD, McKinley KS. Bell: Bit-Encoding online memory leak detection. ACM SIGPLAN Notices, 2006,41(11):61−72. [doi: 10.1145/1168918.1168866]

[36] Bond MD, McKinley KS. Tolerating memory leaks. ACM SIGPLAN Notices, 2008,43(10):109−126. [doi: 10.1145/1449955.1449774]

[37] Yan D, Xu G, Yang S, Rountev A. Leakchecker: Practical static memory leak detection for managed languages. In: Proc. of the Annual IEEE/ACM Int'l Symp. on Code Generation and Optimization. New York: ACM Press, 2014. 87. [doi: 10.1145/2544137.2544151]

[38] Bond MD, McKinley KS. Leak pruning. ACM SIGARCH Computer Architecture News, 2009,37(1):277−288. [doi: 10.1145/1508284.1508277]

[39] Pauw WD, Sevitsky G. Visualizing reference patterns for solving memory leaks in Java. In: Proc. of the European Conf. on Object-Oriented Programming. Berlin, Heidelberg: Springer-Verlag, 1999. 116−134. [doi: 10.1007/3-540-48743-3_6]

[40] Hauswirth M, Chilimbi TM. Low-Overhead memory leak detection using adaptive statistical profiling. ACM SIGPLAN Notices, 2004,39(11):156−164. [doi: 10.1145/1037187.1024412]

[41] Jump M, McKinley KS. Cork: Dynamic memory leak detection for garbage-collected languages. ACM SIGPLAN Notices, 2007, 42(1):31−38. [doi: 10.1145/1190215.1190224]

[42] Mitchell N, Sevitsky G. LeakBot: An automated and lightweight tool for diagnosing memory leaks in large Java applications. In: Proc. of the Object-Oriented Programming (ECOOP 2003). Berlin, Heidelberg: Springer-Verlag, 2003. 351−377. [doi: 10.1007/978-3-540-45070-2_16]

[43] Rayside D, Mendel L. Object ownership profiling: A technique for finding and fixing memory leaks. In: Proc. of the 22nd IEEE/ACM Int'l Conf. on Automated Software Engineering. New York: ACM Press, 2007. 194−203. [doi: 10.1145/1321631.1321661]

[44] Tang Y, Gao Q, Qin F. LeakSurvivor: Towards safely tolerating memory leaks for garbage-collected languages. In: Proc. of the USENIX 2008 Annual Technical Conf. on Annual Technical Conf. Berkeley: USENIX Association, 2008. 307−320.

[45] Xu G, Bond MD, Qin F, Rountev A. LeakChaser: Helping programmers narrow down causes of memory leaks. ACM SIGPLAN Notices, 2011,46(6):270−282. [doi: 10.1145/1993316.1993530]

[46] Xu G, Rountev A. Precise memory leak detection for Java software using container profiling. In: Proc. of the 30th Int'l Conf. on Software Engineering. Piscataway: IEEE, 2008. 151−160. [doi: 10.1145/1368088.1368110]

[47] Gao Q, Xiong Y, Mi Y, Zhang L, Yang WK, Zhou ZP, Xie B, Mei H. Safe memory-leak fixing for C programs. In: Proc. of the 37th Int'l Conf. on Software Engineering, Vol.1. Piscataway: IEEE Press, 2015. 459−470. [doi: 10.1109/ICSE.2015.64]

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

[49] Clarke L. A system to generate test data and symbolically execute programs. IEEE Trans. on Software Engineering, 1976,SE-2(3):215−222. [doi: 10.1109/TSE.1976.233817]

[50] Giantsios A, Papaspyrou N, Sagonas K. Concolic testing for functional languages. In: Proc. of the 17th Int'l Symp. on Principles and Practice of Declarative Programming. New York: ACM Press, 2015. 137−148. [doi: 10.1145/2790449.2790519]

[51] Mesnard F, Payet É, Vidal G. Concolic testing in logic programming. Theory and Practice of Logic Programming, 2015,15(4-5):711−725. [doi: 10.1017/S1471068415000332]

[52] Chen T, Zhang X, Chen R, Yang B, Bai Y. Conpy: Concolic execution engine for python applications. In: Proc. of the Algorithms and Architectures for Parallel Processing. Springer Int'l Publishing, 2014. 150−163. [doi: 10.1007/978-3-319-11194-0_12]

[53] Gulavani BS, Henzinger TA, Kannan Y, Nori AV, Rajamani SK. SYNERGY: A new algorithm for property checking. In: Proc. of the 14th ACM SIGSOFT Int'l Symp. on Foundations of Software Engineering. New York: ACM Press, 2006. 117−127. [doi: 10.1145/1181775.1181790]

[54] Vidal G. Concolic execution and test case generation in prolog. In: Proc. of the Logic-Based Program Synthesis and Transformation. Springer Int'l Publishing, 2014. 167−181. [doi: 10.1007/978-3-319-17822-6_10]

[55] Ge X, Taneja K, Xie T, Tillmann N. DyTa: Dynamic symbolic execution guided with static verification results. In: Proc. of the 33rd Int'l Conf. on Software Engineering. New York: ACM Press, 2011. 992−994. [doi: 10.1145/1985793.1985971]

[56] 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):8. [doi: 10.1145/1348250.1348254]

[57] Zhang S, Saff D, Bu Y, Ernst MD. Combined static and dynamic automated test generation. In: Proc. of the 2011 Int'l Symp. on Software Testing and Analysis. New York: ACM Press, 2011. 353−363. [doi: 10.1145/2001420.2001463]

[58] Babić D, Martignoni L, McCamant S, *et al*. Statically-Directed dynamic automated test generation. In: Proc. of the 2011 Int'l Symp. on Software Testing and Analysis. New York: ACM Press, 2011. 12−22. [doi: 10.1145/2001420.2001423]

[59] Taneja K, Xie T, Tillmann N, Halleux JD. eXpress: Guided path exploration for efficient regression test generation. In: Proc. of the 2011 Int'l Symp. on Software Testing and Analysis. New York: ACM Press, 2011. 1−11. [doi: 10.1145/2001420.2001422]

[60] Cui H, Hu G, Wu J, Yang J. Verifying systems rules using rule-directed symbolic execution. ACM SIGPLAN Notices, 2013,48(4):329−342. [doi: 10.1145/2499368.2451152]

[61] Li Y, Su Z, Wang L, Li X. Steering symbolic execution to less traveled paths. ACM SIGPLAN Notices, 2013,48(10):19−32. [doi: 10.1145/2544173.2509553]

[62] Cui Z, Wang L, Li X. Target-Directed concolic testing. Jisuanji Xuebao/Chinese Journal of Computers, 2011,34(6):953−964 (in Chinese with English abstract).

[63] Ruthruff JR, Penix J, Morgenthaler JD, Elbaum S, Rothermel G. Predicting accurate and actionable static analysis warnings: An experimental approach. In: Proc. of the 30th Int'l Conf. on Software Engineering. New York: ACM Press, 2008. 341−350. [doi: 10.1145/1368088.1368135]

[64] Heckman S, Williams L. On establishing a benchmark for evaluating static analysis alert prioritization and classification techniques. In: Proc. of the 2nd ACM-IEEE Int'l Symp. on Empirical Software Engineering and Measurement. New York: ACM Press, 2008. 41−50. [doi: 10.1145/1414004.1414013]

[65] Kim S, Ernst MD. Which warnings should I fix first. In: Proc. of the 6th Joint Meeting of the European Software Engineering Conf. and the ACM SIGSOFT Symp. on the Foundations of Software Engineering. New York: ACM Press, 2007. 45−54. [doi: 10.1145/1287624.1287633]

[66] Kremenek T, Ashcraft K, Yang J, Engler D. Correlation exploitation in error ranking. ACM SIGSOFT Software Engineering Notes, 2004,29(6):83−93. [doi: 10.1145/1041685.1029909]

[67] Kremenek T, Engler D. Z-Ranking: Using statistical analysis to counter the impact of static analysis approximations. In: Proc. of the Static Analysis. Berlin, Heidelberg: Springer-Verlag, 2003. 295−315. [doi: 10.1007/3-540-44898-5_16]

[68] Boogerd C, Moonen L. Prioritizing software inspection results using static profiling. In: Proc. of the 6th IEEE Int'l Workshop on Source Code Analysis and Manipulation. Piscataway: IEEE, 2006. 149−160. [doi: 10.1109/SCAM.2006.22]

[69] Kim S, Ernst MD. Prioritizing warning categories by analyzing software history. In: Proc. of the 4th Int'l Workshop on Mining Software Repositories. IEEE Computer Society, 2007. 27. [doi: 10.1109/MSR.2007.26]

[70] Dillig I, Dillig T, Aiken A. Automated error diagnosis using abductive inference. ACM SIGPLAN Notices, 2012,47(6):181−192. [doi: 10.1145/2345156.2254087]

[71] Ha J, Rossbach CJ, Davis JV, Roy I, Ramadan HE, Porter DE, Chen DL, Witchel E. Improved error reporting for software that uses black-box components. ACM SIGPLAN Notices, 2007,42(6):101−111. [doi: 10.1145/1273442.1250747]

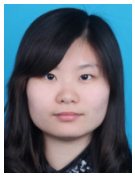
[72] Tripp O, Guarnieri S, Pistoia M, Aravkin A. ALETHEIA: Improving the usability of static security analysis. In: Proc. of the 2014 ACM SIGSAC Conf. on Computer and Communications Security. New York: ACM Press, 2014. 762−774. [doi: 10.1145/2660267.2660339]

[73] Chimdyalwar B, Darke P, Chavda A, Vaghani S, Chauhan A. Eliminating static analysis false positives using loop abstraction and bounded model checking. In: Proc. of the Formal Methods (FM 2015). Springer Int'l Publishing, 2015. 573−576. [doi: 10.1007/978-3-319-19249- 9_35]

[74] Li M, Chen Y, Wang L, Xu G. Dynamically validating static memory leak warnings. In: Proc. of the 2013 Int'l Symp. on Software Testing and Analysis. New York: ACM Press, 2013. 112−122. [doi: 10.1145/2483760.2483778]

**附中文参考文献**:

[62] 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.



**李筱**(1992－),女,山东烟台人,硕士生,主要研究领域为软件测试,测试自动化,移动应用测试.

**周严**(1991－),男,硕士生,主要研究领域为软件自动化测试,代码静态分析.

**李孟宸**(1988－),男,硕士,主要研究领域为软件自动化测试,程序分析.

**陈园军**(1988－),男,硕士,主要研究领域为软件测试,嵌入式系统,操作系统内存模型.

**XU Guo-Qing** (1981－),男,博士,助理教授,主要研究领域为程序设计语言,编译器优化,程序分析,系统软件,分布式系统,大数据分析.

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

**李宣东**(1963－),男,博士,教授,博士生导师,CCF 会士,主要研究领域为软件建模与分析,软件测试与验证.