软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2023,34(10):4724−4742 [doi: 10.13328/j.cnki.jos.006676]

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

RTDMiner: 基于数据挖掘的引用计数更新缺陷检测方法*

边 攀[1,2], 梁 彬[1], 黄建军[1], 游 伟[1], 石文昌[1], 张 健[3]

[1](中国人民大学 信息学院, 北京 100872)

[2](华为技术有限公司, 北京 100085)

[3](计算机科学国家重点实验室 (中国科学院 软件研究所), 北京 100190)

通信作者: 梁彬, E-mail: liangb@ruc.edu.cn

摘 要: 在 Linux 内核等大型底层系统中广泛采用引用计数来管理共享资源. 引用计数需要与引用资源的对象个数保持一致, 否则可能导致不恰当引用计数更新缺陷, 使得资源永远无法释放或者被提前释放. 为检测不恰当引用计数更新缺陷, 现有静态检测方法通常需要知道哪些函数增加引用计数, 哪些函数减少引用计数. 而手动获取这些关于引用计数的先验知识过于费时且可能有遗漏. 基于挖掘的缺陷检测方法虽然可以减少对先验知识的依赖, 但难以有效检测像不恰当引用计数更新缺陷这类路径敏感的缺陷. 为此, 提出一个将数据挖掘技术和静态分析技术深度融合的不恰当引用计数更新缺陷检测方法 RTDMiner. 首先, 根据引用计数的通用规律, 利用数据挖掘技术从大规模代码中自动识别增加或减少引用计数的函数. 然后, 采用路径敏感的静态分析方法检测增加了引用计数但没有减少引用计数的缺陷路径. 为了降低误报, 在检测阶段再次利用数据挖掘技术来识别例外模式. 在 Linux 内核上的实验结果表明, 所提方法能够以将近 90% 的准确率自动识别增加或减少引用计数的函数. 而且 RTDMiner 检测到的排行靠前的 50 个疑似缺陷中已经有 24 个被内核维护人员确认为真实缺陷.

关键词: 引用计数; 缺陷检测; 数据挖掘; 静态分析

中图法分类号: TP311

中文引用格式: 边攀, 梁彬, 黄建军, 游伟, 石文昌, 张健. RTDMiner: 基于数据挖掘的引用计数更新缺陷检测方法. 软件学报, 2023, 34(10): 4724–4742. http://www.jos.org.cn/1000-9825/6676.htm

英文引用格式: Bian P, Liang B, Huang JJ, You W, Shi WC, Zhang J. RTDMiner: Detecting Reference Count Update Bugs Based on Data Mining. Ruan Jian Xue Bao/Journal of Software, 2023, 34(10): 4724–4742 (in Chinese). http://www.jos.org.cn/1000-9825/6676.htm

RTDMiner: Detecting Reference Count Update Bugs Based on Data Mining

BIAN Pan[1,2], LIANG Bin[1], HUANG Jian-Jun[1], YOU Wei[1], SHI Wen-Chang[1], ZHANG Jian[3]

[1](School of Information, Renmin University of China, Beijing 100872, China)

[2](Huawei Technologies Co. Ltd., Beijing 100085, China)

[3](State Key Laboratory of Computer Science (Institute of Software, Chinese Academy of Sciences), Beijing 100190, China)

Abstract: Reference counts are widely employed in large-scale low-level systems including Linux kernel to manage shared resources, and should be consistent with the number of objects referring to resources. Otherwise, bugs of improper update of reference counts may be caused, and resources can never be released or will be released earlier. To detect improper updates of reference counts, available static detection methods have to know the functions which increase reference counts or decrease the counts. However, manually collecting prior knowledge of reference counts is too time-consuming and may be incomplete. Though mining-based methods can reduce the dependency on prior knowledge, it is difficult to effectively detect path-sensitive bugs containing improper updates of reference counts. To this end,

* 基金项目: 国家自然科学基金 (U1836209, 62132020, 61802413, 62002361)
收稿时间: 2021-06-11; 修改时间: 2021-12-18; 采用时间: 2022-03-14; jos 在线出版时间: 2023-04-04
CNKI 网络首发时间: 2023-04-06

this study proposes a method RTDMiner that deeply integrates data mining into static analysis to detect improper updates of reference counts. First, according to the general principles of reference counts, the data mining technique is leveraged to identify functions that raise or reduce reference counts. Then, a path-sensitive static analysis method is employed to detect defective paths that increase reference counts instead of decreasing the counts. To reduce false positives, the study adopts the data mining technique to identify exceptional patterns during detection. The experiment results on the Linux kernel demonstrate that the proposed method can automatically identify functions increasing or decreasing reference counts with the precision of nearly 90%. Moreover, 24 out of the top 50 suspicious bugs detected by RTDMiner have been confirmed to be real bugs by kernel maintainers.

Key words: reference count; bug detection; data mining; static analysis

引用计数法 (reference count) 是一种自动管理共享资源的技术[1], 在操作系统、数据库、浏览器、虚拟机等各类软件系统中有着广泛的应用[2−5]. 在引用计数法中, 会设置一个引用计数来跟踪资源的引用数. 当需要使用资源时, 如果资源还不存在, 就申请内存创建该资源, 并将引用计数初始化为 1; 否则如果资源已经存在, 就直接引用该资源, 并将资源的引用计数加 1. 当释放资源时, 引用计数减 1. 当资源的引用计数变为 0 时, 表明该对象将不再使用, 相应的内存将被释放或者重新分配给其他资源使用.

当资源的引用计数与实际引用它的对象个数不一致时, 将导致不恰当引用计数更新缺陷 (CWE-911: Improper update of reference count. https://cwe.mitre.org/data/definitions/911.html). 在本文中, 为方便叙述, 我们将不恰当引用计数更新缺陷简称为引用计数缺陷. 如果引用计数大于实际引用数时, 资源将永远无法释放, 造成资源耗尽, 导致拒绝服务漏洞, 如 CVE-2008-2136、CVE-2010-4593; 而如果引用计数小于实际引用数时, 资源可能被提前释放, 导致释放后使用漏洞, 如 CVE-2009-3624、CVE-2012-4787. 为避免缺陷, 增加引用计数之后通常会相应地减少引用计数.

为了保证原子性, 对引用计数的更新通常封装在函数中. 我们将初始化引用计数或增加引用计数的函数称为引用获取 (reference taking) 函数, 将减少资源引用计数的函数称为引用释放 (reference dropping) 函数. 用户代码通常通过调用引用获取函数和引用释放函数来管理引用计数, 而非直接修改引用计数的值[6]. 造成引用计数缺陷的原因通常是没有在有需要的地方调用引用获取函数或引用释放函数[7].

引用计数缺陷导致的后果可能并不会在当时就表现出来, 有可能在很长时间之后才会影响到其他组件甚至其他线程, 如资源耗尽或者系统崩溃. 因此, 纯动态分析技术 (如模糊测试[8]) 难以有效检测此类缺陷. 更自然的方法是利用静态分析方法来检测不恰当的引用计数更新缺陷. 例如, Mao 等人提出一个名为 RID 的静态分析方法, 通过检测函数中不同路径上对引用计数更新的不一致来发现不恰当引用计数更新缺陷[6].

然而, 传统静态分析方法面临的最大挑战是需要大量的先验知识[9]. 为检测引用计数缺陷, 静态分析方法通常需要知道哪些是引用获取函数、哪些是引用释放函数. 而引用获取/释放函数通常是特定于应用的, 不同的目标系统的引用获取/释放函数也往往不相同. 在大型系统中, 可能存在成百上千的引用获取/释放函数, 难以通过手动的方式来收集并配置到检查工具中. 此外, 在某些例外场景下, 并不需要获取引用或者释放引用. 例如, 在 Linux 内核中, 当 *inode* 节点通过 d_instantiate() 附加到目录项 dentry 后, 不应该再调用引用释放函数 iput() 来减少引用计数. 在实践中, 手动列举例外场景更是不现实.

为了缓解传统静态分析方法对先验知识的依赖问题, 人们提出代码挖掘方法[10], 利用数据挖掘技术从大规模代码中自动提取检查规则. 其基本原理是从大规模代码中提取数量占优的编码模式. 代码挖掘方法已经取得了较为显著的成果, 能够从实际大型系统中自动提取隐式编码规则并据此检测到数十个真实缺陷. 考虑到路径爆炸问题, 大部分代码挖掘方法都是路径不敏感的[10−12], 难以有效检测诸如引用计数缺陷这样与路径密切相关的缺陷.

本文提出一种新型引用计数缺陷检测方法 RTDMiner, 有机结合数据挖掘和静态分析技术来解决各自面临的挑战. 具体而言, 首先根据引用计数的一般使用规则, 利用数据挖掘技术自动识别引用获取函数和引用释放函数; 其次, 分析引用获取函数和引用释放函数未配对出现的路径, 进一步利用数据挖掘技术, 提取例外模式; 最后, 检测调用引用获取函数但既没有调用引用释放函数又不包含例外模式的路径来发现潜在引用计数缺陷. 此外, 我们还根据是真实缺陷的可能性对疑似缺陷进行排序, 以尽可能凸显真实缺陷.

为了评估本方法的有效性, 我们实现了一个原型系统, 并将其应用于 Linux 内核 (v5.11) 的缺陷检测中. 实验结果显示, 在只知道一些基本增减操作的情况下, RTDMiner 从 Linux 内核中自动识别出 1 250 对候选引用获取函

数和引用释放函数, 经人工确认, 其中 1 118 对都是真实的引用获取函数和引用释放函数, 准确率高达 89.4%. 利用这些引用获取函数和引用释放函数, RTDMiner 从 Linux 内核中检测到 3 140 个可能存在引用计数泄漏缺陷的函数, 即调用引用获取函数之后未调用相应的引用释放函数. 通过人工审计, 我们在排名前 50 个函数中发现 35 条疑似缺陷. 我们将这些疑似缺陷提交给 Linux 内核社区, 其中 24 条已经被 Linux 内核开发人员确认为真实缺陷, 其他 11 条疑似缺陷正在确认中. 目前为止, 我们提交的这些疑似缺陷还没有被证实是误报的. 实验结果表明, RTDMiner 可以在无需手动指定引用获取函数和引用释放函数的情况下检查出相当数量的不恰当的引用计数更新缺陷.

本文主要贡献如下.

(1) 根据引用计数的一般使用规律, 提出一种利用数据挖掘技术来自动识别引用获取函数和引用释放函数的方法.

(2) 提出一种利用数据挖掘技术来自动提取例外模式的方法. 在检测引用计数泄露缺陷时可以根据例外模式减少误报和凸显最可能的缺陷.

(3) 实现了一个不恰当引用计数更新缺陷检测原型系统. 利用该原型系统从大型软件系统中检测到数十个真实缺陷.

本文首先在第 1 节介绍相关工作. 然后, 在第 2 节以 Linux 内核中的一个实际缺陷为例来说明 RTDMiner 的研究动机和基本思想. 第 3 节介绍自动识别引用获取/释放函数的方法. 而第 4 节则说明如何根据挖掘到的引用获取/释放函数来检测引用计数更新缺陷. 第 5 节进行实验评估本方法的有效性. 第 6 节探讨未来可能的研究方向. 第 7 节对本文进行总结.

1 相关工作

1.1 引用计数缺陷检测

与本工作相关性比较大的工作是 Mao 等人提出的方法 RID[6], 通过检测路径之间的不一致性来发现潜在引用计数相关的缺陷. RID 将不一致的路径配对 (inconsistent path pair) 作为潜在缺陷. 对于同一个函数中无法从外部通过检查参数或返回值加以区分的路径, 如果它们对引用计数的更新不一致, 例如其中一条路径减少了引用计数, 但另外一条路径没有减少引用计数, 那么其中一条路径可能存在缺陷. 利用该方法, RID 从 Linux 内核等系统中检测到数十个不恰当引用计数更新缺陷.

Lal 等人提出一个利用静态分析技术检测闭包程序中的引用计数缺陷的方法[13]. 在程序入口 (如函数 main()), 所有的引用计数都被初始化为 0. 而在程序退出时, 如果引用计数不为 0, 说明程序存在引用计数泄漏缺陷. MalCom 提出的 CPyChecker[14]和 Li 等人提出的 Pungi[15]主要通过检查函数中引用计数的增加与逃逸个数不一致来发现潜在缺陷. 引用计数可以通过返回值、特定 API 等逃逸到函数之外.

上述引用计数缺陷检测方法都依赖大量的先验知识, 需要用户指定增加/减少引用计数的函数[6,14,15], 或者表示引用计数的结构成员[13]. 而在实际系统中, 特别是那些大型软件系统, 通过手动方式从成千上万的函数或结构成员中辨别这些先验知识费时费力且容易遗漏. 与上述方法相比, RTDMiner 可以根据更容易获取的递增/递减操作来自动识别引用计数更新函数, 并在此基础上检测可能的引用计数泄漏缺陷. 当然, RTDMiner 识别出的引用计数更新函数还可以作为上述方法的输入来检测其他形式的引用计数缺陷.

1.2 敏感函数识别

人们也提出来一些方法来识别具有特定语义的函数. 其中一些方法利用安全编程实践统计特征来启发式推导函数的语义. 例如, 为避免释放后使用缺陷, 指针传递给内存释放函数之后通常不会再引用. 基于这一经验性观察, Engler 等人统计在调用函数之后, 其参数的使用频率, 如果某个参数极少再被使用, 则该函数可能是内存释放函数[9]. Kremenek 等人进一步提出了利用因子图来整合更多信息的方法, 能够更准确地识别资源分配和释放函数[16]. 还有一些方法根据特定领域的专业知识来识别相关的敏感函数. Ganapathy 等人使用概念分析 (concept

analysis) 根据给定领域的知识 (如一个或多个关键数据结构) 来推导安全敏感操作的指纹[17]. Tan 等人提出了一个名为 AutoISES 的方法来推断应该受某些安全检查函数保护的操作[18].

启发式方法在查找某些类型的安全敏感函数方面很有效, 但同时也存在可伸缩性问题, 因为不同类型的安全敏感函数通常具有不同的关键特征, 需要具有一定的洞察力. 启发式方法通常根据一些容易获取的线索来推断想要的敏感函数. RTDMiner 也采用了类似的思路, 从相对比较容易获取 (甚至通用) 的递增/递减操作出发, 综合采用数据挖掘和程序分析技术来识别引用计数获取/释放函数.

Rasthofer 等人提出名为 SuSi 的方法, 利用分类技术来识别 Android 框架中的污点源 (taint source) 和污点汇 (taint sink)[19]. SuSi 使用数百个已标记的 API 来训练 SVM 分类器, 然后使用 SVM 分类器来预测其他 API 的标签. SuSi 已成功识别上千个 Android 框架中的污点源和污点汇, 用于 Android 应用程序上污点分析[20]. 但 SuSi 需要一定数量的训练样本才能训练出有效的分类器, 而在实际应用中往往难以获取足够多的样本. 为此, Bian 等人提出一个两阶段方法 SinkFinder[21], 首先利用词嵌入技术的类比分析能力, 可以仅根据一对种子函数, 找到与之具有类似语义的函数对, 获取足够多的样本. 然后, 利用这些样本训练一个分类器. SinkFinder 可以在已知样本数量有限的情况下也可以进行工作. SuSi 和 SinkFinder 是通用方法, 聚焦于识别与已知训练样本在行为模式上相似的函数. 而在引用计数缺陷检测中, 需要准确知道引用获取/释放函数. 而 SuSi 和 SinkFinder 难以精确到这些信息. 例如, 对于 SinkFinder 而言, 引用获取/释放函数对 (如 ext4_fc_start_update()/ext4_fc_stop_update()) 在使用模式上可能与加锁/解锁函数 (如 spin_lock()/spin_unlock()) 类似. 虽然难以直接识别引用计数更新函数, SinkFinder 可以与 RTDMiner 相结合, SinkFinder 可以根据通用递增/递减操作 (即“++”和“– –”) 来发现更多的递增/递减函数, 为 RTDMiner 提供更多线索来发现尽可能多的引用计数相关的函数.

1.3 编码模式挖掘

传统静态分析技术根据已知编码规则来检查缺陷. 而某些特定于应用的隐式编码规则往往难以获取. 为了缓解传统静态分析技术对先验知识的依赖, 人们提出利用数据挖掘技术从大规模代码中自动提取频繁出现的编码模式作为隐式编码规则, 并据此检测可能的缺陷[9−12,16,18,22−27]. RTDMiner 也利用了数据挖掘的思想来识别频繁函数对, 以限定引用计数更新函数的搜索范围.

引用计数获取/释放函数对是频繁函数对的子集. 我们自然想知道 RTDMiner 检测到的缺陷是否也是基于挖掘的检测方法 (如文献 [27]) 的子集? 考虑到性能和准确性, 绝大部分纯挖掘的缺陷检测方法都是路径不敏感的. 实际上, 频繁出现在同样上下文的函数在某些路径下可能并不会同时出现. 例如, kmalloc() 和 kfree() 是频繁函数对, 经常出现在同样的上下文. 但在 kmalloc() 分配内存失败, 或者将内存加入到全局链表的路径上, 往往并不会调用 kfree(). 因此, 如果规则挖掘阶段采用路径敏感的算法可能导致无法有效获取编码规则, 而如果在检测阶段进行路径敏感的检测又可能导致大量的误报. 为了解决这一问题, 研究人员通常针对特定类型的缺陷 (如返回值未验证) 提出有针对性的挖掘和检测方法. 为检测引用计数缺陷, 我们设计了 RTDMiner. 首先, 根据引用计数更新函数的一般规律, 利用数据挖掘算法识别引用获取/释放函数; 然后, 利用路径敏感的程序分析方法, 检测在需要释放引用计数的路径上未正确调用引用释放函数的路径, 从而发现纯挖掘的检测方法难以发现的问题.

实际上, 之前的一些工作也探索过将挖掘和传统分析工具相结合的方法. Pradel 等人从动态执行路径中挖掘 Java API 的使用规范, 然后利用静态分析技术来检查规范的违反[28]. 而 Sun 等人则先利用挖掘技术从代码中提取前置规则、后置规则及调用序列规则, 然后将这些规则传递给 Klocwork, 利用静态分析工具的强大检测能力来检测潜在缺陷[29]. 与上述方法只在规则提取阶段使用数据挖掘技术相比, RTDMiner 在检测阶段也需要用到挖掘技术来排除例外场景.

2 问题分析及解决思路

我们以 RTDMiner 从 Linux 内核中发现的一个真实缺陷来说明本文的研究动机. 如图 1 所示, Linux-v5.11 中函数 ext4_setattr() 中存在引用计数泄漏缺陷. 具体而言, 在其中一条路径上, 索引节点 *inode* 的引用计数递增之后却没有递减 (第 13 行). 对象 *inode* 的引用计数 i_fc_updates 用来记录正在执行的快速提交 (fast commit)[30]的个数,

在更新 *inode* 的内容和状态之前需要增加引用计数 i_fc_updates 的值, 而完成更新之后则应当减少引用计数. 如图 2 所示, 函数 ext4_fc_start_update() 会增加该引用计数, 而函数 ext4_fc_stop_update() 则减少引用计数, 并在引用计数的值降为 0 时唤醒其他被阻塞的快速提交. 造成图 1 所示缺陷的根本原因是函数 ext4_setattr() 在执行更新操作之前调用了函数 ext4_fc_start_update() 来增加引用计数, 但当函数 ext4_mark_inode_dirty() 将 *inode* 标记为脏时出现错误却没有调用函数 ext4_fc_stop_update() 来结束更新操作. 此时将导致 *inode* 的引用计数 i_fc_updates 泄漏, 造成 *inode* 上的其他操作可能一直被阻塞, 永远得不到执行.

```
// linux-v5.11/fs/ext4/inode.c
1  int ext4_setattr (struct dentry *dentry, struct iattr *attr)
2  {
3      struct inode *inode=d_inode (dentry);
4      int error, rc=0;
5      ...
6      ext4_fc_start_update (inode);
7      if ((ia_valid & ATTR_UID && !uid_eq (attr->ia_uid, inode->i_uid)) ||
8          (ia_valid & ATTR_GID && !gid_eq (attr->ia_gid, inode->i_gid))) {
9              ...
10             error=ext4_mark_inode_dirty (handle, inode);
11             ext4_journal_stop (handle);
12             if (unlikely (error))
                   // 缺少调用函数 ext4_fc_stop_update (inode), 导致引用计数泄露!
13                 return error;
14     }
15     ...
16 err_out:
17     if (error)
18         ext4_std_error (inode->i_sb, error);
19     if (!error)
20         error=rc;
21     ext4_fc_stop_update (inode);
22     return error;
23 }
```

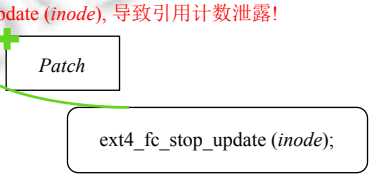

图 1 Linux-v5.11 中一条不恰当引用计数更新缺陷

为检测引用计数缺陷, 传统静态分析方法需要知道哪些函数会增加引用计数, 哪些函数会减少引用计数. 例如, 只有知道函数 ext4_fc_start_update() 和函数 ext4_fc_stop_update() 是一对引用获取函数和引用释放函数, RID[6] 才能检测图 1 所示缺陷. 然而, 在实际大型系统中, 可能存在成百上千更新引用计数的函数. 手动收集的方式费力耗时, 且容易出错和遗漏. Mao 等人[6]根据命名习惯来搜索可能的引用计数函数, 如“get-put”“inc-dec”等. 但在实际编码中, 命名方式可能多种多样. 根据函数名来获取引用计数函数可能会遗漏 ext4_fc_start_update() 和 ext4_fc_stop_update(). 因此, 在实践中, 可能需要对特定目标系统非常了解的专家配合进行反复试验, 才能构建出有针对性的检查工具.

为检测缺陷, 代码挖掘方法[10−12]首先提取形如 $A \Rightarrow B$ 的隐式编码规则, 其中 A 和 B 是编码元素 (如函数调用、条件检查等) 的集合. 规则 $A \Rightarrow B$ 表示包含 A 的上下文同时也应当包含 B. 违反隐式编码规则的代码则可能存在缺陷. 例如, 在 Linux 内核中, 在调用 ext4_fc_start_update() 的 7 个函数中都随后调用了 ext4_fc_stop_update(), 挖掘算法可以提取出隐式编码规则 R: {ext4_fc_start_update()} ⇒ {ext4_fc_stop_update()}. 但由于图 1 中的函数 ext4_setattr() 既调用了 ext4_fc_start_update() 也调用了 ext4_fc_stop_update(), 挖掘算法将认为函数 ext4_setattr() 遵守编码规则 R. 因此, 基于挖掘的方法难以有效检测图 1 所示缺陷.

为解决上述问题, 本文提出了一种将代码挖掘技术与传统静态分析技术有机相结合的方法 RTDMiner, 来检测引用计数缺陷. RTDMiner 的框架如图 3 所示, 其基本思想是通过两次挖掘来提取缺陷模式. 首先, 根据引用计数的一般使用规律, 利用数据挖掘技术从源代码中自动识别引用获取函数和引用释放函数. 一般地, 增加引用和减少引用要成对出现, 并且减少引用之后, 当前上下文通常不应该再使用该对象. 然后, 再次利用数据挖掘技术来提取调用引用获取函数但不需要调用相应的引用释放函数的例外模式. 最后, 根据引用获取/释放函数和例外模式来检测潜在缺陷, 并对检测到的疑似缺陷进行评分和排序, 以凸显真实缺陷.

```
// linux-v5.11/fs/ext4/inode.c
/*
 * Inform Ext4's fast about start of an inode update
 *
 * This function is called by the high level call VFS callbacks before
 * performing any inode update. This function blocks if there's an ongoing
 * fast commit on the inode in question.
 */
void ext4_fc_start_update (struct inode *inode)
{
        struct ext4_inode_info *ei=EXT4_I (inode);
        …
        atomic_inc (&ei->i_fc_updates);   // 增加引用计数i_fc_updates的值
        spin_unlock (&EXT4_SB (inode->i_sb)->s_fc_lock);
}

/*
 * Stop inode update and wake up waiting fast commits if any.
 */
void ext4_fc_stop_update (struct inode *inode)
{
        struct ext4_inode_info *ei=EXT4_I (inode);
        …
        if (atomic_dec_and_test (&ei->i_fc_updates)) // 减少引用计数i_fc_updates的值, 并检查结果是否为 0
            wake_up_all (&ei->i_fc_wait);
}
```

图 2　Linux-v5.11 中的引用获取和引用释放函数

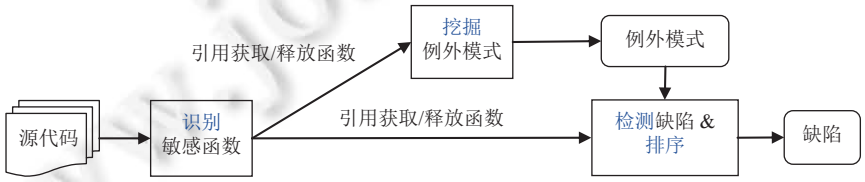

图 3　RTDMiner 框架图

例如, 配对出现的函数 ext4_fc_start_update() 和 ext4_fc_stop_update() 分别增加和减少了对象 *inode* 的成员变量 i_fc_updates. 而在 ext4_fc_stop_update() 之后通常不再使用 *inode*. 因此, 函数 ext4_fc_start_update() 和 ext4_fc_stop_update() 被当作一对引用获取/释放函数. 通过模式挖掘发现, 调用 ext4_fc_start_update() 之后的所有路径都应该调用 ext4_fc_stop_update(). 而函数 ext4_setattr() 中有一条路径违反了上述规则, 因此存在潜在缺陷. 可见, 将代码挖掘和静态分析相结合可以有效检测图 1 所示引用计数缺陷.

3　识别引用获取函数和引用释放函数

3.1　方法概述

如图 4 所示, 根据引用计数的一般使用规律, 利用静态分析技术和数据挖掘技术自动识别引用获取函数和引用释放函数. 我们发现引用计数在实际使用中一般遵守如下规律.

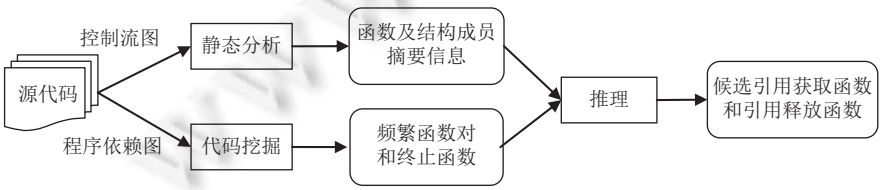

图 4　RTDMiner 识别引用获取函数和引用释放函数的框架图

(1) 资源的引用计数通常用表示资源的结构体的一个成员变量来表示, 对引用计数的值的更新也往往封装在引用获取函数和引用释放函数中.

(2) 在引用获取函数中, 如果新创建一个对象, 则将资源的引用计数初始化为 1; 否则, 如果是已经存在的对象, 则将资源的引用计数加 1.

(3) 在引用释放函数中, 资源的引用计数将被减 1. 有些引用释放函数还会检查引用计数的值, 并在取值降为 0 时释放资源.

(4) 当引用被释放后, 相应的资源在当前上下文通常处于不可用状态, 可能随时被修改甚至释放. 因此, 为了避免非预期错误, 在引用释放函数被调用之后通常不再使用资源.

(5) 为了保证引用计数的一致性, 引用获取函数和引用释放函数通常成对调用.

基于上述规律, 识别引用获取函数和引用释放函数的流程如下.

首先, 利用自底向上的程序归纳技术计算函数及数据结构的摘要信息 (见第 3.2 节). 摘要信息记录了对成员变量的修改情况, 如初始化、递增、递减或其他修改情况.

其次, 利用数据挖掘技术从源代码中挖掘频繁函数对, 并识别参数很少再次引用的终止函数 (见第 3.3 节).

最后, 根据规律 (1)–(3), 只有那些对参数的某个成员变量进行递增或递减的函数才可能是引用获取或释放函数; 而根据规律 (4) 和 (5), 可以进一步筛选引用获取函数和引用释放函数 (见第 3.4 节).

3.2 计算函数和数据结构摘要

我们计算在整个系统和特定函数中对结构成员的变更情况. 根据结构成员的变更摘要信息, 可以识别那些可能用来表示引用计数的结构成员, 并进一步筛选出可能的引用获取函数及引用释放函数.

根据引用计数的使用规律, 通常在分配资源时将引用计数的值初始化为 1, 增加对资源的引用时将引用计数的值递增, 而释放对资源的引用时则将引用计数的值递减. 初始化为 1 可以视为特殊的递增, 即将变量值隐式地从 0 变成 1. 因此, 在本文中, 我们主要关注 3 种变量变更类型, 即递增 (increment, *I*)、递减 (decrement, *D*) 和其他 (other, *O*), 分别 *I*、*D* 和 *O* 来表示. 除了像“*v*++”“*v*– –”“*v* += 1”“*v* –= 1”等通用递增递减操作, 目标系统可能还有一些系统特定的递增递减操作, 例如 Linux 内核中的递增函数 atomic_inc(*v*) 和递减函数 atomic_dec(*v*). 相对于引用获取和引用释放函数, 这些递增递减操作较为容易获取且通常在很长时间保持不变. 因此, 在本文中我们将递增和递减操作作为线索来发现引用获取/释放函数.

我们用集合 $\mathbb{T}$ 来记录所有结构成员的变更情况. 其中 $\mathbb{T}(s,m)$ 记录了程序中对结构体 *s* 的成员 *m* 的所有可能的更新情况, 而 $\mathbb{T}(f,i,m)$ 记录了函数 *f* 的第 *i* 个参数的成员 *m* 的更新情况. 在本文中, 为便于说明, 我们将函数返回值也当成函数的一个参数. 当 *i* = 0 时, 表示函数 *f* 的返回值, 当 *i* 大于 0 时则表示函数 *f* 的第 *i* 个参数.

如算法 1 所示, 我们采用自底向上 (bottom-up) 的路径敏感的过程间分析方法来计算结构成员的更新情况. 算法 1 的输入是函数调用图 $\mathbb{G}$. 算法 1 首先根据函数间的调用关系对函数进行排序, 使得被调用函数先于调用函数被分析 (第 3–4 行).

算法 1. 计算数据结构及函数摘要.

```
1 procedure calculate_summary(𝔾)
2     𝕋 = {};
3     𝔽 = topological_sort(𝔾);
4     for (each function f in F)
5         // 计算当前函数内成员变量的更新类型
6         for (each statement stmt in f)
7             if (stmt updates member m of variable v with type t)
8                 s = structure_of(v); // 变量 v 的类型
```

```
9            𝕋(s,m) = 𝕋(s,m)∪{t}; // 更新结构体摘要
10         end if
11      end for
12      // 计算每条路径上对参数的成员变量的更新类型
13         for (each path p of f)
14            for (each parameter i and member m)
15               t = calculate_update_type(𝕋,p,i,m );
16               if (t != O)
17                  𝕋(f,i,m) = 𝕋(f,i,m)∪{t}; // 更新函数摘要
18               end if
19            end for
20         end for
21      end for
22   return 𝕋; // 返回摘要信息
23 end procedure
24
25 procedure calculate_update_type(𝕋,p,i,m )
26   numInc = 0; // 递增计数, 记录对结构成员 m 进行递增的次数
27   numDec = 0; // 递减计数, 记录对结构成员 m 进行递减的次数
28   for (each statement stmt on path p)
29      t = O;
30      if (stmt calls function g)
31         𝕋 = 𝕋(g, j,m); // 函数 f 的参数 i 对应函数 g 的参数 j
32         if (𝕋 == {I} OR 𝕋 == {D})
33            t = 𝕋.value(0); // 获取在调用函数中的更新类型
34         end if
35      else if (stmt updates member m of parameter i with type t1)
36            t = t1;
37      end if
38      numInc += (t == I); // 如果进行了类型为 I 的更新, 则增加递增计数
39      numDec += (t == D); // 如果进行了类型为 D 的更新, 则增加递减计数
40   end for
41   if (numInc = numDec + 1)
42      return I; // 在该路径上, 结构成员 m 的值递增了
43   else if (numInc + 1 = numDec)
44         return D; // 在该路径上, 结构成员 m 的值递减了
45   end if
46   return O; // 在该路径上, 结构成员 m 的值的变更不是递增也不是递减
47 end procedure
```

对于每个函数 f, 首先遍历函数中所有语句, 如果某条语句更新了结构体 s 的成员变量 m 的值, 假设更新类型为 t, 则将 t 加入 m 的变更集合 $\mathbb{T}(s,m)$ 中 (第 6–11 行). 然后, 遍历函数 f 的路径 (第 13 行), 计算在该路径上对参数

i 的成员变量 m 的更新类型 t (第 14–15 行). 如果是递增或递减操作, 则将 t 加入到集合 $\mathbb{T}(f,i,m)$ 中, 而其他更新类型则不进行处理 (第 16–18 行). 最后, 算法返回数据结构及函数的摘要集 $\mathbb{T}$ (第 22 行).

我们用子程序 caculate_update_type() 来计算成员变量在某条路径上的更新类型 (第 25–47 行). 递增计数和递减计数分别记录了路径上对成员变量进行递增和递减的语句数. 特别是对于函数调用语句, 并没有进入被调用函数进行分析, 而是根据它的摘要信息来确定结构成员的更新情况 (第 30–34 行), 这也是我们将该计算方法称为自底向上的过程间分析方法的由来. 如果递增语句数比递减语句数多一条, 说明在该路径上成员变量的值整体上递增了 (第 41–42 行); 类似的, 如果递增语句数比递减语句数少一条, 说明在该路径上成员变量的值整体上递减了 (第 43–44 行). 需要说明的是, 引用获取函数和引用释放函数的实现通常不会过于复杂, 为了避免路径爆炸问题和提高分析效率, 我们只关心分支数比较少 (默认小于 10) 的函数, 并且对于循环体只展开一次. 此外, 为了提高分析精度, 我们还考虑了变量间的别名关系.

例如, 图 2 所示代码中, 函数 ext4_fc_start_update() 中调用 atomic_inc() 递增了 ei 的成员变量 i_fc_updates 的值. 而 ei 与参数 *inode* 是别名 (见第 10 行). 类似的, 函数 ext4_fc_stop_update() 减少了 ei 和 *inode* 的成员变量 i_fc_updates 的值. 因此, 从函数 ext4_fc_start_update() 和 ext4_fc_stop_update() 计算得到的摘要信息如下:

$$\mathbb{T}(\text{ext4\_inode\_info}, \text{i\_fc\_updates}) = \{I, D\},$$
$$\mathbb{T}(inode, \text{i\_fc\_updates}) = \{I, D\},$$
$$\mathbb{T}(\text{ext4\_fc\_start\_update}, 1, \text{i\_fc\_updates}) = \{I\},$$
$$\mathbb{T}(\text{ext4\_fc\_stop\_update}, 1, \text{i\_fc\_updates}) = \{D\}.$$

3.3 挖掘频繁函数对和终止函数

根据引用计数的使用规律, 引用获取函数和引用释放函数常成对出现, 而且释放对资源的引用后通常不再使用该资源. 因此, 我们利用数据挖掘技术从大规模代码中提取频繁函数对和终止函数, 作为候选引用获取函数或引用释放函数.

引用获取函数和引用释放函数之间通常具有一定的语义关系, 在数据流上表现为具有数据依赖 (DataDep) 关系或数据共享 (DataShare) 关系. 在某个函数定义中, 如果函数 f 的输出值 (如返回值) 是函数 g 的参数, 则称函数 g 数据依赖于函数 f. 如果函数 f 和 g 都采用相同的变量值作为它们的参数, 并且在控制流图中它们之间存在至少一条可行路径, 则它们具有数据共享关系[22]. 如果路径上的所有条件都是可满足的, 则路径是可行的[31]. 用元组 <$f(i)$, $g(j)$> 来表示具有强语义关系的函数 f 和 g, 其中 i 和 j 是具有数据关联关系的参数.

以图 1 所示代码为例, 在函数 ext4_setattr() 中, 函数 ext4_fc_start_update() 和 ext4_fc_stop_update() 的参数相同, 都是变量 *inode*, 且取值都是函数 d_inode() 的返回值. 因此, 在函数 ext4_setattr() 中, 函数 ext4_fc_start_update() 和 ext4_fc_stop_update() 之间存在数据共享关系, 并且都数据依赖于函数 d_inode().

如果函数 f 调用了 g_1 和 g_2, 且 g_1 和 g_2 之间存在数据依赖或数据共享关系, 具有关联的参数分别为 i_1 和 i_2 则称 f 包含函数对 p: <$g_1(i_1)$, $g_2(i_2)$>. 包含 p 的函数定义的个数称为 p 的支持度, 记为 *support*(p). 如果 p 的支持度不小于给定的最小支持度阈值 *min_support*, 则认为 p 是频繁函数对. 例如, 在 Linux-v5.11 中, 有 7 个函数定义包含函数对 p_1: <ext4_fc_start_update(1), ext4_fc_stop_update(1)>. 在最小置信度阈值 *min_support* = 3 的情况下, p_1 是频繁函数对.

终止函数是那些调用之后其参数不再使用的函数, 引用释放函数通常也是终止函数. 我们也采用挖掘的方法来识别终止函数. 如果函数 f 中调用 g 之后, g 的参数 i 不再被使用, 则称在函数 f 中, 函数 g 是一个终止函数, 终止了对第 i 个参数的引用. 假设函数 g 在 n 个函数中是终止函数, 则 g 是终止函数的置信度为:

$$confidence(g) = \frac{n}{support(g)}.$$

如果其置信度不小于给定的最小置信度阈值 *min_confidence*, 则 g 对于参数 i 是一个终止函数, 记为 $g(i)$. 例如, 在 Linux 内核中, 函数 ext4_fc_stop_update() 在 7 个函数中被调用, 且其中 6 个函数中, 它的第 1 个参数不再使用. 因此, 对于第 1 个参数, 函数 ext4_fc_stop_update() 是终止函数的置信度为 0.857. 在最小置信度阈值为 0.8 的情况下, 函数 ext4_fc_stop_update(1) 将被当作终止函数.

3.4 推断引用获取函数和引用释放函数

我们根据引用计数的使用规律, 综合利用摘要信息和挖掘结果来识别可能的引用获取函数和引用释放函数. 为便于维护引用计数的一致性, 引用获取函数和引用释放函数经常在同一个层次被调用. 换言之, 引用获取函数和引用释放函数往往成对出现. 此外, 引用释放函数在解除对资源的占用后, 该资源随时可能会被释放或者分配给其他对象, 它在当前上下文的状态将不可预期. 为避免安全隐患, 该资源通常不会再被使用. 因此, 我们可以以频繁函数对和终止函数来限定搜索范围.

引用计数的变化通常是递增或递减, 极少出现跳跃式的增加或减少. 因此, 如果程序中存在类型为其他类型的更新操作修改了结构体 s 的成员变量 m, 即 $O \in \mathbb{T}(s,m)$, 则 m 不可能是引用计数. 而在引用获取函数或引用释放函数中引用计数的变更类型也只能有一种, 即递增或递减.

基于上述洞察, 我们利用算法 2 来识别可能的引用获取函数和引用释放函数. 算法 2 的输入参数有成员变量更新摘要 $\mathbb{T}$、频繁函数对集合 $\mathbb{P}$、终止函数集合 $\mathbb{E}$.

算法 2. 推断引用获取函数及引用释放函数.

```
1 procedure infer_reference_count_functions(𝕋,ℙ,𝔼)
2   RTSet = {};
3   RDSet = {};
4   RTDSet = {};
5   for (each frequent function pair p:<f(i), g(j)> in ℙ) // 以频繁函数对限定搜索范围
6     if (g(j) ∈ 𝔼) // 根据终止函数进一步限定搜索范围
7       continue;
8     end if
9     s = structure_of(i); // 参数 i 的类型
10    if ( ∃m → (O ∉ 𝕋(s,m) AND 𝕋(f,i,m) == {I} AND 𝕋(g,j,m) == {D}) ) // 筛选递增递减函数
11      RTSet = RTSet ∪ {f(i)}; // 引用获取函数
12      RDSet = RDSet ∪ {g(i)}; // 引用释放函数
13      RTDSet = RTDSet ∪ {p}; // 引用获取-释放函数对
14    end if
15  end for
16  return RTSet, RDSet, RTDSet;
17 end procedure
```

首先, 对于频繁函数对 p: <$f(i)$, $g(j)$> (第 5 行), 如果函数 g 不是终止函数, 则 g 也不太可能是引用释放函数 (第 6–8 行). 其次, 如果存在成员变量 m, 它在整个工程中的变更类型只有递增和递减, 并且在函数 f 只对它进行了递增, 而函数 g 中只对它进行了递减, 则函数 f 和 g 可能是引用获取函数和引用释放函数 (第 10–14 行). 算法 2 最后返回所有可能的引用获取函数、引用释放函数集、引用获取-释放函数对 (第 16 行).

例如, 对于频繁函数对 p_1: <ext4_fc_start_update(1), ext4_fc_stop_update(1)>, 函数 ext4_fc_stop_update() 是终止函数. 在整个 Linux 内核中对成员变量 i_fc_updates 的更新类型只有递增和递减, 且函数 ext4_fc_start_update() 中只有对 i_fc_updates 进行递增的路径, 而函数 ext4_fc_stop_update() 中只有递减路径. 因此, 函数 ext4_fc_start_update() 是引用获取函数, 而函数 ext4_fc_stop_update() 则是引用释放函数.

4 检测引用计数缺陷

得到引用获取函数和引用释放函数之后, 我们采用路径敏感的过程内分析方法来检测引用计数缺陷[32]. 引用

计数缺陷大致可以分为 2 种: (1) 在需要减少引用计数的地方没有调用引用释放函数; (2) 在需要增加引用计数的地方没有调用引用获取函数. 何时需要增加或减少引用计数往往与具体的业务逻辑有关, 难以静态地确定具体的检查场景. 而调用引用获取函数来增加对象的引用计数天然地意味着随后可能需要调用引用释放函数来减少对象的引用计数. 考虑到上述原因, 本文主要检查缺少调用引用释放函数导致的引用计数泄漏缺陷.

一个较为直接的方法是将所有调用了引用获取函数但没有调用相应的引用释放函数的路径当作潜在缺陷. 但在某些场景下调用引用释放函数反而会导致释放后使用缺陷. 首先, 引用获取函数出现异常时, 例如返回错误码或者无效指针, 引用计数并未增加, 此时不需要减少引用计数. 其次, 如果对象被传递到当前上下文之外也不需要减少引用计数. 如果资源被直接返回、赋值给输出参数或全局变量、通过特定 API 加入全局工作队列等, 则可以在其他地方引用该资源或者在不需要时减少资源的引用计数. 这种情况被称为引用逃逸 (reference escaping). 我们将引用获取失败和引用逃逸统称为例外场景, 此时不需要调用引用释放函数. 而手动指定例外场景需要投入大量人力, 难以应用于实际缺陷检测中.

为解决上述问题, 我们首先采用代码挖掘技术来识别例外模式 (见第 4.1 节). 然后, 检查那些调用了引用获取函数但却既不包含例外模式又没有调用相应的应用释放函数的路径, 作为潜在缺陷 (见第 4.2 节).

4.1 挖掘例外模式

例外模式的挖掘方法主要基于如下洞察: 在同一个函数中, 调用引用获取函数 f 之后, 有些路径上调用了相应的引用释放函数 g, 而其他路径上没有调用函数 g, 那么那些没有调用函数 g 的路径上则可能包含例外模式, 而调用函数 g 的路径上则很有可能不包含例外模式. 如图 5 所示, 调用引用获取函数 *acquire*() 之后, 在路径 p_2 上调用了引用释放函数 *release*(), 但在另外一条路径 p_1 上没有调用 *release*(), 因此函数 *consumer*() 可能是例外模式, 而函数 *handle*() 则不太可能是例外模式.

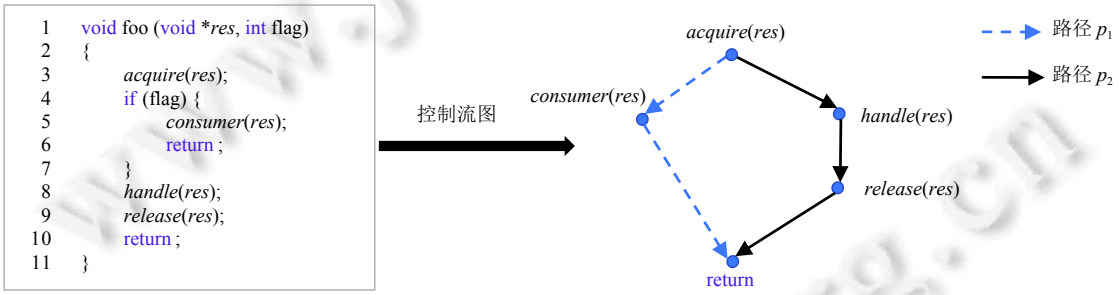

图 5 例外模式示例

根据上述分析, 例外模式的挖掘过程如下.

首先, 对于引用获取函数 f 的每个调用实例, 如果从它开始, 有些路径调用了引用释放函数而其他路径则没有调用引用释放函数, 我们就进行前向数据流分析, 找到所有与 f 具有数据依赖或数据共享关系的语句. 如果某条语句 s 出现在调用引用释放函数的路径上, 则加入常规集合 *RSet* (regular set) 中, 否则加入例外集合 *ESet* (exceptional set) 中. 在加入集合之前, 我们对语句 s 进行规范化处理. 对于函数调用语句, 用函数名来表示; 其他语句类型, 则根据变量的取值来源重命名变量, 消除由于变量命名上的差异带来的影响. 对于条件检查语句, 我们还会在条件表达式之前分别加上“true”或“false”来区分该路径上条件成立或不成立.

然后, 计算语句 s 是例外模式的置信度, 计算公式为:

$$confidence(s)=\frac{esupport(s)}{esupport(s)+rsupport(s)},$$

其中, $rsupport(s)$ 是语句 s 在常规集合中出现的次数, 成为常规支持度; 类似的, $esupport(s)$ 是语句 s 在例外集合中出现的次数. 例外置信度越高, 说明在包含语句 s 的路径上越有可能不需要调用引用释放函数. 如果语句 s 的置信度不小于最小置信度阈值 *min_confidence*, 则称语句 s 是引用获取函数 f 的例外模式. 当语句 s 出现时, 相应的路径

上不需要再调用与引用获取函数 f 相对应的引用释放函数.

4.2 检测缺陷

为检测潜在缺陷, 我们从引用获取函数 f 的调用点出发, 前向遍历每条潜在可执行路径. 如果不包含例外模式的路径上没有调用相应的引用释放函数, 则该路径可能存在引用计数泄漏缺陷. 为了避免不可行路径造成的误报, 我们利用约束求解技术对路径进行剪枝. 具体而言, 首先, 利用符号执行技术将变量符号化, 并收集路径上所有条件表达式 $\{c_1,c_2,\ldots,c_n\}$, 形成布尔表达式 $c_1 \wedge c_2 \wedge \ldots \wedge c_n$; 然后利用约束求解器 STP[33]对布尔表达式进行求解, 如果该表达式的值永远为假, 说明该路径不可行. 不可行路径在遍历过程中将被忽略.

与其他静态分析方法类似, RTDMiner 的检查结果也不可避免会有一些误报. 我们采用广泛使用的排名机制来缓解误报的影响[10−12]. 其基本原理是一条隐式规则的违反数越多, 规则约束力越小, 据此检测到的缺陷越可能是误报. 对于引用计数泄露缺陷而言, 与引用获取函数 f 相关的缺陷越多, 越可能是误报. 此外, 我们除了直接根据例外模式排除疑似缺陷, 还将缺陷路径上语句的例外置信度引入到评分中. 例外置信度越高, 缺陷评分越低. 对于引用获取函数 f, 缺陷路径 p 的评分计算公式如下:

$$score(f,p)=\min\{(1-econfidence(s))|s\in p\}\times\frac{N-n}{N},$$

其中, N 是 f 的总的调用次数, 而 n 是可能有缺陷的 f 的调用次数. 此外, 我们选择路径上例外置信度最大的语句 s 来参与评分.

最后, 按照疑似缺陷的评分从高到低进行排序, 排名越靠前越可能是真实缺陷.

需要注意的是, 为了避免路径爆炸问题, 与挖掘引用获取/释放函数相似, 每个循环体也只展开一次. 此外, 我们还设置了路径上限 (默认值为 1 000), 当检测的路径数超过上限时, 就停止对当前函数的分析, 继续分析下一个函数. 这种折中方法虽然可能会造成一些漏报, 但能够保证整个检测过程顺利结束.

5 实验评估

5.1 实验设置

为了评估 RTDMiner 的有效性, 我们实现了相应的原型系统, 并将其应用到 Linux 内核上的缺陷检测中. 在实验中, 我们主要通过回答以下两个研究问题来评估本方法的有效性.

RQ1: 是否能够有效识别引用获取函数和引用释放函数? (见第 5.3 节)

RQ2: 各项参数如何影响方法的有效性? (见第 5.4 节)

RQ3: 自动识别的引用获取/释放函数是否有助于检测未知缺陷? (见第 5.5、5.6 节)

Linux 内核已被广泛用作静态缺陷检查方法[6,34−37]和基于挖掘的缺陷检测方法[9−12,22−26]的评估目标 (TOE). 例如, Mao 等人提出的引用计数缺陷检查方法 RID 就主要以 Linux 内核为测试目标[6]. 我们选择 Linux 内核作为测试目标的主要原因是希望通过检测出一些真实的未知缺陷来证明本方法的有效性. Linux-v5.11 是我们实验时的最新内核版本, 包含约 2.5 万个 C 文件和 1.9 万个头文件, 约 1 500 万行代码和 37 万个函数.

RTDMiner 需要指定两个参数: (1) 用来挖掘频繁函数对的最小置支持度阈值 *min_support* 和 (2) 用来识别终止函数和例外模式的最小置信度阈值 *min_confidence*. 理论上, *min_support* 和 *min_confidence* 的值越大, 结果越准确可靠, 但得到的频繁函数对和终止函数越少. 我们还通过实验来评估支持度和置信度阈值对准确率的影响 (见第 5.4 节). 为了能够挖掘尽可能多的引用计数获取/释放函数以期检测尽可能多的潜在引用计数缺陷, 我们将 *min_support* 和 *min_confidence* 的默认值分别设置为 3 和 0.8. 在实际使用中, 用户可以根据实际情况进行相应调整, 确定合理的最小支持度阈值和最小置信度阈值.

本实验在 Ubuntu 18.04 系统上进行, 机器内存为 16 GB, 识别引用获取/释放函数和检测缺陷都采用单线程.

5.2 性能开销

RTDMiner 共包含 5 个环节: (1) 解析代码; (2) 分析语句间的依赖关系; (3) 挖掘频繁函数对和终止函数; (4) 识

别引用获取/释放函数; (5) 检查引用计数泄漏缺陷. 在 Linux-v5.11 上, RTDMiner 共耗时约 370 min. 其中大部分时间花在解析代码和分析语句间的依赖关系上, 共耗时约 284 min, 约占总体时间的 76.8%. 此外, 我们还监控了 RTDMiner 在整个运行周期的内存开销, 发现最多占用 8.7 GB 内存. 考虑到 Linux 内核的代码体量 (超千万), 时间和内存开销是可以接受的.

5.3 识别引用获取函数和引用释放函数

我们在 Linux 内核上进行实验来评估 RTDMiner 在自动识别引用计数更新函数方面的有效性. RTDMiner 根据结构成员的递增和递减情况来识别可能被作为引用计数的成员变量. 表 1 列举了本次实验中用到的更改变量值的操作及对应的操作类型. 除了“*v*++”“*v* += 1”“*v*– –”“*v* –= 1”等通用递增递减操作, 我们还配置了 Linux 内核所特有的一些修改变量值的操作, 如对参数值进行递增的函数 *atomic_inc*() 和对参数值进行递减的函数 *atomic_dec*(). 为了效率, 函数 *atomic_inc*() 和 *atomic_dec*() 采用汇编语言来实现, 这使得我们无法通过源代码分析从通用递增递减操作来推断它们的行为. 当然, 可能可以通过分析汇编代码, 自动识别可能的递增和递减函数. 但在我们的原型系统中引入对汇编代码的分析会带来额外的开发工作量, 而自动识别这类基本的递增和递减函数也不是本研究的重点. 此外, 考虑到在实际系统中, 类似于 *atomic_inc*() 和 *atomic_dec*() 的函数通常被设计为公共函数, 它们往往定义在特定的头文件中, 而且在很长时间都会保持不变. 例如, 在 Linux 内核中, 表 1 所列举的那些更新变量值的函数都定义在头文件 atomic_fallback.h 和 refcount.h 中, 比较容易获取. 因此, 在本实验中我们选择手动提取 Linux 内核中的基本递增和递减函数作为线索.

● 准确率. RTDMiner 从 Linux-v5.11 中共挖掘出 100 535 个频繁函数对. 从这些频繁函数对中识别出 1 250 对候选引用计数获取/释放函数对, 包含 1 167 个候选引用获取函数和 654 个候选引用释放函数. 我们对这些自动识别的引用获取函数、引用释放函数及函数对进行人工验证. 如果一个函数确实增加了某个参数的引用计数, 则将其标记为真实的引用计数获取函数, 记为“TrueRT”; 类似的, 如果一个函数减少了某个参数的引用计数, 则将其标记为真实的引用释放函数, 记为“TrueRD”. 如果 *f* 是引用获取函数且 *g* 是引用释放函数, 则函数对<*f*, *g*> 是引用获取-释放函数对, 标记为“TrueRTD”. 经人工确认, 1 167 个候选引用获取函数中有 1 044 个函数被标记为“TrueRT”, 准确率为 89.5%; 654 个候选释放函数中有 578 个被标记为“TrueRD”, 准确率约为 88.4%; 而 1 250 对候选引用计数获取/释放函数对中有 1 118 对被标记为“TrueRTD”, 准确率约为 89.4%. 从人工验证结果看, RTDMiner 可以较为准确地自动识别引用计数更新函数.

● 召回率. 一些检查工具已经能够支持检查不恰当的引用计数更新缺陷, 其中最具代表性的是 Coccinelle. Coccinelle 是一个开源静态分析工具, 用于匹配 SmPL (semantic patch language) 语言编写的缺陷模式, 还可以根据指定的修复模式为缺陷生成相应的补丁文件. Coccinelle 已经被 Linux 内核采纳作为日常开发工具之一 (https://www.kernel.org/doc/html/v4.16/dev-tools/coccinelle.html). Linux 内核开发者和维护者广泛使用 Coccinelle 来发现潜在代码缺陷. 在公开的的 Coccinelle 缺陷检查模式中, 有 7 条专门用来检查引用技术泄漏缺陷 (Coccinelle 公开的 Linux 内核缺陷模式列表. https://coccinelle.gitlabpages.inria.fr/website/impact_linux.html). 此外, Coccinelle 官网还公布了根据这些缺陷模式检测到的 25 个引用计数泄露缺陷. 这 7 条缺陷模式共涉及 14 个引用获取函数和 6 个引用释放函数. 实验结果表明, RTDMiner 成功识别出其中 10 个引用获取函数和所有的引用释放函数, 召回率约为 80%. 而另外 4 个引用释放函数未被识别的主要原因是出现频率太低. 例如, Coccinelle 规则集中配置的引用获取函数 of_find_all_nodes() 在 Linux-v5.11 中仅有两处调用, 因小于最小支持度而被忽略. 更重要的是, RTDMiner 自动识别出上千个通过人工总结的方式难以发现的引用获取/释放函数, 对人工归纳的引用计数缺陷知识是一个重要的补充.

我们还统计了符合 Mao 等人总结的命名规律[6]的引用获取/释放函数对. 结果如表 2 所示, 可见 RTDMiner 发现的 1 118 个引用获取/释放函数对中只有约 29% 用了“get-put”和“inc-dec”的命名方式. 因此, 根据命名方式来确定引用获取/释放函数可能会造成大量遗漏.

表 1 更新变量值的操作列表

适用	基本增减操作	类型
通用	v++, ++v, $v = v + 1$, $v \mathrel{+}= 1$, $v = 1$	I
	v– –, – –v, $v = v - 1$, $v \mathrel{-}= 1$, $v = 0$	D
	$v = t$, $v \mathrel{+}= t$, $v \mathrel{-}= t$, $v = v + t$, $v = v - t$ **where** t != 0 **and** t != 1	O
Linux内核特有	atomic.*_inc.*(v)	I
	atomic.*_dec.*(v)	D
	atomic.*_add.*(1, v)	I
	atomic.*_sub.*(1, v)	D
	atomic.*_add.*(t, v), atomic.*_sub.*(t, v) **where** t != 1	O
	refcount_inc.*(v)	I
	refcount_dec.*(v)	D
	refcount_add.*(1, v)	I
	refcount_sub.*(1, v)	D
	refcount_add.*(t, v), refcount_sub.*(t, v)_ **where** t != 1	O

表 2 不同命名习惯的引用获取-释放函数对个数

命名习惯	示例函数对	数量
get-put	<of_get_next_child, of_node_put>	318
inc-dec	<chcr_inc_wrcount, chcr_dec_wrcount>	1
其他	<affs_bread, affs_brelse> <f2fs_find_entry, f2fs_put_page> <ocfs2_start_trans, ocfs2_commit_trans> <qxl_bo_kmap_atomic_page, qxl_bo_kunmap_atomic_page>	799

此外, 对于每个引用获取函数, RTDMiner 会挖掘例外模式来减少误报. RTDMiner 从 Linux-v5.11 中共挖掘出 358 个例外模式. 我们随机抽检了其中 50 个例外模式, 经过人工确认, 其中 47 个例外模式被真实为真. 在包含这些模式的路径上, 确实不需要再调用相应的引用释放函数. 例外模式挖掘算法的准确率约为 94%.

5.4 参数敏感性

我们还在 Linux-v5.11 上进行了一系列实验来研究参数最小支持度 *min_support* 最小置信度 *min_confidence* 对引用获取/释放函数的识别的影响.

为评估最小支持度 *min_support* 的影响, 我们进行了 11 个实验. 在实验中, 我们固定了最小置信度的值 (0.8), 变化 *min_support* 的值 (从 3 到 50). 图 6 展示了在不同的最小支持度设置下识别的引用获取-释放函数对个数及准确率. 从实验结果可见, 随着最小支持度阈值的增加, 函数对数量下降较为明显, 而准确率却没那么敏感, 围绕 0.85 波动. 通常, 支持度阈值越小, 能够识别的引用获取和释放函数越多. 但如果取值过小 (如小于 3), 则可能将一些偶然出现在同一上下文的函数识别成频繁函数对, 其结果是不可靠的. 因此, 为了识别尽可能多的引用获取和释放函数, 我们将最小支持度阈值的默认值设置为 3.

类似地, 我们进行了 8 个实验来评估最小置信度 *min_confidence* 的影响. 结果如图 7 所示, 随着最小置信度 *min_confidence* 的增加 (取值从 0.65 到 1.0), 识别出的引用获取-释放函数对数量在下降但准确率在上升. 为了平衡准确率和数量, 我们将最小置信度阈值的默认值设置为 0.8.

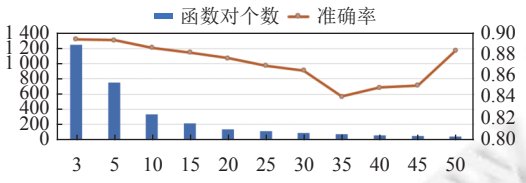

图 6 不同最小支持度 *min_support* 下 RTDMiner 识别的函数对

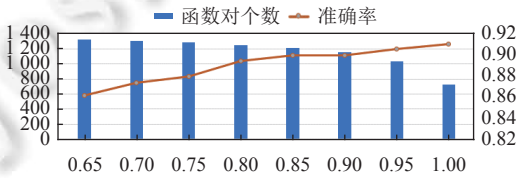

图 7 不同最小置信度 *min_confidence* 下 RTDMiner 识别的函数对

5.5 缺陷检测

RTDMiner 根据前面挖掘到的引用获取函数和引用释放函数来检测违反引用计数使用规范的缺陷代码. 换言之, 如果在某条路径上调用了引用获取函数来增加对象的引用计数, 但在对象不再使用时却没有调用对应的引用

释放函数来减少引用计数, 则该路径可能存在不恰当引用计数更新缺陷. 为评估 RTDMiner 在无人工干预的情况下检测缺陷的效果, 我们利用 RTDMiner 自动识别出的所有候选引用获取/释放函数对来检测缺陷.

根据第 5.3 节识别出的候选引用获取/释放函数, RTDMiner 从 Linux-v5.11 中检测出 3 140 个违反. RTDMiner 对这些违反按照评分从高到低进行排序. 理论上, 排行越靠前的违反越有可能是真实缺陷. 由于人力有限, 我们仅审计了前 50 个违反, 发现其中 35 个是疑似缺陷. 我们为这些疑似缺陷编写了补丁, 并提交到 Linux 内核社区. 到目前为止, 其中 24 个已经被内核维护人员确认为真实缺陷, 相应的补丁也已经被合入到主干分支. 其他 11 个疑似缺陷仍在进一步确认中. 截至目前, 我们提交的缺陷还没有被证实为误报的. 因此, 对于排行靠前的检出缺陷, 准确率约为 48%–70%.

表 3 展示了上述 24 个已经被确认的缺陷列表, 包括缺陷所在函数, 相应的引用计数获取-释放函数对, 以及我们将缺陷提交到 Patchwork (https://lore.kernel.org/patchwork) 上 Linux 内核社区所分配的补丁编号. 通常, 可以将补丁编号与 Patchwork 网址拼接来查看缺陷报告. 例如, 补丁编号为 1368826 的链接为 https://lore.kernel.org/patchwork/patch/1368826/. 需要说明的是, 表 3 中的最后一个缺陷是例外, 它被提交到了另外一个社区, 没有分配补丁编号, 可以直接访问该缺陷的确认链接 (https://patchwork.kernel.org/project/linux-arm-kernel/patch/20210120050025.25426-1-bianpan2016@163.com/) 来查看确认情况.

表 3 RTDMiner 检测到的 Linux-v5.11 中的缺陷

编号	缺陷函数	引用获取/释放函数对	PatchID
1	a2mp_createphyslink_req	<hci_dev_get, hci_dev_put>	1368826
2	hci_inquiry	<hci_dev_get, hci_dev_put>	1368638
3	bpf_fd_inode_storage_update_elem	<fget_raw, fput>	1368458
4	bsg_sg_io	<blk_get_request, blk_put_request>	1367641
5	qcom_ebi2_probe	<of_get_next_available_child, of_node_put>	1368847
6	aemif_probe	<of_get_next_available_child, of_node_put>	1368718
7	hisi_spi_nor_register_all	<of_get_next_available_child, of_node_put>	1368668
8	chtls_recv_sock	<ip6_dst_lookup_flow, dst_release>	1369011
9	igt_cs_tlb	<i915_gem_object_create_internal, i915_gem_object_put>	1369135
10	ext4_setattr	<ext4_fc_start_update, ext4_fc_stop_update>	1367564
11	affs_xrename	<affs_bread, affs_brelse>	1367960
12	do_isofs_readdir	<isofs_bread, brelse>	1367063
13	bcm_sf2_mdio_register	<of_find_compatible_node, of_node_put>	1368813
14	fec_enet_mii_init	<of_get_child_by_name, of_node_put>	1368154
15	axp20x_regulator_parse_dt	<of_get_child_by_name, of_node_put>	1368096
16	s5m8767_pmic_dt_parse_pdata	<of_get_child_by_name, of_node_put>	1368477
17	mlx5e_health_rsc_fmsg_dump	<alloc_pages, __free_page>	1368656
18	intel_eth_plat_probe	<stmmac_probe_config_dt, stmmac_remove_config_dt>	1368143
19	bcm_sysport_probe	<alloc_etherdev_mqs, free_netdev>	1367876
20	nfc_genl_stop_poll	<nfc_get_device, nfc_put_device>	1368934
21	rawsock_connect	<nfc_get_device, nfc_put_device>	1369003
22	xilinx_cpm_pcie_init_irq_domain	<of_get_next_child, of_node_put>	1368113
23	berlin2_reset_probe	<of_get_parent, of_node_put>	1368857
24	atmel_spi_probe	<spi_alloc_master, spi_controller_put>	已确认

在实验中, 我们还测试了 RTDMiner 对 Coccinelle 已公布的 25 个引用计数缺陷的检测情况来为评估本方法的检出率. 这些缺陷分布在不同的 Linux 内核版本中, 且在我们实验中所用的 Linux-v5.11 中已经被修复. 为了便于验证, 我们将这 25 个缺陷的相关代码统一移植到 Linux-v5.11 中. 在移植了已知缺陷的 Linux 内核上进行实验, RTDMiner 共检测到其中 17 个已知引用计数泄露缺陷, 其检出率为 68%. 漏报的主要原因: (1) 部分引用获取函数未识别 (见第 5.3 节); (2) 缺陷检测方法实现不够精准. 而因缺少相关知识, Coccinelle 漏报了表 3 所示的所有 24 个缺陷. 未来 RTDMiner 可以和传统分析工具 (如 Coccinelle) 进行优势互补, 以更好地看护代码安全.

5.6 缺陷实例分析

图 8 是 RTDMiner 从 Linux 内核中发现的一个真实缺陷 (在表 3 中编号为 5), 其根本原因是在第 12 行跳出循环体时没有调用引用释放函数 of_node_put(). 宏 for_each_available_child_of_node() 封装了一个 for 循环, 用来遍历所有子节点. 引用获取函数 of_get_next_available_child(*node*, *prev*) 返回节点 *node* 的下一个可用的子节点, 并增加子节点的引用计数 (第 35 行). 当参数 *prev* 是空指针时, 返回节点 node 的第 1 个可用的子节点; 而当 *prev* 不是空指针时, 则返回与 *prev* 相邻的下一个可用子节点, 同时通过引用释放函数 of_node_put() 来减少节点 *prev* 的引用计数 (第 38 行). 可见, 如果没有在循环体内跳出 for_each_available_child_of_node() 循环, 引用计数增加和减少是平衡的, 不会导致引用计数泄露问题. 而如果在循环体内跳出 for 循环, 就需要手动调用函数 of_node_put() 来减少引用计数. 而在图 8 函数 qcom_ebi2_probe() 中在跳出循环体时却没有调用函数 of_node_put(), 导致引用计数泄露缺陷.

```
    // linux-v5.11/drivers/bus/qcom-ebi2.c
1   static int qcom_ebi2_probe (struct platform_device *pdev)
2   {
3           struct device_node *np=pdev->dev.of_node;
4           struct device_node *child;
5           ...
6           /* Walk over the child nodes and see what chipselects we use */
7           for_each_available_child_of_node (np, child) {
8               u32 csindex;
9               /* Figure out the chipselect */
10              ret=of_property_read_u32 (child, "reg", &csindex);
11              if (ret)
12                  return ret;
13              ...
14          }
15          ...
16          return ret;
17  }
18
19  // linux-v5.11/include/linux/of.h
20  #define for_each_available_child_of_node (parent, child) \
21          for (child=of_get_next_available_child (parent, NULL); child!=NULL; \
22               child=of_get_next_available_child (parent, child)
23
24  // linux-v5.11/drivers/of/base.c
25  /*  of_get_next_available_child -Find the next available child node*/
26  struct device_node *of_get_next_available_child (const struct device_node *node,
27           struct device_node *prev)
28  {
29          struct device_node *next;
30          raw_spin_lock_irqsave (&devtree_lock, flags);
31          next=prev ? prev->sibling : node->child;
32          for (; next; next=next->sibling) {
33              if (!__of_device_is_available (next))
34                  continue;
35              if (of_node_get (next))   //增加下一个节点的引用计数
36                  break;
37          }
38          of_node_put (prev);     // 减少前一个节点的引用计数
39          raw_spin_unlock_irqrestore (&devtree_lock, flags);
40          return next;
41  }
```

Patch

```
of_node_put (child);
```

图 8 Linux-v5.11 中一个真实的引用计数泄露缺陷

在公开的 Coccinelle 的规则集中, 有一条专门检查在有副作用的 for 循环中因未调用函数 of_node_put() 而导致引用计数泄漏的规则 (https://coccinelle.gitlabpages.inria.fr/website/impact/missing_put.html). 但从实验结果看, Coccinelle 并没有发现图 8 所示缺陷. 主要原因在于 Coccinelle 的规则中只配置了 for_each_node_by_name(), for_each_node_by_type() 和 for_each_compatible_node() 等有限几个有副作用的宏, 而没有将 for_each_available_

child_of_node() 配置进规则中. 实际上, Coccinelle 及其他方法如 RID[6]所依赖的先验知识的主要来源是历史经验, 难免会有遗漏.

从上面的分析可见, 利用挖掘的手段自动识别引用计数获取/释放函数对检测引用计数缺陷至关重要. 客观地讲, 对于 Coccinelle 和 RID 而言, 如何发现这些知识并不是它们的工作重心, 它们主要关注如何根据已知先验知识来更高效的发现潜在缺陷. 而 RTDMiner 的工作恰好可以与 Coccinelle 和 RID 形成互补, 相信有了 RTDMiner 发现的引用计数获取/释放函数, Coccinelle 和 RID 可以发现更多潜在缺陷.

5.7 小 结

Linux 内核上的实验结果表明, 本文所提出的方法可以准确地识别引用获取/释放函数, 准确率达到近 90%, 召回率约 80%. 而在缺陷检测方面, RTDMiner 检测出 68% 的缺陷检查工具 Coccinelle 发现的已知引用计数泄露缺陷. 更重要的是, RTDMiner 自动识别出上千个通过人工归纳总结难以发现的引用获取/释放函数, 并检测到 24 个已经被 Linux 内核维护人员确认的未知缺陷. RTDMiner 可以与传统分析工具 (如 Coccinelle、RIP) 形成很好的互补.

6 讨论与未来工作展望

虽然 RTDMiner 取得了一定的成果, 可以有效地从实际大型系统中识别成百上千个引用计数获取/释放函数, 并据此检测到数十个不恰当引用计数更新缺陷, 但仍然有一些局限性需要在未来的工作中加以解决.

(1) 过程间分析. 为了提高检测效率, RTDMiner 采用过程内分析算法来检测疑似缺陷. 然而只进行过程内分析, 难以跟踪全局信息, 可能会损失检测精度. 特别是对于函数封装的场景, 可能会导致较为严重的误报漏报问题. 一方面, 如果引用获取函数被封装在另一个函数 f 中, 函数 f 的调用方就需要负责调用相应的引用释放函数. 但在过程内分析中难以进行这样的检查, 从而导致漏报; 另一方面, 如果引用释放函数被封装在另一个函数 g 中, 在过程内分析中, 即使调用了 g, 也会被误认为没有调用引用释放函数, 从而可能导致误报. 未来我们将引入过程间分析方法[38]来提高检测精度, 减少漏报和误报.

(2) 挖掘例外模式. RTDMiner 在检测缺陷时会自动识别例外模式来尽可能降低误报. 但目前我们只考虑了涉及单条语句的例外模式. 但在实际系统中可能存在多条语句组合在一起形成的例外模式. 例如, 将对象成功添加到列表中是一个例外模式, 涉及至少两条语句, 即添加对象 (如“*ret* = list_add(*l*, *o*)”) 和状态检查 (如“if (*ret*)”). 未来我们将探索挖掘包含多条语句的例外模式的方法. 一个可行的方法是首先利用频繁模式挖掘算法挖掘频繁模式[10], 然后采用类似第 4.1 节的方法识别例外频繁模式.

(3) 自动识别基本递增递减操作. 目前 RTDMiner 仍依赖一定的先验知识, 需要用户告诉它哪些是递增和递减操作. 未来我们希望也能够自动识别递增和递减函数. 可以通过程序分析技术 (包括汇编代码分析技术) 来识别对参数的值进行递增或递减的函数. 此外, 我们还可以利用类似于 SinkFinder 的方法, 对程序元素进行向量化处理, 将语义相似的编码元素映射到相似的向量[21]. 而那些与基本递增/递减操作 (如“++”或“– –”) 在向量上相近的函数则很有可能是递增/递减函数.

7 总 结

本文提出一种基于数据挖掘的不恰当引用计数更新缺陷的检测方法 RTDMiner. 首先, 根据引用计数的一般使用规律, 利用数据挖掘技术识别引用获取函数和引用释放函数. 然后, 再次利用数据挖掘技术, 识别例外模式; 最后, 进行路径敏感的分析, 检测增加引用计数后既未减少引用计数且不包含例外模式的路径, 作为潜在缺陷. 基于上述方法, 我们实现了一个原型系统, 并在 Linux 内核上进行测试. 实验结果表明, RTDMiner 可以有效识别引用计数更新函数, 准确率达到将近 90%. 此外, RTDMiner 还从 Linux 内核中检测到 24 个真实缺陷.

References:

[1] Wilson PR. Uniprocessor garbage collection techniques. In: Proc. of the 1992 Int'l Workshop on Memory Management. St. Malo: Springer, 1992. 1–42. [doi: 10.1007/BFb0017182]

[2] Levanoni Y, Petrank E. An on-the-fly reference counting garbage collector for Java. In: Proc. of the 16th ACM SIGPLAN Conf. on Object-oriented Programming, Systems, Languages, and Applications. Tampa Bay: ACM, 2001. 367–380. [doi: 10.1145/504282.504309]

[3] Bevan D. Distributed garbage collection using reference counting. In: Proc. of the 1987 Int'l Conf. on Parallel Architectures and Languages Europe. Eindhoven: Springer, 1987. 176–187. [doi: 10.1007/3-540-17945-3_10]

[4] McKenney PE, Slingwine JD. Read-copy update: Using execution history to solve concurrency problems. In: Proc. of the 1998 Parallel and Distributed Computing and Systems. 1998. 509–518.

[5] McKenney PE. Overview of Linux-kernel reference counting. Technical Report, N2167=07-0027, IBM Beaverton.

[6] Mao JJ, Chen Y, Xiao QX, Shi YC. RID: Finding reference count bugs with inconsistent path pair checking. In: Proc. of the 21st Int'l Conf. on Architectural Support for Programming Languages and Operating Systems. Atlanta: ACM, 2016. 531–544. [doi: 10.1145/2980024.2872389]

[7] Maebe J, Ronsse M, Bosschere KD. Precise detection of memory leaks. In: Proc. of the 2nd Int'l Workshop on Dynamic Analysis. 2004. 25–31.

[8] Manès VJM, Han HS, Han C, Cha SK, Egele M, Schwartz EJ, Woo M. The art, science, and engineering of fuzzing: A survey. IEEE Trans. on Software Engineering, 2021, 47(11): 2312–2331. [doi: 10.1109/tse.2019.2946563]

[9] Engler D, Chen DY, Hallem S, Chou A, Chelf B. Bugs as deviant behavior: A general approach to inferring errors in systems code. ACM SIGOPS Operating Systems Review, 2001, 35(5): 57–72. [doi: 10.1145/502059.502041]

[10] Li ZM, Zhou YY. PR-Miner: Automatically extracting implicit programming rules and detecting violations in large software code. ACM SIGSOFT Software Engineering Notes, 2005, 30(5): 306–315. [doi: 10.1145/1095430.1081755]

[11] Liang B, Bian P, Zhang Y, Shi WC, You W, Cai Y. AntMiner: Mining more bugs by reducing noise interference. In: Proc. of the 38th IEEE/ACM Int'l Conf. on Software Engineering. Austin: IEEE, 2016. 333–344. [doi: 10.1145/2884781.2884870]

[12] Yun I, Min C, Si XJ, Jang Y, Kim T, Naik M. APISAN: Sanitizing API usages through semantic cross-checking. In: Proc. of the 25th USENIX Conf. on Security Symp. Austin: USENIX Association, 2016. 363–378.

[13] Lal A, Ramalingam G. Reference count analysis with shallow aliasing. Information Processing Letters, 2010, 111(2): 57–63. [doi: 10.1016/j.ipl.2010.08.003]

[14] Malcom D. CPyChecker. 2021. https://gcc-python-plugin.readthedocs.org/en/latest/cpychecker.html

[15] Li SL, Tan G. Finding reference-counting errors in Python/C programs with affine analysis. In: Proc. of the 28th European Conf. on Object-oriented Programming. Uppsala: Springer, 2014. 80–104. [doi: 10.1007/978-3-662-44202-9_4]

[16] Kremenek T, Twohey P, Back G, Ng A, Engler D. From uncertainty to belief: Inferring the specification within. In: Proc. of the 7th Symp. on Operating Systems Design and Implementation. Seattle: USENIX Association, 2006. 161–176.

[17] Ganapathy V, King D, Jaeger T, Jha S. Mining security-sensitive operations in legacy code using concept analysis. In: Proc. of the 29th Int'l Conf. on Software Engineering (ICSE 2007). Minneapolis: IEEE, 2007. 458–467. [doi: 10.1109/ICSE.2007.54]

[18] Tan L, Zhang XL, Ma X, Xiong WW, Zhou YY. AutoISES: Automatically inferring security specification and detecting violations. In: Proc. of the 17th USENIX Security Symp. San Jose: USENIX Association, 2008. 379–394.

[19] Rasthofer S, Arzt S, Bodden E. A machine-learning approach for classifying and categorizing Android sources and sinks. NDSS. 2014, 14: 1125. http://www.bodden.de/pubs/rab14classifying.pdf

[20] Arzt S, Rasthofer S, Fritz C, Bodden E, Bartel A, Klein J, Le Traon Y, Octeau D, McDaniel P. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android Apps. In: Proc. of the 2014 ACM SIGPLAN Conf. on Programming Language Design and Implementation. Edinburgh: ACM, 2014. 259–269. [doi: 10.1145/2594291.2594299]

[21] Bian P, Liang B, Huang JJ, Shi WC, Wang XD, Zhang J. SinkFinder: Harvesting hundreds of unknown interesting function pairs with just one seed. In: Proc. of the 28th ACM Joint Meeting on European Software Engineering Conf. and Symp. on the Foundations of Software Engineering. Virtual: ACM, 2020. 1101–1113. [doi: 10.1145/3368089.3409678]

[22] Bian P, Liang B, Shi WC, Huang JJ, Cai Y. NAR-miner: Discovering negative association rules from code for bug detection. In: Proc. of the 26th ACM Joint Meeting on European Software Engineering Conf. and Symp. on the Foundations of Software Engineering. Lake Buena: ACM, 2018. 411–422. [doi: 10.1145/3236024.3236032]

[23] Yamaguchi F, Wressnegger C, Gascon H, Rieck K. Chucky: Exposing missing checks in source code for vulnerability discovery. In: Proc. of the 2013 ACM SIGSAC Conf. on Computer & Communications Security. Berlin: ACM, 2013. 499–510. [doi: 10.1145/2508859.2516665]

[24] Lu S, Park S, Hu CF, Ma X, Jiang WH, Li ZM, Popa RA, Zhou YY. MUVI: Automatically inferring multi-variable access correlations and detecting related semantic and concurrency bugs. ACM SIGOPS Operating Systems Review, 2007, 41(6): 103–116. [doi: 10.1145/1323293.1294272]

[25] Yamaguchi F, Maier A, Gascon H, Rieck K. Automatic inference of search patterns for taint-style vulnerabilities. In: Proc. of the 2015 IEEE Symp. on Security and Privacy. San Jose: IEEE, 2015. 797–812. [doi: 10.1109/SP.2015.54]

[26] Kang Y, Ray B, Jana S. Apex: Automated inference of error specifications for C APIs. In: Proc. of the 31st IEEE/ACM Int'l Conf. on Automated Software Engineering. Singapore: ACM, 2016. 472–482. [doi: 10.1145/2970276.2970354]

[27] Acharya M, Xie T, Pei J, Xu J. Mining API patterns as partial orders from source code: From usage scenarios to specifications. In: Proc. of the 6th Joint Meeting of the European Software Engineering Conf. and the ACM SIGSOFT Symp. on the Foundations of Software Engineering. Dubrovnik: ACM, 2007. 25–34. [doi: 10.1145/1287624.1287630]

[28] Pradel M, Jaspan C, Aldrich J, Gross TR. Statically checking API protocol conformance with mined multi-object specifications. In: Proc. of the 34th Int'l Conf. on Software Engineering. Zurich: IEEE, 2012. 925–935. [doi: 10.1109/ICSE.2012.6227127]

[29] Sun BY, Shu G, Podgurski A, Robinson B. Extending static analysis by mining project-specific rules. In: Proc. of the 34th Int'l Conf. on Software Engineering (ICSE 2012). Zurich: IEEE, 2012. 1054–1063. [doi: 10.1109/ICSE.2012.6227114]

[30] Fast commits for ext4. 2021. https://lwn.net/Articles/842385

[31] Hess B, Bekker H, Berendsen HJC, Fraaije JGEM. LINCS: A linear constraint solver for molecular simulations. Journal of Computational Chemistry, 1997, 18(12): 1463–1472. [doi: 10.1002/(SICI)1096-987X(199709)18:12<1463::AID-JCC4>3.0.CO;2-H]

[32] Xie YC, Chou A, Engler D. ARCHER: Using symbolic, path-sensitive analysis to detect memory access errors. In: Proc. of the 9th European Software Engineering Conf. Held Jointly with the 11th ACM SIGSOFT Int'l Symp. on Foundations of Software Engineering. Helsinki: ACM, 2003. 327–336. [doi: 10.1145/940071.940115]

[33] Ganesh V, Dill DL. A decision procedure for bit-vectors and arrays. In: Proc. of the 2007 Int'l Conf. on Computer Aided Verification. Berlin, Heidelberg: Springer, 2007. 519–531. [doi: 10.1007/978-3-540-73368-3_52]

[34] Gens D, Schmitt S, Davi L, Sadeghi AR. K-Miner: Uncovering memory corruption in Linux. In: Proc. of the 2018 Network and Distributed System Security Symp. San Diego: The Internet Society, 2018.

[35] Rubio-González C, Liblit B. Defective error/pointer interactions in the Linux kernel. In: Proc. of the 2011 Int'l Symp. on Software Testing and Analysis. Toronto: ACM, 2011. 111–121. [doi: 10.1145/2001420.2001434]

[36] Jana S, Kang YJ, Roth S, Ray B. Automatically detecting error handling bugs using error specifications. In: Proc. of the 25th USENIX Conf. on Security Symp. Austin: USENIX Association, 2016. 345–362.

[37] Engler DR, Chelf B, Chou A, Hallem S. Checking system rules using system-specific, programmer-written compiler extensions. In: Proc. of the 4th Symp. on Operating System Design & Implementation. San Diego: USENIX Association, 2000. 1.

[38] Sharir M, Pnueli A. Two approaches to interprocedural data flow analysis. New York: Courant Institute of Mathematical Sciences, Computer Science Department, New York University, 1978. https://www.cse.psu.edu/~trj1/cse598-f11/docs/sharir_pnueli1.pdf

边攀(1987－), 男, 博士, CCF 专业会员, 主要研究领域为软件缺陷检测, 缺陷自动修复.

游伟(1988－), 男, 博士, 副教授, CCF 专业会员, 主要研究领域为安全漏洞挖掘, 恶意程序分析, 移动安全, Web 安全.

梁彬(1973－), 男, 博士, 教授, 博士生导师, CCF 专业会员, 主要研究领域为信息安全, 软件分析.

石文昌(1964－), 男, 博士, 教授, 博士生导师, CCF 杰出会员, 主要研究领域为网络空间系统安全, 网络空间安全治理, 网络空间安全科学.

黄建军(1986－), 男, 博士, 讲师, CCF 专业会员, 主要研究领域为软件安全分析, 移动应用安全.

张健(1969－), 男, 博士, 研究员, 博士生导师, CCF 会士, 主要研究领域为自动推理与约束求解, 软件测试及分析.