定稿修改稿

软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software, [doi: 10.13328/j.cnki.jos.000000]

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

大规模代码增量式资源泄漏检测方法研究*

高志伟, 计卫星, 石剑君, 王一拙, 高玉金, 廖心怡, 罗 辉, 石 峰

(北京理工大学 计算机学院, 北京 100081)

通讯作者: 计卫星, E-mail: jwx@bit.edu.cn

摘 要: 资源泄漏是影响软件质量和可靠性的一种重要软件缺陷, 存在资源泄漏的程序长时间运行会由于资源耗尽而发生异常甚至崩溃。静态代码分析是进行资源泄漏检测的一种有效技术手段, 能够基于源代码或者二进制代码有效发现程序中潜在的资源泄漏问题。然而, 精确的资源泄漏检测算法的复杂性会随着程序规模的增加呈现指数级增长, 无法满足生产中即时对缺陷进行分析检测的实际应用需求。本文面向大规模源代码提出一种增量式的静态资源泄漏检测方法, 该方法支持过程间流敏感的资源泄漏检测, 在用户编辑代码的过程中从变更的函数入手, 通过资源闭包、指向分析过滤等多种技术手段缩小资源泄漏检测范围, 进而实现了大规模代码的即时缺陷分析与报告。实验结果表明, 本文所提出的方法在保证准确率的前提下, 90%的增量检测实验可以在 10s 内完成, 能够满足在用户编辑程序过程中对缺陷进行即时检测和报告的实际应用需求。

关键词: 质量保障; 缺陷检测; 资源泄漏; 指向分析; 数据流分析

中图法分类号: TP311

中文引用格式: 高志伟,计卫星,石剑君, 王一拙, 高玉金,廖心怡,罗辉,石峰.大规模代码增量式资源泄漏检测方法研究.软件学报. http://www.jos.org.cn/xxx

英文引用格式: Gao ZW, Ji WX, Shi JJ, Wang YZ, Gao YJ, Liao XY, Luo H, Shi F. Incremental Resource Leak Detetection for Large Scale Source Code. Ruan Jian Xue Bao/Journal of Software, 2017 (in Chinese). http://www.jos.org.cn/xxx

Incremental Resource Leak Detetection for Large Scale Source Code

GAO Zhi-Wei, JI Wei-Xing, SHI Jian-Jun, WANG Yi-Zhuo, GAO Yu-Jin, LIAO Xin-Yi, Luo Hui, SHI Feng

(School of Computer Science and Technology, Beijing Institute of Technology, Beijing 100081, China)

Abstract: Resource leak is an important software defect that affects the quality and reliability of software. The existence of resource leak in program running for long time will cause abnormal or even collapse of resource depletion. Based on the source code or binary code, static code analysis is an effective appraoch to resource leak detection in the program. However, as the scale of the programs increases the complexity of the accurate resource leak detection algorithm can not meet the real-time detection of defects in practical

* 基金项目: 国家重点研发计划(No.2016YFB1000801)

收稿时间: 0000-00-00; 修改时间: 0000-00-00; 采用时间: 0000-00-00; jos 在线出版时间: 0000-00-00
CNKI 在线出版时间: 0000-00-00

applications. In this paper, we propose an incremental static resource leak detection algorithm for large-scale source code projects. This algorithm supports interprocedural flow-sensitive resource leak detection. During the process of code editing, our approach starts from the modified functions and achieve the hundreds of thousands of lines of real-time defect analysis by generating method closures, performing points-to analysis to narrow down the scope of resource leak detection. Based on small test benchmarks and large scale open source projects, compared with the existing tools, our method proposed in this paper finished more than 90% of the experiments within 10s. Our experimental results show that incremental resource leak detection can meet the practical requirements of detecting and reporting defects in the process of coding.

Key words: quality assurance; defect detection; resource leak; points-to analysis; data flow analysis

7 引言

随着云计算和大数据分析技术的不断发展与应用，软件系统的规模日益庞大，结构日益复杂，参与程序设计与开发的人员众多，从而使软件系统的可靠性受到严重影响。因此，如何保证大规模软件的可靠性和稳定性是目前学术界和工业界普遍关注的问题[1]。在众多的软件缺陷和安全漏洞当中，资源泄漏是影响软件系统稳定性和可靠性的重要软件故障之一。资源相关的缺陷是程序对资源对象在调用、分配和回收等操作上的错误导致的缺陷[2]。本文所提到的资源主要指的是系统资源，例如文件句柄、网络连接、数据库连接、远程方法调用等。用户程序向系统申请资源之后，如果在程序中存在一条执行路径未对所申请的资源进行显式释放，则称该路径上存在资源泄漏。由于系统资源有限，当程序中存在资源泄漏时，系统资源会随着程序的运行逐渐耗尽，直至程序出现异常或者崩溃。研究表明，导致系统停机的缺陷中有 86%是由于资源泄漏引起的[3]。

静态分析是目前常见的代码缺陷检测技术，该方法不需要运行程序，检测过程无需人工干预，只需要扫描程序全部或者部分代码即可发现潜在的软件缺陷和安全漏洞。相比动态检测技术，静态分析方法具有更高的覆盖率，因此，研究人员提出了大量静态分析方法用以完成代码缺陷检测。目前针对资源泄漏提出了多种检测方法[4][5]，也出现了例如 Fortify、Coverity 和 Klocwork 等多款商业软件，开源软件 FindBugs[6]也实现了对方法内资源泄漏的分析与检测。

分析精度和分析效率是衡量静态分析工具的主要指标。根据 Rice 定理[7]，静态分析对程序的任何非平凡属性不可能做到既是可靠的又是完备的，从而导致计检测结果中会出现误报（False Positive）和漏报（False Negative）的情况。大量的误报和漏报会使用户失去使用工具的信心，而通常提高分析的精度（例如路径敏感和上下文敏感的相关算法）则会大幅提高分析的复杂度[8]。静态分析的效率是影响其能否应用于大型程序进行缺陷检测的关键，它与分析过程中的计算复杂度密切相关。特别是针对几十万行甚至上百万行的大规模代码，路径敏感和上下文敏感分析会使得分析规模呈指数级增长，实施一次过程间的全局分析经常需要几个小时，甚至几天的时间。然而 Boehm 明确指出，修正错误的代价随时间几乎是呈指数增长[9]，因此软件故障发现的越早代价越小。如何针对大规模代码进行精确的即时缺陷检测，已成为学术界和工业界急需解决的关键问题之一。

本文针对大规模代码提出一种增量式即时资源泄漏检测方法。首先，该方法动态维护程序的全局函数调用图以及资源相关的类图，以用户当前编辑的函数为入口点进行增量式缺陷检测，基于路径敏感的过程间分析对增量代码进行检测，能够及时发现潜在的资源泄漏缺陷；其次，该方法考虑函数上下游调用关系，对方法进行资源闭包求解，并利用别名分析技术进一步缩小检测范围，提高检测的准确性和即时性；最后，该方法进行过程间的流敏感数据流分析，与过程内的检测方法相比，能够大幅降低检测的漏报率和误报率。

本文首先在第 2 节介绍了已有研究工作，第 3 节详细阐述了面向大规模代码的资源泄漏增量检测方法，第 4 节对算法的具体实现进行了介绍，第 5 节对所提出的方法进行了评估，并与现有的工具进行了对比和分

析，最后第 6 节对论文工作进行了总结。

8 相关工作

资源泄漏检测是备受关注的研究热点，国内外许多学者和企业都投入大量的精力去研究能够检测安全漏洞的的工具。

肖庆和宫云战等人提出一种多项式复杂度的路径敏感的静态缺陷检测方法，该方法采用变量的抽象取值范围来表示属性状态条件；通过属性状态条件中变量取值为空来判断不可达路径；通过在控制流汇合节点上进行相同属性状态的属性状态条件合并来降低计算复杂度[4]。论文[5]从程序中提取函数摘要，概括函数的资源行为，在分析中用函数摘要模拟函数调用的效果。通过函数间分析、改进异常处理和资源别名分析，减少误报数，提高故障检测的准确率。

IBM 的 Emina Torlak 等人在 2010 年提出了一种方法间资源泄漏检测的方法[10]。基于 access-path，该方法构建了一个在数据流分析中表示资源类对象的三元组，并随着数据流分析构建其在每条语句下的别名集。该方法能够解决跨方法间的别名分析，使得数据流分析更准确。虽然该方案在全局检测中有不错的效果，但是并没有给出一种有效增量式检测方案。

FindBugs[5]作为一个基于规则的代码缺陷检测工具，因其轻量、高效的特点被广泛使用。其检测的缺陷包括资源泄漏、空指针引用、跨站脚本、SQL 注入等在内的多种代码缺陷。但是 FindBugs 使用的是基于规则的策略，通过发掘各种缺陷的发生模式，制定相应的检测策略。这种方案对数据流传播路径不敏感，所以检测的准确率较低，另外，FindBugs 只支持方法内的资源泄漏检测。

Lisa Nguyen Quang Do 等人提出了分层检测的基本思想[11]，将缺陷的检测分为方法、类、文件、包和项目等多个不同的层次。在检测的过程中，从方法内开始，到项目级逐级进行检测，并及时报告发现的不同层次的缺陷。分层检测的思想能够对较小范围内的缺陷进行即时检测并报告给用户，但是也存在误报和漏报的情况，以图 1 的代码片段为例，当用户正在编辑 *foo* 方法时，如果从当前方法开始向下游分析，会发现 *input* 是被关闭的，不存在资源泄漏的问题，但是加上 *main* 函数一起分析，则会发现 *input* 被改写了一次，且改写之前 *input* 指向的资源变量并没有被关闭，所以，资源泄漏发生与否不仅取决于当前函数，还包括了与这个函数相关的上下游函数。文献所给出的算法只考虑以当前函数为起点向下调用函数，但是忽略其上游的函数，从而会造成误报或者漏报的情况。已有的商业工具更多采用批量式检测方法，即从程序的 *main* 函数入口进行检测，从而不存在上游函数调用的问题。另外，该方法随着检测层次的逐渐增加，检测的时间也会增加。

```
public class Main {
    public static void main(String[] args) throws IOException {
        FileInputStream input = new FileInputStream(new File("1.txt"));
        foo(input);
    }
    public static void foo(FileInputStream input)  {
        try {
            input = new FileInputStream(new File("2.txt"));
            foo1(input);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
    public static void foo1(FileInputStream input) {
        try {
            input.read();
            input.close();
        } catch (IOException e) {
            if(input!=null){
                try {
                    input.close();
                } catch (IOException e1) {
                    e1.printStackTrace();
                }
            }
        }
    }
}
```

Fig 1. Resource leak code snippet

图 1 资源泄漏代码片段

9 大规模代码资源泄漏增量检测算法

传统的资源泄漏检测算法，在每次检测过程中，都会对整个项目进行资源泄漏的分析检测，这种检测通常适用于代码量较小的项目；而对于大规模代码项目检测效率则会受到严重影响。尤其是在代码修改量远小于原始代码量的情况下，采用传统的资源泄漏检测算法，耗费大量的时间进行冗余检测分析，极大地降低了检测效率。

本文提出的针对大规模代码的增量式资源泄漏检测算法，是在进行资源泄漏检测的过程中，通过逐步缩小待分析资源相关方法的范围，并对未进行资源操作的相关方法进行“剪枝”，从而避免非资源相关方法的冗余分析，提高资源泄漏检测的效率和准确性。

9.1 分析准备工作

在对大规模代码进行资源泄漏增量式检测过程前，首先需要对项目进行一个全局检测，在全局检测过程中，会构建全局方法调用图和资源相关的类图，并在增量式检测过程中，对方法调用图和资源类包含图进行动态维护。

方法调用图是从项目中抽取出所有方法的调用关系图。对于方法调用图 $G=(M, E)$，其中 M 表示从项目中提取的方法集合，E 表示方法之间的调用关系。G 构建出了项目中所有方法的上下文关系，从一个方法 M_i 出发，可以找出所有调用方法 M_i 以及被 M_i 调用的方法。

资源类图是从项目中抽取的所有与资源相关的类，以及这些类之间的关系图。对于资源类包含图 $C=(N, S)$，其中 N 表示资源类的集合，S 表示资源类之间的关联关系。给定资源类 A 和资源类 B，若 A 的成员变量中存在一个 B 类的对象，则资源类包含图中有一条边 s 由 A 指向 B。若 B 是 Java 中资源相关的基类（即 Java 提供的资源操作的类），而 A 是项目中的自定义类，则 B 称为直接资源类，A 称为间接资源类。如图 2 所示，

java.io.InputStream 和 java.io.OutputStream 是直接资源类，A、B 和 C 为间接资源类，D 和 E 为非资源类。BR_A 表示 A 直接或间接操作的直接资源类集合，BR_A={java.io.InputStream，java.io.OutputStream}，同理 BR_B={ java.io.InputStream}，BR_C={ java.io.OutputStream }。

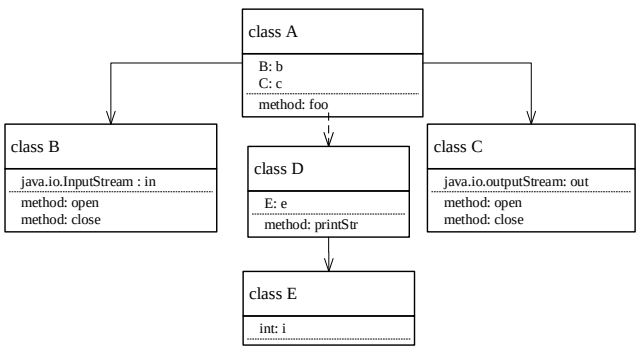


Fig 2. Resource related class diagram

图 2 资源相关类图

为了便于后面分析，定义每个类中的方法和成员变量如下：

```
class A{
    B b;
    C c;
    public void foo() throws IOException {
        b = new B();
        b.open();
        b.close();
        c = new C();
        c.open();
        c.close();
        new D().printStr();
    }
}
class B{
    java.io.InputStream in;
    public void open() throws IOException {
        in = new FileInputStream(new File("b.txt"));
        OutputStream out = new FileOutputStream("b");
    }
    public void close() throws IOException {
        in.close();
    }
}
class C{
    java.io.OutputStream out;
    public void open() throws IOException {
        out = new FileOutputStream("c.txt");
    }
    public void close() throws IOException {
        out.close();
    }
}

class D{
    E e;
    public void printStr() {
    System.out.println("Hello");
    }
}
class E{
    int e1;
}
```

Fig 3 Definition of Class A, B, C, D and E

图 3 类 A、B、C、D 和 E 的定义

根据上面个类定义，构建的的方法调用关系图，如图 4 所示。

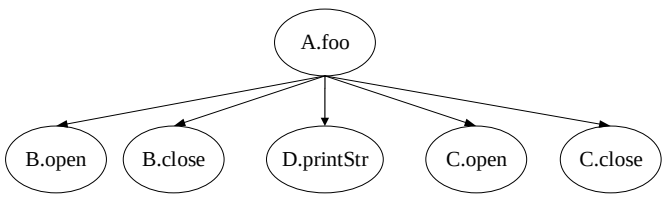


Fig.4 Method call diagram

图 4 方法调用关系图

9.2 资源闭包分析

资源闭包分析是在生成的方法调用图和资源类图基础之上，通过缩小资源泄漏的检测范围，消除传统检

测算法中大量冗余计算而造成的性能下降，从而提高检测效率。一个资源闭包指基于某种特定的规则有机结合起来的，并具有调用关系的一组方法集合。

如算法 1 所示，对于类 A，BR_A 表示 A 的实例能够直接或者间接操作的直接资源类的集合，在构建资源类图的时候会计算得到所有资源类直接或间接操作的资源类集合。对于一个给定的方法 m，为了求解该方法的资源闭包 closure(m)，首先需要从该方法操作的所有数据对象集合 S_{obj} 出发，通过 S_{obj} 找出 m 操作的直接资源类集合 R_m；然后根据方法调用图，从方法 m 开始，进行 m 的资源闭包求解。对于任意的方法 m'，如果 m'与 m 存在直接的或者间接的调用或被调用关系，且 m'和 m 操作的资源对象对应的类包含的直接资源类有重合，就会被加入到 m 的资源闭包 closure(m)中。

算法 1：计算方法 *m* 资源闭包 *closure(m)*
输入：方法调用图 *G=(M, E)*，资源类图 C=*(N, S)*，待分析方法 *m*
输出：*closure(m)*

```
1  begin
2    closure(m) ← {m}
3    S_obj ← m 操作的所有数据对象实例集合
4    R_m ← Ø  // 记录方法 m 能够直接或间接操作的直接资源类的集合
5    foreach obj ∈ S_obj do
6        if obj instanceof Class X and X ∈ N then
7            BR_X ← 图 C 中类 X 可达的直接资源类的集合
8            R_m ← R_m ∪ BR_X
9        end if
10   end for
11   M ← closure(m)
12   while M != Ø do
13       T ← Ø
14       foreach n ∈ M do
15           if ((n,m') ∈ E or (m', n) ∈ E) and m' ∉ closure(m) then
16               S'_obj ← m'操作的所有数据对象实例集合
17               foreach obj ∈ S'_obj do
18                   if obj instanceof Class X then
19                       BR_X ← 图 C 中类 X 可达的直接资源类的集合
20                       if BR_X ∩ Rm ≠ Ø then
21                           closure(m) ← closure(m) ∪{ m' }
22                           T ← T ∪{ m' }
23                       end if
24                   end if
25               end for
26           end if
27   end for
```

```
28        M ← T
29    end while
30    return closure(m)
31 end
```

由于全局方法调用图和资源类图是一次性事先构建好的，在代码增量修改中只需要进行少量同步更新与维护，因此算法 1 的运行时开销非常小。

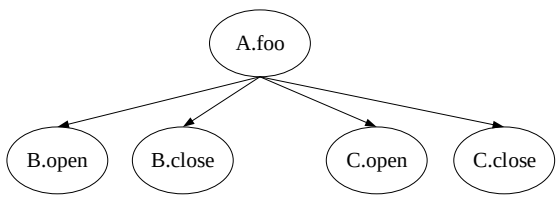


Fig.5 The generation of method closure for B.open

图 5 方法 B.open 的资源闭包

根据 3.1 中定义的方法调用图，假设当前正在编辑方法 *B.open*，根据算法 1，首先找到 *B.open* 中包含的直接资源类，即 *InputStream* 和 *OutputStream*，所以 $R_{B.open}$*={InputStream,OutputStream}*，再搜索与 *B.open* 存在调用关系的方法，即 *A.foo*，由于 *A.foo* 方法中包含类 A 和类 B 的对象，所以 $R_{A.foo}=BR_A \cup BR_B$,即 $R_{A.foo}$*={ InputStream,OutputStream}*，$R_{B.open}$ 和 $R_{A.foo}$ 的交集不为空，所以，将 *A.foo* 加入到 *closure(B.open)*中，然后扫描与 *A.foo* 存在调用关系的方法，按照算法进行下去，最后求得的 *closure(B.open)*，如图 5 所示方法即为算法 1 求得的方法 *B.open* 的资源闭包。

9.3 资源对象指向分析

3.2 节中的资源闭包分析得到操作相同类型资源的所有方法，然而有可能这些方法操作的是多个不同的资源对象，例如图 3 中的方法 *B.open* 和方法 *C.open*，虽然都包含了 *OutputStream* 对象，但是不是同一个实例，因此在分析方法 *B.open* 时，并不需要分析方法 *C.open*。在计算资源闭包时，可以进一步利用资源变量的指向分析得到操作同一个资源对象的资源闭包。而在指向分析的过程中，首先以方法为单位获得一个方法中所有资源变量的指向集合的并集，接着判断方法之间的指向集合的交集是否为空即可排除无关方法，进一步缩小检测范围。

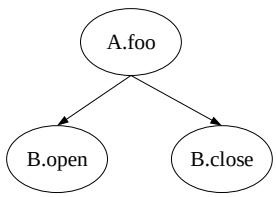


Fig.6 The generation of method closure for B.open after points-to analysis

图 6 通过指向分析得到的 *B.open* 的资源闭包

若要得到与方法 *B.open* 中具有相同资源变量引用对象的所有相关方法，则需要首先提取方法 *B.open* 中

所有的资源变量，并构建其指向集，然后搜索所有与方法 *B.open* 中资源变量指向集有交集的方法。最后得到如图 6 所示的结果，从而进一步缩小资源泄漏的检测范围，提高检测效率。算法的具体描述如下所示，其中 reachingObjects(obj)表示 obj 在内存中的一个指向集。

算法 2：使用指向分析对资源闭包进行过滤
输入：待分析方法 *m* 及其资源闭包 *closure(m)*
输出：过滤后的资源闭包 *closure(m)*

```
1  begin
2     ptsA ← ∅
3     S_obj ← m 操作的所有数据对象变量集合
4     foreach obj ∈ S_obj do
5          if obj instanceof Class X and X ∈ N then
6               ptsA ← ptsA ∪{ reachingObjects(obj) }
7          end if
8     end for
9     foreach m' ∈ closure(m) do
10         ptsB ← ∅
11         S_obj ←m'操作的所有数据对象变量集合
12         foreach obj ∈ S_obj do
13              if obj instanceof X and X ∈ N then
14                   ptsB ← ptsB ∪{reachingObjects(obj)}
15              end if
16         end for
17         if ptsA ∩ ptsB = ∅ then
18              closure(m) ← closure(m) −{m'}
19         end if
20    end for
21    return closure(m)
21 end
```

假设 *m* 为用户修改的方法，该算法首先计算方法 *m* 可能操作的资源对象的指向集合，存储在 *ptsA* 中；其次对于 *closure(m)*中的任一非 *m* 的方法 *m'*，计算 *m'*可能操作的所有资源对象指向集合，存储在 *ptsB* 中；并判断 *m* 是否有可能与 *m'*操作同一个资源对象，即 *ptsA* 与 *ptsB* 是否有交集，如果没有交集，则将 *m'*从 *closure(m)* 去除掉。该算法对方法操作的资源对象指向集合两两求交集，而非所有方法资源对象集合一起求交集，因而可能会将一些非实际相关的方法加入到检测集合中来，但是由于在检测阶段会进行精确的数据流分析，因此并不影响最终的检测结果。

9.4 资源泄漏检测

根据生成的资源闭包，找出其入口方法，从入口方法开始进行资源泄漏检测。一个资源闭包的入口方法

定义为一个资源闭包中只有调用边而没有被调用边的方法。一个资源闭包可能包含一个或多个入口方法，资源泄漏的检测须从多个入口方法开始，利用 IFDS/IDE 资源泄漏检测算法进行检测。

如图 6 中的方法调用图，只存在一个入口，即方法 *A.foo*，在用户编辑方法 *B.open* 时，通过资源闭包分析求得方法 *B.open* 的资源闭包为 *A.foo*，*B.open* 和 *B.close* 构成的方法集，其中 *A.foo* 方法是入口方法，在资源泄漏检测时，只需要从 *A.foo* 开始分析，分析资源闭包中的方法即可。

本文提出的资源泄漏增量检测算法，以划分和生成资源闭包的形式，将资源泄漏的全局检测范围逐步划分到以资源闭包为单位的检测，以缩小检测范围；在生成资源闭包的过程中，对未引用资源操作的方法进行“剪枝”，从而避免非资源相关方法的冗余分析；另外，在资源闭包内进行资源变量的指向分析，进一步构建出范围缩小的资源闭包，从而提高资源泄漏检测的效率和准确性。增量检测算法的具体过程，主要分为以下几个步骤：

(5) 生成增量方法调用图和类图：获取当前项目中被修改的所有方法集合和相关类集合，根据相关方法和类的增加、删除、修改操作，动态修改方法调用图和类图;

(6) 生成资源相关方法的资源闭包：对于所有被增量修改的方法，判断是否包含资源类变量，过滤出所有包含资源变量的方法集合；从所有操作资源变量的方法向外扩展，找出操作相同资源类的方法，形成与被修改方法中包含相同直接资源类的资源闭包;

(7) 指向分析：若被修改方法的资源闭包中包含的方法数目大于阈值 *k*，在资源闭包内找出操作相同资源对象的所有方法，生成更小的资源闭包；

(8) 资源泄漏检测：从资源闭包入口开始，利用过程间的资源泄漏检测算法进行检测。

增量检测框架如下所示：

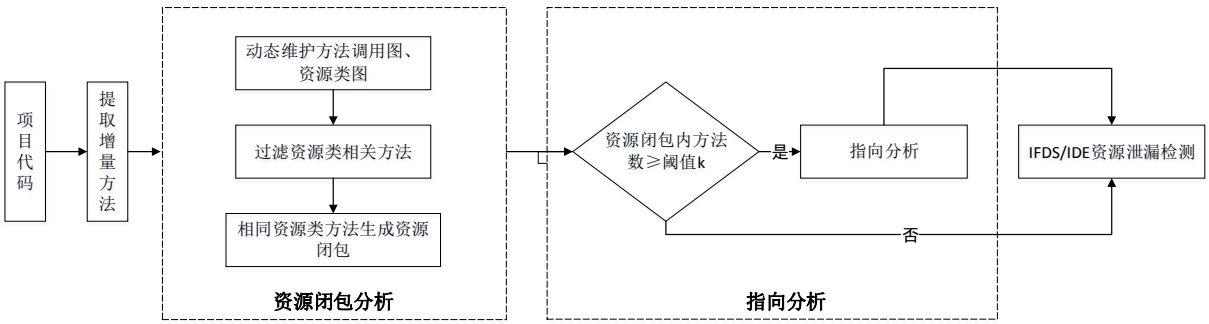


Fig.7 A framework for incremental resource leak detection

图 7 资源泄漏增量检测框架

10 算法实现

本文针对 Java 语言的资源泄漏检测实现了一个 Eclipse 插件。该插件基于 Emina Torlak 等人在 2010 年提出的方法间资源泄漏检测的方案[10]实现，并加入了本文所提出的增量式检测方法，支持 Eclipse 中项目的全局检测，同时也支持用户在编辑过程中，对增加和修改的代码进行增量的检测，被检测代码必须是语法正确的，且能成功编译为 class 文件。其检测方案的实现是基于 Soot[12]实现的静态分析。静态分析是指在软件不运行的前提下进行的分析过程[13]。Soot 是 Eric Bodden 等人开发的一款用于 Java 程序分析的开源框架，能够提供 Java 代码的数据流分析，控制流图的构建，Baf[14]、Shimple[15]、Grimp[16]和 Jimple[17]等中间表示的生成以及指向分析等诸多功能。Heros 是基于 Soot 开发的一种针对 IFDS/IDE 问题的解决方案，其主要特点包括：

支持解决 IFDS/IDE 问题；基于多线程实现，具有良好的扩展性；提供了简单的程序接口，使用方便；具有普遍性，可以针对多种语句进行程序分析。

Soot 对于 Java 类文件的加载和解析是一个比较耗时间的过程，因此在增量检测过程中，对 Soot 的类加载过程做了一些优化，实现了 Soot 的对类文件增量加载机制，极大的减少 Soot 类的加载时间。Soot 提供了 PointsToAnalysis 和 PointsToSet 接口，PointsToAnalysis 接口包含了方法 reachingObjects(Local l)方法，这个方法可以返回变量 l 指向的集合，该集合实现了 PointsToSet 接口。PointsToSet 接口包含了判断该集合是否与其他指向集合相交的方法。

Spark 是 Soot 提供的一种用于指向分析的框架，指向分析是计算给定变量的所有可能指向的内存区域的集合。利用该分析框架可以对别名分析以及提高方法调用图的准确度提供帮助。利用 spark 提供的指向分析功能，可以在增量检测过程中找出与改动方法相关联的资源闭包。如果仅仅考虑资源泄漏情况，可以只针对资源类型的变量进行分析，从而进一步缩小该分析闭包。

在对 Java 项目的分析中是以方法作为一个分析单位的，所以需要构建用于跨方法分析的方法调用图。在方法调用图的构建过程中，由于 Java 的语言特性，如抽象、接口、继承等，其声明的对象都是在实际运行中进行后期绑定解析才能确定属于哪一个类对象，所以对于方法调用图的构建需要基于 Soot 提供的指向分析机制，分析每个方法中调用语句中对象的指向关系，确定其运行时实际对应的类，然后确定其调用的方法。

11 实验评估

11.1 实验设置

在本次评估中，使用的测试环境为 Windows 7 64 位系统，4 核 CPU 以及 8GB 内存。Eclipse 版本为 4.5.2，使用的 JDK 版本为 1.8。

本文中测试用例一共分为两类，大型测试用例和小型测试用例。其中大型测试用例为 Tomcat 和 Weka。Tomcat 是 Apache 软件基金会的一个程序，其开源性、性能稳定并且免费的特点广受各位 Java 爱好者的喜爱，是目前较为流行的 web 应用服务器。Weka 是一款免费的、开源的、非商业化，基于 Java 环境下开源的机器学习以及数据挖掘软件。而本文的小型测试用例由两部分构成，DroidLeaks[18]是相关研究人员从 34 个现实生活中开源 Android App 中抽取出来的真正存在的资源泄漏测试集，由于 DroidLeaks 着重于 Android 相关的测试用例，本文只抽取了测试集中与 Java 包有关的测试用例；此外，本文作者编写了一些关于资源泄漏的测试用例（Defects-bench），各个测试用例的详细说明如表 1 所示：

Table 1 Details of benchmarks

表 1 各类测试用例详细说明

大型测试用例			小型测试用例		
测试用例名	Java 文件个数	代码行数（万行）	测试集名称	测试用例个数	代码行数（行）
Tomcat-8	1591	31	DroidLeaks	16	958
Weka	1568	50	Defects-bench	71	2132

DroidLeaks 中共有 16 个测试用例，其中包含资源泄漏缺陷的测试用例 15 个，没有资源泄漏的测试用例 1 个；Defects-bench 中共有 73 个测试用例，其中包含资源泄漏缺陷的测试用 53 个，不存在资源泄漏的测试用例 18 个。由于 DroidLeaks 主要包含方法内的资源泄漏，所以测试不够全面，因此本文自己设计了更全面的的资源泄漏测试用例 Defects-bench，从 Java 语言的各个方面考虑，包含多态、反射、内部类、静态成员、

集合、别名、try-with-resource 和方法间的数据传递等各种特性，用以保证检测测试全面性。

FindBugs 是一个静态分析工具，它通过一组缺陷模式对比分析来发现可能的问题。FindBugs 提供对 Eclipse 的无缝插件集成，可以即时查找代码存在的缺陷。Fortify SCA 是一个静态的、白盒的软件源代码安全测试工具。它通过内置的五大主要分析引擎：数据流、语义、结构、控制流、配置流等对应用软件的源代码进行静态的分析。Fortify 支持多达 21 种编程语言代码的检测。本文分别使用这两种工具对测试用例进行检测，并与本文提出的方法得到的检测结果进行对比。其中 FindBugs 的版本为 3.0.1，Fortify 的版本为 5.1。其中 Fortify 由于版本原因，检测时最高的 JDK 版本为 1.6。

11.2 全局检测对比分析

本文将小型测试用例分别使用 FindBugs、Fortify 及本文方法进行分析检测，对每一个测试结果进行人工检测，将最后的结果统计成正确率、漏报率和误报率，统计结果如表 2 和表 3 所示：

Table 2 Detecting results using DroidLeaks

表 2 DroidLeaks 检测结果

方法	正确率(%)	漏报率(%)
FindBugs	6.25	93.75
Fortify	87.50	12.50
本文方法	93.75	6.25

Table 3 Detecting results using Defects-bench

表 3 Defects-bench 检测结果

方法	正确率(%)	漏报率(%)	误报率(%)
FindBugs	38.03	84.65	1.41
Fortify	59.15	39.44	0.00
本文方法	61.97	35.21	1.41

评估中准确率包括三部分：正确率，漏报率和误报率。正确率是指测试用例的确存在着资源泄漏问题而工具也检测出来该问题（True Positive）和测试用例没有资源泄漏的问题工具也没有检测出来问题（True Negative）个数的和，漏报是指测试用例存在资源问题但是检测工具没有并没有检测出来该问题（False Negative），误报是指测试用例不存在资源泄漏问题但是方法检测出来存在问题（False Positive）。从表中可以看出在两个测试用例集中本文方法表现出比 FindBugs 和 Fortify 更高的正确率，较低的误报及漏报率，这是由于本文应用了过程间路径敏感的数据流分析和别名分析。FindBugs 漏报率较高的原因是其不支持方法间的资源泄漏检测。

11.3 增量检测分析

在进行增量检测时，本文对召回率的定义是：针对每个全局检测中检测出来的问题，假设每个方法为当前编辑方法，从每个方法开始进行增量检测，看是否能够再次检测到这些问题。本文对大型测试用例进行时间评估和召回率评估，对小型测试用例只进行召回率的评估。在本实验中，设置使用指向分析的阈值 K 为

200。

图 8 和图 9 分别表示 Tomcat 和 Weka 在增量检测中每个方法完成检测的时间散点图分布。横坐标表示入口方法序号，纵坐标则表示从该方法开始进行增量检测完成检测的时间。图 10(a)和 10(b)分别表示 Tomcat 和 Weka 在增量检测中每个方法完成检测的时间分布。

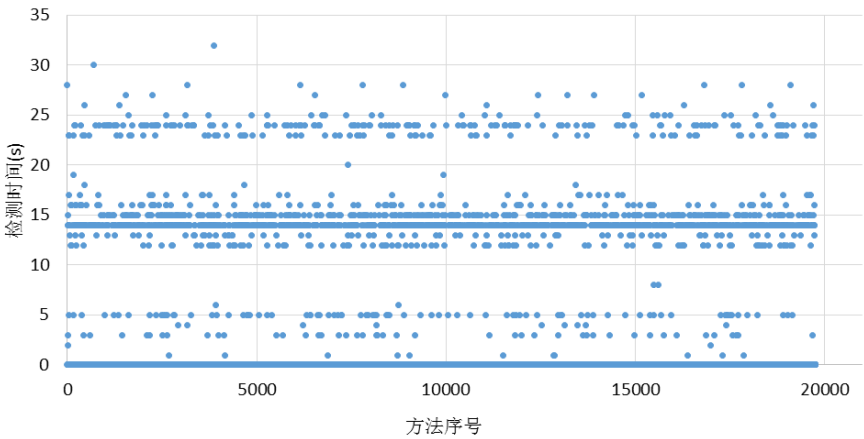


Fig 8 Tomcat's time distribution of incremental detection

图 8 Tomcat 中方法增量检测时间分布

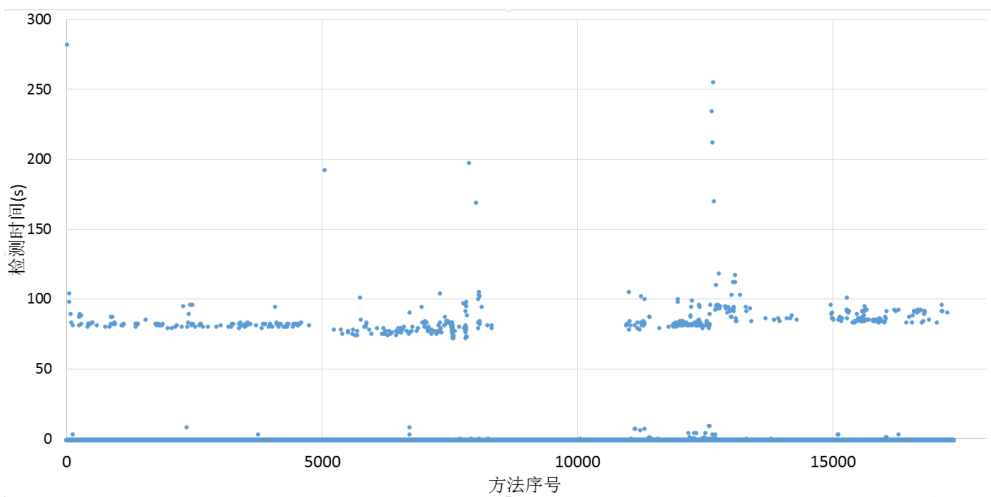
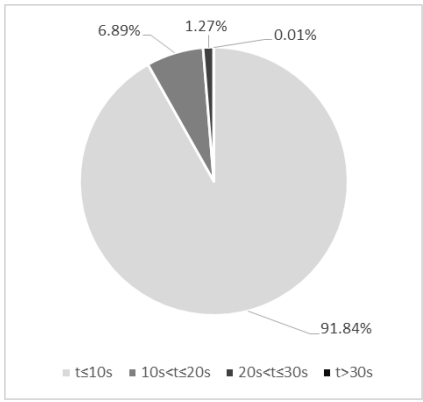
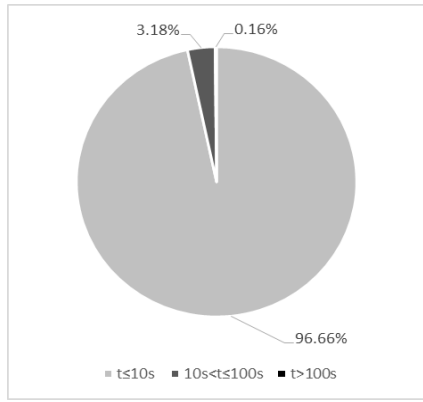


Fig. 9 Weka's time distribution of incremental detection

图 9 Weka 中方法增量检测时间分布





10(a) 10(b)

Fig.10 Time proportion distribution in incremental detection

图 10 方法增量检测时间占比分布

根据图 8 和图 9，在 Tomcat 近 2 万个方法中，绝大多数的方法增量检测都是在 30s 内完成，只有极个别的方法执行时间超出 30s，但也都控制在 35s 以内。而由于 Weka 比 Tomcat 超出了接近 20 万行代码，导致 Weka 的部分方法检测时间超过了 100s，但是从图 10（a）和 10（b）的时间分布图来看，Tomcat 和 Weka 检测时间小于 10s 的方法占比仍达到 90%以上，因此，大多数情况下仍可以达到即时检测和报告的效果。

Table 4 The recall ratio of different benchmark in incremental detection

表 4 不同测试用例集在增量检测中的召回率

测试用例名	召回率(%)
DroidLeaks	100.00
Defects-bench	100.00
Tomcat	98.26
Weka	87.87

表 4 给出了增量检测相对于批量全局检测的召回率情况，可以看出对于小规模测试程序，本文所提出方法虽然实施的增量检测，仍然能够达到 100%的召回率；但是对于 Weka，仍然能够获得 87%以上的召回率，亦即全局批量检测中超过 87%的资源泄漏可以在增量检测中被即时检测到。召回率降低主要是由于方法调用图在增量代码分析的时构建不完整造成的。在 Java 项目中，由于多态的特性，从某一个方法出发，并不能确定某些对象实际对应的类，所以也不能确定其对应的成员方法，在进行增量分析时，构建的方法调用图不完整，故召回率达不到 100%。因此，实际应用中可以采用增量和批量相结合的方式，既解决了检测时间的问题，也保证了较高的准确率。

12 总结

本文针对大规模代码提出一种增量式资源泄漏分析与检测方法，该方法以用户修改的方法为入口点，进行即时方法间资源泄露检测。传统的资源泄漏检测算法，在代码修改量远小于原始代码量的情况下，通常耗费大量的时间进行冗余检测分析，极大地降低了检测效率。本文提出的针对大规模项目代码的增量式资源泄

漏检测算法，是在进行资源泄漏检测的过程中，通过逐步缩小待分析资源相关方法的范围，并对未进行资源操作的相关方法进行“剪枝”，从而避免非资源相关方法的冗余分析，提高资源泄漏检测的效率和准确性。该方法支持过程间流敏感的资源泄漏检测，在用户编辑代码的过程中，从变更的函数入手，通过资源闭包求解、指向分析过滤等多种技术手段缩小资源泄漏检测的范围，进而实现了几十万行代码的即时缺陷分析与报告。与现有的工具比较分析发现，本文所提出的方法在保证准确率的前提下，90%的增量检测实验在 10s 内完成。实验结果表明，本文所提出的方法能够满足在用户编辑程序过程中即时对缺陷进行检测和报告的实际应用需求。

References:

[1] Wang KC, Wang TT, Su XH. Research on automatically located of software errors key scientific questions[J]. Chinese Journal of Computers, 2015, 38(11): 2262-2278.

[2] Wang T. Research and implementation of the source-oriented software vulnerability static detection technology[D]. Huazhong university of science and technology, 2015.

[3] http://www．javaperformancetuning.corn/news/newsl/16.shtml．

[4] Xiao Q, GongYZ, Yang CH. A path sensitive static defect detection method[J]. Journals of Software, 2010, 21(2): 209-217.

[5] Yang X, Gong YZ, Jin DH. A resource leakage detection method based on static analysis[C]. The third national software testing conference and mobile computing, 2009.

[6] Ayewah N, Pugh W, Morgenthaler J D, et al. Using findbugs on production software[C]//Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion. ACM, 2007: 805-806.

[7] Péter R. Rice HG. Classes of recursively enumerable sets and their decision problems. Transactions of the American Mathematical Society, vol. 74 pp. 358–366[J]. 1954.

[8] Zhao YS, Gong YZ, Liu L. Research on improving the efficiency and precision of the detection method of path sensitive defect[D]. 2011.

[9] Pioneers and Their Contributions to Software Engineering: Sd& M Conference on Software Pioneers, Bonn, June 28/29, 2001, Original Historic Contributions[M]. Springer Berlin Heidelberg, 2001.

[10] Torlak E, Chandra S. Effective interprocedural resource leak detection[C]//Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1. ACM, 2010: 535-544

[11] Nguyen L, Ali K, Livshits B, et al. Just-in-Time Static Analysis[J]. 2016.

[12] Vallée-Rai R, Co P, Gagnon E, et al. Soot-a Java bytecode optimization framework[C]//Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research. IBM Press, 1999: 13.

[13] Vallée-Rai R, Gagnon E, Hendren L, et al. Optimizing Java bytecode using the Soot framework: Is it feasible?[J]. CC, 2000, 1781: 18-34.

[14] Mei H, WangQX, Zhang L,Progress on software analysis technique[J]. Chinese Journal of Computers, 2009, 32(9): 1697-1710.

[15] Umanee N. Shimple: And Investigation of Static Single Assignment Form[D]. McGill University, 2005.

[16] Østvold B M, Kristoffersen T. Analysis of object-oriented programs: a survey[J]. 2005.

[17] Vallee-Rai R, Hendren L J. Jimple: Simplifying Java bytecode for analyses and transformations[J]. 1998.

[18] Liu Y, Wei L, Xu C, et al. DroidLeaks: Benchmarking Resource Leak Bugs for Android Applications[J]. 2016.

附中文参考文献:

[1] 王克朝, 王甜甜, 苏小红, 等. 软件错误自动定位关键科学问题及研究进展[J]. 计算机学报, 2015, 38(11): 2262-2278.

[2] 王涛. 面向源码的软件漏洞静态检测技术研究与实现[D]. 华中科技大学, 2015.

[4] 肖庆, 宫云战, 杨朝红, 等. 一种路径敏感的静态缺陷检测方法[J]. 软件学报, 2010, 21(2): 209-217.

[5] 杨绣, 宫云战, 金大海. 一种基于静态分析的资源泄漏检测方法[C]//第三届全国软件测试会议与移动计算, 栅格, 智能化高级论坛论文集. 2009.

[8] 赵云山, 宫云战, 刘莉, 等. 提高路径敏感缺陷检测方法的效率及精度研究[D]. 2011.

[14] 梅宏, 王千祥, 张路, 等. 软件分析技术进展[J]. 计算机学报, 2009, 32(9): 1697-1710.