软件学报 ISSN 1000-9825, CODEN RUXUEW
*Journal of Software*,2019,30(5):1330−1341 [doi: 10.13328/j.cnki.jos.005715]

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

# C 程序内存泄漏智能化检测方法*

朱亚伟, 左志强, 王林章, 李宣东

(计算机软件新技术国家重点实验室(南京大学),江苏 南京 210023)

通讯作者: 王林章, E-mail: lzwang@nju.edu.cn

**摘 要**: 内存泄漏在采用显式内存管理机制的C语言中是一种常见的代码缺陷,内存泄漏的检测方法目前主要是静态分析与动态检测.动态检测开销大,且高度依赖测试用例;静态分析目前被学术界和工业界广泛应用,但是存在大量误报,需要人工对检测结果进行确认.内存泄漏静态分析的误报通常是由于对指针、分支语句和全局变量分析的不准确性导致的.提出了一种内存泄漏的智能化检测方法,通过使用机器学习算法学习程序特征与内存泄漏之间的相关性,构建机器学习分类器,并应用机器学习分类器进一步提高内存泄漏静态分析的准确性.首先构建机器学习分类器,然后通过静态分析方法构建从内存分配点开始的 Sparse Value Flow Graph(SVFG),并从中提取内存泄漏相关特征,再使用规则和机器学习分类器进行内存泄漏的检测.实验结果显示,该方法在分析指针、分支语句和全局变量时是有效的,能够提高内存泄漏检测的准确性,降低内存泄漏检测结果的误报.最后,对未来研究的可行性以及面临的挑战进行了展望.

**关键词**: 内存泄漏;内存泄漏检测;静态分析;机器学习;特征提取

**中图法分类号**: TP311

中文引用格式: 朱亚伟,左志强,王林章,李宣东.C 程序内存泄漏智能化检测方法.软件学报,2019,30(5):1330−1341. http://www.jos.org.cn/1000-9825/5715.htm

英文引用格式: Zhu YW, Zuo ZQ, Wang LZ, Li XD. Memory leak intelligent detection method for C programs. Ruan Jian Xue Bao/Journal of Software, 2019,30(5):1330−1341 (in Chinese). http://www.jos.org.cn/1000-9825/5715.htm

## Memory Leak Intelligent Detection Method for C Programs

ZHU Ya-Wei, ZUO Zhi-Qiang, WANG Lin-Zhang, LI Xuan-Dong

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

**Abstract**: Memory leak is a common code bug for C programs which uses explicit memory management mechanisms. At present, the main detection methods of memory leaks are static analysis and dynamic detection. Dynamic detection has huge overhead and it is highly dependent on test cases. Static analysis is widely used by academic and industry, but there are a large number of false positives, which need to be manually confirmed. Inaccuracy in the analysis of pointers, branch statements, and global variables leads to false positives in static analysis of memory leaks. In this study, an intelligent detection method is proposed for memory leak. By using machine learning algorithms to learn the correlation between program's features and memory leaks, a machine learning classifier is built and applied to improve the accuracy of static analysis of memory leaks. Firstly, a machine learning classifier is trained. Then, the sparse value flow graph (SVFG) starting from allocation should be constructed by using the static analysis, the features related to memory leaks can be extracted from the SVFG. Lastly, the memory leaks are detected by using rules and machine learning classifier. Experimental results show

* 基金项目: 国家重点研发计划(2016YFB1000802); 国家自然科学基金(61802168, 61632015); 中央高校基本科研业务费专项资金(020214380047)

Foundation item: National Key Research and Development Program of China (2016YFB1000802); National Natural Science Foundation of China (61802168, 61632015); Fundamental Research Funds for the Central Universities (020214380047)

本文由智能化软件新技术专刊特约编辑申富饶教授和李戈副教授推荐.

收稿时间: 2018-08-29; 修改时间: 2018-10-31; 采用时间: 2018-12-13

that the proposed method is effective in analyzing pointers, branch statements, and global variables, and can reduce the false positives of memory leak detection. At the end of this paper, the feasibility of future research and the upcoming challenges are presented.

**Key words**: memory leak; memory leak detection; static analysis; machine learning; extracting feature

内存泄漏会严重降低软件的性能,甚至造成软件在运行时崩溃.在 C 语言中,内存的分配与释放都是人为控制的,随着计算机软件的规模和复杂度的不断增加,人为的疏忽极易导致内存泄漏.由于 C 语言在计算机领域的广泛应用,C 程序中的内存泄漏问题不可忽视.目前,内存泄漏的检测方法主要是静态分析与动态检测.

- 动态检测方法[1−4]需要执行程序,在程序运行的过程中对内存的分配、使用以及释放进行动态跟踪.由于动态检测能够针对当次的运行结果得到一个明确的结论,因此动态检测相对静态分析,其结果更加准确.但是动态检测的准确性受限于测试用例,无法分析程序执行中不可达位置的错误.目前,成熟的内存泄漏动态检测工具有 Purify[5]、Valgrind[6]等,都存在内存开销较高和可扩展性较差的问题.
- 静态分析是在不实际运行程序的情况下对程序代码及其结构进行分析.针对 C 语言的内存泄漏静态分析方法主要通过分析内存的分配点以及从内存分配点开始的不同路径,在相应的路径中查找与内存分配点对应的内存释放点,验证是否所有路径都存在正确的内存释放.目前,有许多内存泄漏的静态分析工作[7−12],也有一些成熟的静态分析工具,如 Fortify[13]、Coverity[14]、Klocwork[15],这些工具在工业界的软件开发中应用广泛.静态分析方法又可根据流敏感、上下文敏感以及域敏感等进行分类,实现这些静态分析方法可以在一定程度上提高内存泄漏检测的准确率,但会极大降低检测效率.

内存泄漏的静态分析目前主要缺点是:当内存泄漏中存在一些特殊案例时,会降低静态分析的准确性,导致内存泄漏的检测出现误报或者漏报.其主要原因在于:

(1) 对分支条件缺少分析,无法识别不可达路径;对全局变量和指针的重定向未做细致分析;忽略指针偏移问题.可以通过流敏感、上下文敏感等静态分析方法在一定程度上解决以上问题,但会造成检测效率大幅降低.

(2) 动态数组的分配问题、链表的分配释放不匹配问题、循环或递归中指针分析的准确性问题(限制循环或递归的展开次数)会导致指针分析不准确.流敏感、上下文敏感等目前已有的静态分析方法无法解决上述问题.

针对静态分析的上述不足,本文利用机器学习算法以提高内存泄漏静态分析的准确性,获得更加准确的内存泄漏检测结果.本文利用已有的内存泄漏案例,即含有标签为虚假内存泄漏和真实内存泄漏的两类数据集,通过使用机器学习算法进行训练,构建内存泄漏分类器以分析程序特征与内存泄漏的相关性.

本文的主要贡献在于:

(1) 本文提出一种 C 程序内存泄漏智能化检测方法,该方法基于静态分析提取内存泄漏相关特征,能够提高内存泄漏检测的准确性,减少误报和漏报.

(2) 我们实现了内存泄漏的智能化检测工具 I_Mem.该工具使用多种机器学习算法构建内存泄漏检测模型,并基于静态分析提取内存泄漏相关特征,能够提高内存泄漏检测的准确性.

(3) 我们在 LLVM-4.0.0 上实现了 I_Mem,并利用 4 个开源的 C 程序(2KLOC)进行实验评估.I_Mem 总共发现了 114 个内存泄漏,漏报为 4 个,误报为 13 个,准确率为 85.6%.

本文第 1 节主要介绍背景知识.第 2 节介绍 C 程序内存泄漏智能化检测方法,包括方法的基本框架、内存泄漏模型的构建、特征提取与缺陷检测.第 3 节对本文实现的工具 I_Mem 进行实验和评估.第 4 节主要介绍相关工作,包括内存泄露的静态分析方法、内存泄露的动态检测技术和基于机器学习的缺陷检测.第 5 节进行总结和展望.

## 1 背 景

针对内存泄漏的程序静态分析需要在获取所有内存分配点后,关注内存的定义、使用及释放.此外,C 语言

中由于函数可以返回指针,内存泄漏检测需要对调用点作分析.因此,Saber[8,9]关注 C 程序中的 6 种语句,见表 1.在表 1 中,$p$ 和 $q$ 是变量,$v$ 表示变量或者堆对象,$F$ 是函数.

**Table 1** Six types of statements

**表 1** 6 种语句类型

| 名称 | 句法 |
|---|---|
| 取址 | $p=\&v$ |
| 赋值 | $p=q$ |
| Load | $p=^{*}q$ |
| Store | $^{*}p=q$ |
| 函数调用 | $p=F(\ldots,q,\ldots)$ |
| 返回 | return $p$ |

Saber 中使用的静态分析方法是 Full-Sparse Value-Flow Analysis,通过构建 SVFG 来检测内存泄漏.其中,VFG(value flow graph)[7,16]表示的是句法上的语义等价,它关注变量的定义、使用,能够表示程序中变量的价值流向.VFG 不同于 CFG(control flow graph)和 DFG(data flow graph),CFG 表示的是控制程序逻辑执行的先后顺序,一个程序的 CFG 被用来确定对变量的一次赋值可能传播到程序中的哪些位置;DFG 是在 CFG 基础上实现的,它描述的是程序运行过程中数据的流转方式及其行为状态.在 VFG 中,每一个节点表示变量的定义,每条边表示变量的 def-use 关系.Saber 针对 VFG 进行改进构建了 SVFG.SVFG 的构建主要分为 3 个步骤.

(1) 预分析:根据内存分配相关的 API(如 malloc)确定内存位置.使用域敏感、调用点敏感、流和上文不敏感的安德森指针分析获取 C 程序的指针信息.

(2) 全稀疏 SSA(static single assigment):在 SSA 形式中,每个被使用的变量都有唯一的定义,这可以确保精确地 def-use 关系链.针对所有内存位置,构造每个函数的 SSA 形式.关注内存位置的间接访问,如 load,store 以及函数调用操作.用程序内流敏感的指针分析对预分析的指针信息进一步稀疏提高指针分析的准确性.

(3) SVFG:基于全稀疏 SSA,获取程序内所有内存位置的 def-use 关系链和 value-flow,并构建 SVFG.每个 def-use 边会有一个"警卫"来获取分支条件.

## 2 C 程序内存泄漏智能化检测方法

本节我们主要介绍 C 程序内存泄漏智能化检测方法的基本框架.本方法是在内存泄漏静态分析的基础上进行改进,利用机器学习技术提高了内存泄漏静态分析的准确性.图 1 是 C 程序内存泄漏智能化检测方法的主要框架.

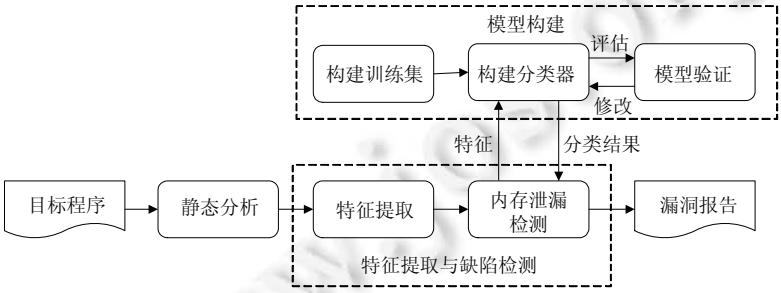


Fig.1 Framework of our work

图 1 本文的工作框架

该方法主要可分为两个步骤——模型构建阶段以及特征提取与缺陷检测阶段.

(1) 模型构建阶段

- 首先,根据已有的内存泄漏构建训练集,构建训练机分为两步:第 1 步,构建包含真正内存泄漏与

虚假内存泄漏的数据集;第 2 步,从两个数据集中分别提取内存泄漏特征;

- 然后,将训练集输入机器学习的分类器进行训练,并应用交叉验证对分类器进行评估;
- 随后,修改分类器类型及参数,选取分类效果最好的作为内存泄漏检测模型.

(2) 特征提取与缺陷检测阶段

- 首先,利用静态分析方法分析源程序,获取所有的内存位置(内存分配点 *o*),并构建从 *o* 开始的 SVFG,提取 SVFG 中每条路径对应的内存泄漏特征并构成数据集;
- 根据内存泄漏特征,我们可将数据集分成两部分:一部分可根据规则直接判断是否为内存泄漏,另一部分需要输入到内存泄漏检测模型中进行判断;
- 最后,将内存泄露检测结果与静态分析阶段的分配点的信息相结合,得到漏洞报告.

### 2.1 模型构建阶段

针对内存泄漏的静态分析中存在的不足,我们提取了 16 个内存泄漏特征.本文根据从内存分配点开始的 SVFG 提取内存泄漏特征,每一个内存分配点对应一个内存泄漏特征.提取的内存泄漏特征信息(内存分配点 *o*,内存释放点指针 *p*)包括 3 类:类型信息、分支信息、释放信息.类型信息包括数组(判断 *o* 是否为数组元素)、结构体(判断 *o* 是否为结构体元素)、链表(判断 *o* 是否为链表元素),分支信息包括循环分配(*o* 是否在循环内部分配)、循环释放(*p* 是否在循环内部释放)、循环匹配(循环分配与释放次数是否一致)、同一循环(分配与释放是否在同一循环内)、链表匹配(链表的分配与释放次数是否一致)、分支条件(若当前分支中不存在对 *o* 的释放,则判断分支条件是否为真),释放信息包括函数间距(从 *o* 出发的 SVFG 最多经过的函数数量)、释放之前为空(*p* 释放之前是否为空指针)、*p* 指向对象数目(释放 *p* 时,*p* 指向的对象数目)、别名数目(释放 *p* 时,指向 *o* 的指针数目)、指针偏移量(释放 *p* 时,*p* 与 *o* 的偏移量)、全局指针(判断在 SVFG 中有全局变量指向 *o*)、释放(指针 *p* 是否释放).特征见表 2.

**Table 2** Features of memory leak

**表 2** 内存泄漏特征

| 类别 | 序号 | 特征 | 类型 | 描述 |
|---|---|---|---|---|
| 类型信息 | 1 | 数组 | 布尔值 | *o* 是否为数组元素 |
| | 2 | 结构体 | 布尔值 | *o* 是否为结构体元素 |
| | 3 | 链表 | 布尔值 | *o* 是否为链表元素 |
| 分支信息 | 4 | 循环分配 | 布尔值 | o 是否在循环内部分配 |
| | 5 | 循环释放 | 布尔值 | *p* 是否在循环内部释放 |
| | 6 | 循环匹配 | 布尔值 | 循环分配次数与循环释放次数是否一致 |
| | 7 | 同一循环 | 布尔值 | *o* 的分配与 *p* 的释放是否在同一循环中 |
| | 8 | 链表匹配 | 布尔值 | 链表的分配与释放次数是否一致 |
| | 9 | 分支条件 | 布尔值 | 当前分支中不存在对 *o* 的释放,判断分支条件是否为真 |
| 释放信息 | 10 | 函数间距 | 整型 | 当前路径中,从分配点*o*到释放 *p* 所经过的函数数量 |
| | 11 | 释放之前为空 | 布尔值 | *p* 释放之前是否为空指针 |
| | 12 | *p* 指向对象数目 | 整型 | 释放 *p* 时,*p* 指向的对象数目 |
| | 13 | 别名数目 | 整型 | 释放 *p* 时,指向 *o* 的指针数目 |
| | 14 | 指针偏移量 | 整型 | 释放 *p* 时,*p* 与 *o* 的偏移量 |
| | 15 | 全局指针 | 布尔值 | SVFG 中,是否有全局变量指向 *o* |
| | 16 | 释放 | 布尔值 | 指针 *p* 是否释放 |

本文针对内存泄漏共提取 16 个特征,其中,部分特征与内存泄漏静态分析的不足存在如下的对应关系.

(1) 针对内存泄漏有关的数组、链表、循环或递归问题,我们使用特征 1、特征 3~特征 8 进行判断.

(2) 特征 9 用于对分支条件进行判断,特征 11、特征 12、特征 15 用于判定全局变量以及指针重定向问题,指针偏移量问题使用特征 14 进行判断.

(3) 其余特征用于对内存泄漏的一些复杂情况进行辅助判断.

确定内存泄漏相关特征后,我们需针对各种内存泄漏构建训练集.我们使用的是开源的 C 程序源码(包括一

些大型程序和小程序),在原程序中,我们会插入各种内存泄漏以构建尽量丰富的训练集,然后从中区分真正的与虚假的内存泄漏.在人工对训练数据进行标记时,我们发现如下规律.

(1) 特征 15、特征 16 都为假(即在 SVFG 中,不存在全局变量指向内存分配点,也不存在内存释放点),通常是真正内存泄漏.

(2) 特征 11 为真(即 $p$ 释放之前为空指针)、或者特征 12 大于 1(即 $p$ 指向的对象数目超过 1)、或者特征 14 不为 0(存在指针偏移)、或者特征 16 为假(即未进行内存释放),通常是真正内存泄漏.

在对训练数据进行人工标记后,我们获取每个训练样本的内存泄漏特征,并将这些作为训练集输入分类器进行模型构建.现阶段主流的机器学习算法有 SVM、决策树、朴素贝叶斯分类、隐马尔可夫、随机森林、循环神经网络、长短期记忆(LSTM)与卷积神经网络等.本文采用 3 种机器学习算法:SVM[17]、随机森林(RF)[18]和决策树[19].下面介绍 3 种算法的优缺点.

(1) 决策树是指 C4.5 算法,优点是产生的分类规则易于理解,训练时间复杂度较低,准确率较高;缺点是针对连续属性值的特征时计算效率低,且容易过拟合.

(2) 随机森林的优点是训练速度快,易并行化,能够处理高维度的数据,适合做多分类问题;缺点是在处理噪音较大的数据集上容易过拟合,过于随机导致无法控制模型内部运行.

(3) SVM 可以通过计算数学函数将训练数据分开,这些函数被称为核函数.常用的核函数有 4 种:线性、多项式、径向基核函数(RBF)和 Sigmoid 函数.根据文献[20,21],使用 RBF 核函数将每个特征向量映射到高维空间,这样可以防止过拟合.缺点是对大规模训练样本难以实施,解决多分类问题存在困难.

在构建模型时,需要确定分类器类型及参数,我们使用分类的准确率作为评估标准,然后进行迭代确定最优的分类器类型及参数,步骤如下.

(1) 首先选定第 1 个分类器类型以及参数并进行训练,使用五折交叉验证得到的准确率作为基线;

(2) 多次修改分类器参数(根据分类器类型可自行调整),记录五折交叉验证的准确率超过基线的分类器类型、参数以及准确率;

(3) 修改分类器类型,并重复第(2)步;

(4) 选取准确率最高的分类器(类型及参数)作为本文内存泄漏检测模型.

### 2.2 特征提取与缺陷检测阶段

特征提取与缺陷检测主要分为以下 3 个步骤:程序静态分析、特征提取、内存泄漏检测.程序静态分析就是第 1 节所介绍的 Full-Sparse Value-Flow Analysis.

#### 2.2.1 特征提取

本文通过构建 SVFG 并获取特征,算法 1 阐述了如何通过 SVFG 来获取内存泄漏特征,该算法的输入是内存位置(内存分配点)集合 *src* 以及 *SVFG*.基本思想是:分析内存分配点的信息获取内存泄漏的部分类型和指针信息,分析内存分配点到释放点的路径信息获取内存泄漏的其余信息.

- 首先,在第 1 行~第 5 行,我们初始化存放特征信息的向量 *V*,然后遍历内存分配点的集合 *src*,并从内存分配点开始向前遍历 *SVFG*,获取当前内存分配点的部分类型信息和分支信息(如内存分配点是否为数组或者结构体、内存分配点是否在循环中等)存入 *V* 中,记录经过的节点集合 *FNode* 以及内存释放点集合 *dst*,并初始化 *BNode* 集合(用于存放内存释放点向后遍历 *SVFG* 经过的节点).
- 然后,在第 6 行至第 11 行,遍历内存释放点集合 *dst*,并从内存释放点开始向后遍历 *SVFG*,若当前节点出现在 *FNode* 中,则将该节点放入 *BNode* 中.
- 最后遍历 *FNode* 与 *BNode*,获取当前内存分配点对应的特征信息存入 *V* 中.

**算法 1**. SVFG 遍历获取内存泄漏特征.

输入:内存分配点的集合 *src*;

*SVFG*.

**Begin**

1: 初始化存放特征信息的向量 *V*
2: **for** (*i*=0; *i*<*src.size*(); *i*++)
3: 初始化当前内存泄漏点对应的向量 *V*[*i*]
4: 从内存分配点开始,向前遍历 *SVFG* //记录经过的节点集合 *FNode* 以及内存释放点集合 *dst*
5: 获取 *src*[*i*]的部分类型和特征信息存入 *V*[*i*],初始化 *BNode* 集合 //BNode 类似于二维数组
6: **for** (*j*=0; *j*<*dst.size*(); *j*++)
7: 从 *dst*[*j*]开始,向后遍历 *SVFG*,当前节点为 *CurNode*
8: **if** (*CurNode* in *FNode*)
9: *CurNode* 放入 *BNode*[*j*]集合中
10: **end if**
11: **end for**
12: 遍历 *FNode* 与 *BNode*,获取 *src*[*i*]的其余特征信息存入 *V*[*i*]
13: **end for**
**End**

如图 2 所示,我们给出了一个具体示例展示如何从源程序中提取特征.图 2(a)是 C 程序源码,图 2(a)左边为代码行号.图 2(b)是根据源码获取的特征以及对应的属性值.其中,$O_1$ 对应第 16 行内存分配点的特征信息,$O_2$ 对应第 5 行内存分配点的特征信息,$O_3$ 根对应第 6 行内存分配点的特征信息.根据图 2(b)可知.

- $O_1$ 所对应的特征 1、特征 4~特征 6、特征 9、特征 16 为 TRUE,表明该内存位置在循环中分配与释放,且分配与释放次数一致;特征 10 的值为 3,表示从内存分配点到释放位置经过 3 个函数.
- $O_2$ 所对应的特征 15、特征 16 均为 FALSE,表明该内存位置没有进行内存释放,也没有全局变量指向该内存位置.
- $O_3$ 所对应的特征 15 为 TRUE,特征 16 为 FALSE,表明该内存位置没有进行内存释放,但存在全局变量指向该内存位置.

```
1    char *p;
2    void f (){
3          bool t = false;
4          char ** buf = readBuf();
5          char *p 1 = malloc();
6          char *p 2 = malloc();
7          p =p2;
8          if (t)
9                printf ();
10         else
11               freeBuf( buf);
12   }
13   char ** readBuf(){
14         int ** Buf = malloc(10);
15         for ( n =0 ; n <10 ; n++)
16               Buf[ i ] = malloc();
17         return Buf;
18   }
19     void freeBuf( char **Buf){
20         for ( j =0 ; i <10 ; i++)
21               free (Buf[ i ]);
22         free (Buf);
23   }
```

(a) 输入程序

| 类别 | 序号 | 特征 | $O_1$属性值 | $O_2$属性值 | $O_3$属性值 |
|---|---|---|---|---|---|
| 类型信息 | 1 | 数组 | TRUE | FALSE | FALSE |
| | 2 | 结构体 | FALSE | FALSE | FALSE |
| | 3 | 链表 | FALSE | FALSE | FALSE |
| 分支信息 | 4 | 循环分配 | TRUE | FALSE | FALSE |
| | 5 | 循环释放 | TRUE | FALSE | FALSE |
| | 6 | 循环匹配 | TRUE | FALSE | FALSE |
| | 7 | 同一循环 | FALSE | FALSE | FALSE |
| | 8 | 链表匹配 | FALSE | FALSE | FALSE |
| | 9 | 分支条件 | TRUE | TRUE | TRUE |
| 释放信息 | 10 | 函数间距 | 3 | 1 | 1 |
| | 11 | 释放之前为空 | FALSE | FALSE | FALSE |
| | 12 | *p*指向对象数目 | 1 | 1 | 1 |
| | 13 | 别名数目 | 2 | 1 | 2 |
| | 14 | 指针偏移量 | 0 | 0 | 0 |
| | 15 | 全局指针 | FALSE | FALSE | TRUE |
| | 16 | 释放 | TRUE | FALSE | FALSE |

(b) 特征

Fig.2 Example of feature extraction

图 2 特征提取示例

2.2.2 内存泄漏检测

获取所有内存泄漏特征的数据集后,我们首先根据判定规则依次对数据集进行筛选,判断结果主要分为 3 类:疑似内存泄漏,表示为 T;可能非内存泄漏,表示为 F;无法判断,表示为 N.内存泄漏判定规则如下.

(1) 若特征 15、特征 16 全部为假(即在 SVFG 中,没有全局变量指向该内存分配点,也不存在内存释放语句),则判定结果为 T.

(2) 若特征 15 为真、特征 16 为假(即在 SVFG 中,存在全局变量指向该内存分配点,不存在内存释放语句),则判定结果为 N.

(3) 对于不满足第 1 条规则且不满足第 2 条规则的数据,提交给内存泄漏检测模型,得到内存泄漏检测结果.内存泄漏模型中检测结果分为两类:T 和 F.

根据判定规则(1)和规则(2):若在 SVFG 中没有全局变量指向该内存分配点,也不存在内存释放语句,则可直接判断为疑似内存泄漏;若在 SVFG 中存在全局变量指向该内存分配点,但不存在内存释放语句,由于全局变量可能在任何地方释放,我们不做判断,视为警报.因此根据规则(1)和规则(2),我们可直接判断数据对应的分类结果,无需使用内存泄漏检测模型.根据第 3 条规则,图 2(b)中 $O_1$ 所对应的特征应该输入内存泄漏检测模型.根据第 1 条规则,图 2(b)中 $O_2$ 所对应特征的判定结果为疑似内存泄漏.根据第 2 条规则,图 2(b)中 $O_3$ 所对应的特征则无法判断是否发生内存泄露.

提取数据集中判定结果为疑似内存泄漏的分配点,并结合内存位置信息给出漏洞报告.报告中标明疑似内存泄漏,并给出每个泄漏点的文件名、行号以及分配语句.

## 3 工具与评估

我们基于 LLVM 编译器(版本 4.0.0)实现我们的 C 程序内存泄漏智能化检测工具 I_Mem,工具框架如图 3 所示.我们的智能化检测工具 I_Mem 主要分为 3 个模块:内存泄漏检测模型、特征提取和内存泄漏检测.分别于第 2.1 节、第 2.2.1 节和第 2.2.2 节对应.

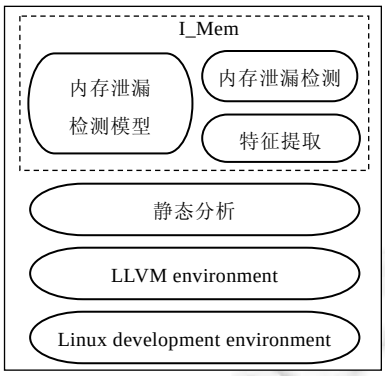


Fig.3 Framework of I_Mem

图 3 I_Mem 框架

在实验中,每个 C 程序的源文件都由 Clang 编译成 LLVM bitcode 文件格式,再由 LLVM Gold Plugin 进行合并,生成整个程序的 bc 文件.在模型构建阶段,我们使用的 SVM 分类器是 libSVM[22],随机机森林和决策树使用的是机器学习工具 weka[23].我们的实验分为两部分:一是在模型构建阶段对我们的模型进行评估;二是在特征提取与缺陷检测阶段对内存泄漏的检测结果进行评估.

### 3.1 模型构建阶段

我们从开源的 C 程序中提取真正的与虚假的内存泄漏实例来构建机器学习分类器,我们提取内存泄漏示

例的开源 C 程序主要有 icecast-2.3.1,cluster-3.0,droplet-3.0,wine-0.9.24 以及 SPEC 2000 中的 ammp,equak,然后,通过以下步骤获取真正的与虚假的内存泄漏.

(1) 关注源码中所有的内存分配点,通过添加或者注释内存释放点来获取真正的与虚假的内存泄漏.

(2) 仿照已经获取的内存泄漏实例,在源码中插入各种内存泄漏以构建丰富的训练集,尽量满足各个特征的属性.

通过以上两个步骤,我们可以获取大量的内存泄漏实例用于机器学习分类器的构建.模型构建阶段总过获取了 1 728 个内存泄漏实例(396 个虚假内存泄漏,1 332 个真正内存泄漏).构建模型中,我们采用五折交叉验证来确定准确率最高的分类器类型和参数.准确率是正确分类的样本数与总样本数之比.

我们使用五折交叉验证,在 1 728 个实例中进行模型训练与评估,对于 SVM、随机森林和决策树这 3 种机器学习算法,我们选取每种机器学习算法在训练过程中准确率最高的进行展示,见表 3.

**Table 3** Results of classification

**表 3** 分类结果

| 机器学习算法 | 分类准确率(%) |
|---|---|
| SVM | 97.7 |
| 随机森林 | 99.6 |
| 决策树 | 99.6 |

如表 3 所示,SVM 的分类准确率为 97.7%,随机森林与决策树的准确率 99.6%.因为决策树容易出现过拟合现象,因此我们选取随机森林作为我们内存泄漏检测模型.实验结果表明,我们构建的随机森林模型在对真实与虚假的内存泄漏进行分类时是有效的.

### 3.2 特征提取与缺陷检测阶段

我们的实验分为两部分.

- 第 1 部分的实验数据来自基准程序 Siemens[24].共有 4 个程序,这些实验对象的规模都比较小,存在的内存泄漏不多,因此,我们在源码中手工植入内存泄漏,特别是植入与数组、循环、链表等有关的内存泄漏,以验证内存泄漏模型在检测各种类型内存泄漏时的有效性.
- 第 2 部分数据来源于原生 SPEC 2000[25],我们选取了 3 个程序,以验证内存泄漏模型在检测大规模程序时的有效性.

实验对象见表 4.

**Table 4** Experimental subjects

**表 4** 实验对象

| 来源 | 名称 | 代码行数(KLOC) |
|---|---|---|
| Siemens | tcas | 0.17 |
| | replace | 0.56 |
| | print_tokens | 0.72 |
| | print_tokens2 | 0.57 |
| SPEC 2000 | vpr | 17.0 |
| | mesa | 49.7 |
| | vortex | 52.7 |

我们的方法是在 SVFG 的基础上提取内存泄漏特征,并利用机器学习技术进行内存泄漏检测,因此,我们选取 Saber 来比较本文方法和静态分析方法的效果.Saber 通过构建源码的 SVFG 来判断内存泄漏.在表 5 中展示了本文方法与 Saber 在 Siemens 程序上的对比实验结果.

根据表 5 中的结果,我们可以得到如下结论.

(1) 两种内存泄漏检测方法的漏报数目都比较低;

(2) 针对内存泄漏的一些特殊案例,如内存泄漏的分配、使用或者释放出现了循环、递归、链表等情况时,Saber 误报较多,例如 print_tokens 程序中的 34 个内存分配点,Saber 的误报是 10 个,本文只有 5 个.

**Table 5** Experimental results of Siemens

**表 5** Siemens 实验结果

| 程序 | 内存分配点 | Saber 漏报 | Saber 误报 | Saber 准确率(%) | I_Mem 漏报 | I_Mem 误报 | I_Mem 准确率(%) |
|---|---|---|---|---|---|---|---|
| tcas | 12 | 0 | 3 | 75 | 0 | 0 | 100 |
| replace | 26 | 0 | 8 | 69.2 | 0 | 3 | 88.5 |
| print_tokens | 34 | 0 | 10 | 70.6 | 0 | 5 | 85.3 |
| print_tokens2 | 46 | 1 | 14 | 67.4 | 1 | 5 | 87.0 |
| 合计 | 118 | 1 | 35 | 69.5 | 1 | 13 | 88.1 |

表 6 为 SPEC 2000 的实验结果.总结两部分实验,我们可以得到如下结论.

(1) 本文方法在静态分析的基础上利用机器学习算法提高了内存泄漏检测在特殊案例上的准确性.

(2) 本文的内存泄漏模型在检测大规模程序时是有效的.

**Table 6** Experimental results of SPEC 2000

**表 6** SPEC 2000 实验结果

| 程序 | 内存分配点 | Saber 漏报 | Saber 误报 | I_Mem 漏报 | I_Mem 误报 |
|---|---|---|---|---|---|
| vpr | 120 | 0 | 3 | 0 | 1 |
| mesa | 58 | 0 | 4 | 0 | 1 |
| vortex | 6 | 0 | 4 | 0 | 0 |
| 合计 | 184 | 0 | 11 | 0 | 2 |

### 3.3 讨 论

内存泄漏静态分析方法的缺点在于大规模程序误报多,需要人工确认.本文利用机器学习方法获取已有的知识经验,帮助提高内存泄漏静态分析的准确性.在选取内存泄漏特征时,我们研究内存泄漏机理,调研内存泄漏检测方法,从而确定内存泄漏相关特征,并构建内存泄漏检测模型.实验结果表明,本文方法确实有助于提高内存泄漏检测的准确率.在模型构建阶段,我们对构建的内存泄漏检测模型进行五折交叉验证,交叉验证结果准确率高达 95%以上.实验结果表明,我们构建的内存泄漏检测模型在对真实与虚假的内存泄漏进行检测时是有效的.在特征提取与缺陷检测阶段,在 Siemens 的实验数据上,我们的方法与 Saber 进行对比实验,Saber 的平均准确率只有 69.5%,我们的方法准确率高达 88.1%;在 SPEC 2000 的实验数据上,总共 184 内存分配点,Saber 的误报为 11 个,我们的误报只有 2 个.

综上所述,我们的实验结果表明,C 程序内存泄漏智能化检测方法在针对数组、循环等相关的内存泄漏时能够得到更加准确的检测结果.在静态分析的基础上,利用机器学习算法,使得内存泄漏检测结果更加准确.此外,实验中存在一些不足需要注意:目前,我们只针对 C 语言内存泄漏进行检测,并不支持 C++;实验中,我们所选取的是简单的基准程序,并在源码中插入一些内存泄漏特殊案例,并不保证该方法对内存泄漏其他特殊案例检测都具有较高的准确率.但是我们相信,实验的结果确实表明了本文的方法在检测内存泄漏上的可行性及准确性.

## 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工作:(1) 内存泄漏的静态分析;(2) 内存泄漏的动态检测;(3) 基于机器学习的缺陷检测.

### 4.1 内存泄漏的静态分析

内存泄漏通常是由于人为的对程序中动态内存的管理不当造成的,内存泄漏会导致内存空间被消耗,且在程序运行期间无法回收和重新利用.内存泄漏十分隐蔽,在程序运行初期不易被发现;但是对于长期运行的程序,特别是服务器,影响十分显著,它会降低程序性能,甚至导致程序在运行时崩溃.特别是在 C 语言中,内存的分配与释放都是人为控制的,内存释放这一步骤极容易被忽略,从而导致内存泄漏.

静态分析主要是根据特定的错误模式来查找内存泄漏,或者建立内存状态模型来进行内存泄漏检测.Cherem 等人[7]通过构建数据流图进行数据流分析,分析数值从内存分配点到内存释放点的路径中是否正确传

播来检测内存泄露.Saber 通过构建 C 程序的 SVFG 检测内存泄漏.Orlovich 等人[10]首先假设内存泄漏存在,然后进行反向数据流分析,验证该假设是否成立.RL_Detector[11]基于控制流图(CFG)进行数据流分析,利用静态符号执行对于每个资源(包括内存泄露)收集所有路径约束,通过路径约束计算数据流条件并检测资源泄露.Heine 等人[12]开发了一个描述指针隶属关系的模型,在该模型中,每个内存对象只能被 1 个拥有指针指向,因此该指针是唯一的且具有传递性,基于此对程序生成一种约束,用来检测内存泄漏.Cai 等人[26]提出了基于上下文无关文法可达性的由调用上下文向引用上下文自动转换的方法,用于辅助内存泄漏动态检测方法,从而提供对象引用路径等更丰富的报告信息,也为将来扩展我们方法中内存泄漏特征提供参考.

内存泄漏静态分析的优点是能够自动化运行,检测速度快;缺点是误报较多.目前,也有一些工作是在静态分析的基础上对内存泄漏进行修复.内存泄漏修复需要首先定位内存泄漏位置,因此,内存泄漏修复的准确性首先取决于内存定位的准确性,其次是插入释放语句的准确性.目前,主要的内存泄漏修复工作有:

- Leakfix[27]基于指针分析和数据流分析,判断内存泄露位置并进行内存泄漏的修复.Leakfix 能够保证为一个内存泄露生成多个修复程序,但不能保证生成的修复程序完全解决了该内存泄漏.
- AutoFix[28]是根据已有的静态分析警报,通过指针分析构建 VFG,再根据活性分析判断内存泄漏位置,进行内存泄漏的修复.Autofix 通过代码插桩进行内存泄漏的修复,并在一个沙箱中运行程序进行检查,确保修复的安全性.

内存泄漏的修复首先需要确保内存泄漏检测的准确性,不能对误报进行修复;其次,内存泄漏修复需要保证修复的正确性.Leakfix 在修复之后需要使用内存泄漏检测工具进行检测,AutoFix 则是自动的在修复之后运行程序进行安全性检查.因此,内存泄漏的修复受限于规模,如何提高内存泄漏修复的可扩展性以及准确性依然是一个难题.

本文的主要工作是在静态分析的基础上,利用机器学习技术进行内存泄漏检测,减少了静态分析的漏报和误报,提高了静态分析的准确性,尤其在针对内存泄漏的特殊案例时,能够显著提高内存泄漏检测的准确性.

### 4.2 内存泄漏的动态检测

内存泄漏的动态检测方法需要运行源代码,在程序运行过程中对内存分配、使用以及释放进行动态跟踪.LeakPoint[1]基于污点传播的思想监控内存对象的状态,追踪内存最后的使用位置以及失去引用的位置.DOUBLETAKE[2]将程序执行拆分为多个块,在每个块运行开始之前保存程序状态,在该块之行结束之后检查程序状态,判断内存是否发生错误.Sniper[3]利用处理器的监视单元(PMU)进行指令采样,来跟踪对于堆内存的访问指令,然后通过离线模拟器分析指令计算堆对象的陈旧度,并重新执行相关指令,捕获程序执行期间的内存泄漏.Omega[4]主要采用指针计数的思想记录内存对象的引用计数.

动态检测相对于静态分析更加准确,但是动态检测无法分析程序执行中不可达位置的错误.动态检测最大的缺点是效率低,耗时长.目前,代码规模和复杂度日渐增加,内存泄漏的动态检测效率远远无法满足工业界的需求,并且随着静态分析技术的发展,内存泄漏静态分析结果的准确率逐渐提高,静态分析的使用更加广泛.因此,本文的方法是基于静态分析,利用规则和机器学习技术进行内存泄漏检测,具有高效率和高可靠性.

### 4.3 基于机器学习的缺陷检测

目前,机器学习技术已被广泛运用于程序分析中以检测程序缺陷.Alatwi 等人[29]实现了一种检测安卓应用程序是否属于恶意软件的方法,其主要思想是将 apk 反汇编为源码,然后利用静态分析方法提取代码代征,并选取可用于模型预测的最佳特征组合,构建 SVM 分类器进行检测.Tac[20]主要利用机器学习算法来消除 typestate 和指针分析的差距,它从源码中提取 35 个特征,并训练 SVM 分类器以检测 Use-After-Free.Nagano 等人[30]对执行文件进行静态分析,然后利用机器学习的分类算法及自然语言处理技术来识别恶意软件.Grieco 等人[31]从二进制文件中提取程序静态特征,从程序执行中提取动态特征,然后利用机器学习技术训练模型来检测内存冲突.

本文的主要工作是通过静态分析提取内存泄漏特征,然后利用规则和机器学习模型检测内存泄漏.我们关注的重点是内存泄漏的特殊案例,因此在检测内存泄漏上准确性更高.

## 5 总结与展望

本文提出了一种 C 程序内存泄漏智能化检测方法,首先构建机器学习模型,然后在内存泄漏静态分析的基础上提取内存泄漏特征,并利用规则和机器学习模型进行内存泄漏的检测.实验结果表明,我们构建的内存泄漏检测模型在对真实与虚假的内存泄漏进行检测时是有效的.相对于目前的内存泄漏静态分析方法,本文方法在针对数组、循环等相关的内存泄漏时检测结果更加准确.

在本文方法的实验中,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) 目前,本文在特征选取与缺陷检测阶段选取的实验数据为简单程序且人为插入了各种内存泄漏,下一步需要选取一些开源的大型程序进行实验;(2) 本文目前关注的重点是 C 语言中与循环、结构体、数组以及链表有关的内存泄漏,可以扩展至 C++中,关注类以及各类容器有关的内存泄漏问题;(3) 当前的工作可以扩展到其他内存缺陷中,目前提取的特征适用于构建内存泄漏的分类器,我们可以提取更多的程序特征,构建多种内存缺陷检测的分类器;(4) 本文目前的静态分析方法是对每个内存分配点提取一个内存泄漏特征,我们可以进行细化,对于每个内存分配点的每条路径提取一个内存泄漏特征,这样,检测结果会更加准确,检测报告会更加详细.

**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] Liu T, Curtsinger C, Berger ED. DoubleTake: Fast and precise error detection via evidence-based dynamic analysis. In: Proc. of the 38th Int'l Conf. on Software Engineering. New York: ACM Press, 2016. 911−922. [doi: 10.1145/2884781.2884784]

[3] 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]

[4] Omega: An instant leak detector tool for valgrind. 2018. http://www.brainmurders.eclipse.co.uk/omega.html

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

[6] 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]

[7] 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]

[8] 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]

[9] 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]

[10] Orlovich M, Rugina R. Memory leak analysis by contradiction. In: Proc. of the 13th Int'l Conf. on Static Analysis. Berlin, Heidelberg: Springer-Verlag, 2006. 405−424. [doi: 10.1007/11823230_26]

[11] Ji X, Yang J, Xu J, Feng L, Li X. Interprocedural path-sensitive resource leaks detection for C programs. In: Proc. of the 4th Asia-Pacific Symp. on Internetware. New York: ACM Press, 2012. 1−9. [doi: 10.1145/2430475.2430494]

[12] 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]

[13] HP fortify. 2018. http://www.fortify.net/intro.html

[14] Coverity. The coverity static analysis tools. 2018. http://www.coverity.com/

[15] Klocwork. The Klocwork static analysis tool. 2018. http://www.klocwork.com/

[16] Steffen B, Knoop J, Rüthing O. The value flow graph: A program representation for optimal program transformations. In: Proc. of the 3rd European Symp. on Programming. Berlin, Heidelberg: Springer-Verlag, 1990. 389−405. [doi: 10.1007/3-540-52592-0_76]

[17] Cortes C, Vapnik V. Support-vector networks. Machine Learning, 1995,20(3):273−297.

[18] Breiman L. Random forests. Machine Learning, 2001,45(1):5−32.

[19] Safavian SR, Landgrebe D. A survey of decision tree classifier methodology. IEEE Trans. on Systems, Man, and Cybernetics, 1991, 21(3):660−674. [doi: 10.1109/21.97458]

[20] Yan H, Sui Y, Chen S, Xue J. Machine-learning-guided typestate analysis for static use-after-free detection. In: Proc. of the 33rd Annual Computer Security Applications Conf. New York: ACM Press, 2017. 42−54. [doi: 10.1145/3134600.3134620]

[21] Keerthi SS, Lin CJ. Asymptotic behaviors of support vector machines with Gaussian kernel. Neural Computation, 2003,15(7): 1667−1689. [doi: 10.1162/089976603321891855]

[22] LIBSVM—A library for support vector machines. 2018. https://www.csie.ntu.edu.tw/~cjlin/libsvm/

[23] Weka. 2018. http://www.cs.waikato.ac.nz/~ml/weka/

[24] Siemens. 2018. http://sir.unl.edu/

[25] SPEC: Standard performance evaluation corporation. 2018. http://www.spec.org

[26] Cai C, Zhang Q, Zuo Z, Nguyen K, Xu G, Su Z. Calling-to-reference context translation via constraint-guided CFL-reachability. In: Proc. of the 39th ACM SIGPLAN Conf. on Programming Language Design and Implementation. New York: ACM Press, 2018. 196−210. [doi: 10.1145/3192366.3192378]

[27] Gao Q, Xiong Y, Mi Y, Zhang L, Yang W, Zhou Z, 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]

[28] Yan H, Sui Y, Chen S, Xue J. AutoFix: An automated approach to memory leak fixing on value-flow slices for C programs. ACM SIGAPP Applied Computing Review, 2017,16(4):38−50. [doi: 10.1145/3040575.3040579]

[29] Alatwi HA, Oh T, Fokoue E, *et al*. Android malware detection using category-based machine learning classifiers. In: Proc. of the 17th Annual Conf. on Information Technology Education. New York: ACM Press, 2016. 54−59. [doi: 10.1145/2978192.2978218]

[30] Nagano Y, Uda R. Static analysis with paragraph vector for malware detection. In: Proc. of the 11th Int'l Conf. on Ubiquitous Information. New York: ACM Press, 2017. 1−7. [doi: 10.1145/3022227.3022306]

[31] Grieco G, Grinblat GL, Uzal L, *et al*. Toward large-scale vulnerability discovery using machine learning. In: Proc. of the 6th ACM Conf. on Data and Application Security and Privacy. New York: ACM Press, 2016. 85−96. [doi: 10.1145/2857705.2857720]

**朱亚伟**(1993－),男,安徽寿县人,硕士生,CCF 学生会员,主要研究领域为软件分析,软件测试.

**王林章**(1973－),男,博士,教授,博士生导师,CCF 杰出会员,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.

**左志强**(1986－),男,博士,助理研究员,CCF 专业会员,主要研究领域为系统软件,软件工程,程序语言,大数据系统.

**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工程.