计 算 机 研 究 与 发 展
Journal of Computer Research and Development
DOI: 10.7544/issn1000-1239.202220768
60(7): 1489−1500, 2023

基于错误路径行为一致性的内核引用计数缺陷检测

熊 忻 谈 心 张 源
(复旦大学计算机科学技术学院 上海 200438)
(xiongx18@fudan.edu.cn)

Kernel Refcount Bug Detection Based on the Consistency of Error Path Behavior

Xiong Xin, Tan Xin, and Zhang Yuan
(*College of Computer Science and Technology*, *Fudan University*, *Shanghai* 200438)

Abstract Reference counting (refcount) bugs in the kernel could cause critical security problems including memory leak and use-after-free vulnerabilities. To detect such defects, we propose a refcount bug detection system based on consistency analysis of error path behavior. Compared with the existing work, our method introduces semantic information of the error paths to infer the appropriate refcount behavior on these paths, thus detecting refcount defects cannot be covered by the existing work. First, the system identifies all the error paths in the target function based on the function return value and fault handling code. Second, path-sensitive analysis is performed to collect the specific refcount behavior on each error path within the target function, which is aggregated to infer the dominant tendency of refcount behavior of the error paths in the target function. Finally, based on the idea of consistency checking, the error paths whose refcount behavior is inconsistent with the dominant tendency are identified as potential refcount bugs. In the evaluation, the proposed system finds 21 and 9 bugs on Linux kernel version 5.6-rc2 and version 5.17, respectively, most of which have been confirmed by the kernel developers. In addition, on kernel version 5.6-rc2, the system detects 9 new refcount bugs that could not be identified by existing work.

Key words bug detection; kernel refcount bug; static program analysis; consistency analysis; error-path-based analysis

摘 要 内核中的引用计数缺陷会引起内存泄露、释放后使用漏洞等严重安全问题.针对这类缺陷,提出基于错误路径行为一致性分析的缺陷检测方案.相比已有工作,该方案引入错误路径的语义信息来推断合理的引用计数行为,从而检出以往难以覆盖的引用计数缺陷.具体而言,首先,该方案基于代码特征识别函数中所有的错误路径.其次,采用路径敏感的静态分析对各条错误路径上的引用计数行为进行分析汇总,以推断该函数在错误路径上引用计数操作的主流倾向.最终,基于一致性分析原理,将与主流倾向不一致的路径标识为潜在缺陷.实验表明,该方案在 Linux 内核版本 5.6-rc2 和版本 5.17 上分别发现 21 个和 9 个引用计数缺陷,且大部分都被开发者确认;其中,在内核版本 5.6-rc2 上有 9 个缺陷是已有工作无法覆盖的.

关键词 缺陷检测;内核引用计数缺陷;静态程序分析;一致性分析;错误路径分析

中图法分类号 TP311

收稿日期: 2022-08-29; **修回日期**: 2023-01-10

基金项目: 国家自然科学基金项目(U1836210, 62172105);上海市青年科技启明星计划项目 (21QA1400700);上海市基础研究特区计划项目 (21TQ1400100:21TQ012)

This work was supported by the National Natural Science Foundation of China (U1836210, 62172105), the Shanghai Rising-Star Program(21QA1400700), and the Shanghai Pilot Program for Basic Research(21TQ1400100:21TQ012).

通信作者: 张源(yuanxzhang@fudan.edu.cn)

引用计数(reference counting, refcount)技术是现代编程语言中一种常见的内存对象管理技术. 例如, Lisp, Python, Ruby 等语言的垃圾回收算法的实现即基于引用计数技术. 与这些编程语言相比, C 语言没有自带的垃圾回收机制. 因此, 许多用 C 语言开发的重要开源软件, 如 Linux 内核、FreeBSD 内核等, 往往需要自己实现引用计数机制来管理内存对象. 然而, 由于软件代码日益复杂, 程序开发者时常会对引用计数进行错误的操作, 引发引用计数缺陷, 进而造成内存泄露、释放后使用等内存安全问题. 这些内存安全问题会进一步被利用于执行本地提权攻击、DoS 攻击等, 严重危害内核系统的正常运行[1-2]. 近年来, 由引用计数导致的安全漏洞层出不穷(例如 CVE-2022-28356[3], CVE-2021-20226[4], CVE-2022-29581[5] 等), 对引用计数缺陷进行检测也成为构建可信内核的重要手段.

由于引用计数机制的重要性, 近年来许多研究者都在探索针对引用计数缺陷的检测方案. 火狐开发者基于动态测试的方法, 采用引用计数动态追踪与平衡技术[6] 来检测运行时发生的引用计数问题. 同时, 由于动态测试方法覆盖代码有限, 大部分的研究工作[7-11] 通过基于源代码的静态程序分析技术来检测普通用户态程序[7-9] 和操作系统内核[10-11] 中的引用计数缺陷. 其中, 针对 Linux 内核, 工具 CID[10] 基于其对内核中引用计数行为特征的观察, 提出了二维不一致性检测方案; LinKRID[11] 采用符号执行的方法, 借助局部作用域中引用计数变化与全局引用变化的数量的关系检测缺陷. 然而文献 [10−11] 所述的 2 个工作的检测方法均存在一定的局限性, 无法覆盖内核中所有引用计数缺陷的模式, 存在一定的漏报. 具体而言, CID 无法分析使用函数较少的引用对象; LinKRID 不关注内部引用对象的行为正确性. 总之, 由于 Linux 内核代码的复杂性, 当前研究者还在探索针对 Linux 内核中引用计数缺陷检测的各种不同思路和方案, 目前尚没有一个完备的检测方案.

与此同时, 在内核缺陷检测方面, 近年来国内外的许多安全研究者都关注错误处理行为在缺陷检测当中的作用. 具体来说, 当内核的功能出现错误时, 需要执行包括状态回滚等行为在内的诸多特定操作. 因此, 发生错误之后的代码路径(称为错误处理路径)本身含有特殊的程序语义, 并可为检测各类漏洞提供丰富的语义信息. Hector[12] 发现邻近的故障处理块通常需要做出相似的资源释放行为, 其检测一类由于故障处理代码中资源释放行为缺失而造成的缺陷. APEx[13] 则提出了一个识别故障处理代码的方案, 并进一步对错误检查缺失缺陷进行检测. 越来越多的研究者认识到错误路径特征可以帮助检测多类内核缺陷[14-16], 并取得了显著成果. 但是, 尚未有工作讨论错误路径对于内核引用计数缺陷检测的帮助.

本文探讨了如何利用错误处理路径提供的语义信息来进行引用计数缺陷检测, 提出了基于错误路径行为一致性分析的引用计数缺陷检测方案. 该检测方案主要基于以下观察: 尽管功能不同的函数采取的错误处理行为不同, 相应的引用计数操作也不同. 但是, 对于一个函数内部而言, 所有的错误路径对同一个被计数对象往往都维持类似的引用计数操作和行为. 这是因为, 如果一个函数中不同的错误路径造成了不同的引用变化, 那么该函数的调用者会因此产生困扰. 这意味着该函数的调用者需要准确判断该函数内部产生的错误情况, 并根据不同的情况进行相应的引用计数操作, 以确保该函数失败时最终的对象引用计数是正确的. 这会给函数调用者带来额外的开发负担, 并且极易导致更多的代码缺陷. 因此, 在内核中同一个函数中的不同错误路径应对同一个对象的引用计数有相似的故障处理行为. 这样可以保证函数发生错误退出时, 各路径上的引用计数的状态是一致的, 以便函数的调用者可以对所有错误路径做统一的引用计数处理.

基于上述观察, 本文提出了一种基于错误路径行为一致性分析的检测方案. 同一个函数内, 错误路径上的引用计数行为应趋于一致; 当大部分路径行为一致而少部分路径行为异常时, 则认为这些异常路径上存在引用计数缺陷. 具体来说, 本方案根据路径上的行为将错误路径划分为进行引用处理和不进行引用处理 2 种情况. 其中, 对引用进行处理主要包含引用计数减操作和引用对象逃逸操作 2 类具体的代码行为. 具体来说, 对于一个函数, 本方案首先识别各条错误处理路径上是否存在引用计数减和引用逃逸 2 类处理行为, 将路径对应地划分为存在处理行为和不存在处理行为 2 类. 随后统计存在处理行为和不存在处理行为的路径数量比例, 并依此评估错误路径上引用计数相关行为的一致性程度. 当一个函数中的绝大部分错误路径行为分类一致, 仅有少部分路径异常时, 则推断少数路径的行为存在问题, 即存在引用计数缺陷. 与已有工作相比, 本检测方案引入了错误路径的语义来推断函数应该采取正确的引用计数行为, 只需要引用计数对象被错误路径操作即可进行检测, 其适用限制与已有方案[10-11] 不同.

因此可以覆盖到已有工作检测不到的缺陷，并与已有工作形成检测能力的互补.

基于上述检测方案，本文实现了一个针对 Linux 内核的引用计数缺陷检测系统. 该系统以内核源代码编译而成的 LLVM（low level virtual machine）的中间表示（intermediate representation, IR）为输入，经过预处理、行为分析和缺陷检测 3 个步骤，最终输出缺陷报告. 本系统在 Linux 内核版本 5.6-rc2 和版本 5.17 上分别有 17 个和 7 个缺陷已经得到了开发者的确认. 此外，在与 CID 的对比实验中，本方案在内核版本 5.6-rc2 上发现了已有工作无法发现的 9 个引用计数缺陷. 该结果说明本文提出的新方案是有效且具有实际意义的，并且与已有工作形成检测能力的互补.

简而言之，本文的主要贡献包括 3 个方面：

1）将错误路径信息引入引用计数缺陷检测中，提出了一种基于错误路径行为一致性分析的检测方案. 该方案通过对同一个函数中的不同错误路径上的引用计数行为进行一致性分析以检测缺陷.

2）基于 1）中的检测方案，实现了一个引用计数缺陷检测系统. 该系统对错误路径和引用计数操作分别进行了自动识别与收集，分析各错误路径上的引用计数行为，最终通过一致性分析识别其中违背了函数主流倾向的行为，将其作为潜在的缺陷进行报告.

3）将检测系统应用于开源内核 Linux，发现了 Linux 内核中真实存在的缺陷. 工具在内核版本 5.6-rc2 和版本 5.17 上分别报告了 66 个和 46 个潜在缺陷，经过人工验证，分别有 21 个和 9 个报告被确认为真实的引用计数缺陷.

1 相关工作

1.1 引用计数缺陷检测

基于不同的假设，研究者们提出多种检测内核中的引用计数缺陷的方式. 这些检测方案通常的步骤为:首先，观察引用计数缺陷特征或引用计数行为通用模式；然后，根据特征进行建模；最后，筛选出不符合模型的情况作为潜在缺陷.

火狐开发者与测试者尝试使用动态测试的方法，采用引用计数追踪与平衡技术[6]，对引用计数进行追踪与平衡. 但由于动态测试极大受制于输入，更多研究采用了静态分析以及符号执行的方法来检测引用计数缺陷. Referee[7] 提出了一种利用符号模型检测的方法，但该方法仅针对全部控制流已知的封闭程序. Pungi[8] 基于函数中引用计数变化与引用对象逃逸数量相等的假设，对 Python/C 程序进行了引用计数缺陷的检测. 但这种方案需要非常精确的函数间逃逸分析，在操作系统内核中难以实际应用. RID[9] 通过比较函数参数与返回值，对函数外无法区分的路径上的引用计数行为进行不一致性检测. 但由于其假设相对严格，这种方法能检测到的范围比较小. CID[10] 基于引用计数增加行为与引用计数减少行为之间存在严格关联，以及引用计数增加行为和引用计数减少行为所使用的函数在不同的使用情境下行为具有相同倾向性的假设，提出了针对 Linux 内核的二维一致性检测方案. 但该方法依旧无法覆盖内核中的所有代码，当分析的引用计数对象被使用的函数较少时，该方案效果较差. LinKRID[11] 同样针对 Linux 内核，其采用符号执行的方法，通过判断局部作用域中引用计数变化与全局引用变化是否相同来检测缺陷. 但在检测时，LinKRID 直接排除了内部引用对象，因此它也无法覆盖所有代码. 同样针对 Linux 内核，本文提出了一种不同的检测方案，该方案主要依赖于对错误路径信息的分析. 因此，本文工作并不具有上述的限制，可以覆盖部分已有工作[10-11] 覆盖不到的代码，与其形成检测能力的互补.

1.2 利用错误路径进行缺陷检测

部分研究者观察到错误路径的语义能够被用来推断路径相关的代码行为，尤其是安全相关行为. 进一步地，研究者发现这些与错误路径相关的语义可被利用以进行缺陷检测. Hector[12] 主要研究了错误处理代码和资源释放行为之间的关系，其认为当一个故障处理块需要进行资源释放时，其附近的故障处理块通常需要做同样的资源释放行为. 研究者们还基于该假设对由于故障处理代码中资源释放行为缺失而造成的缺陷进行了检测. APEx[13] 则关注于 API 调用相关的错误处理代码. 首先，对于每一个 API，APEx 会收集不同返回值上的约束情况，并推断完整的错误处理行为. 其次，APEx 检测该 API 所有的调用点，检查每一个调用点是否都执行了完整的错误处理行为，从而识别错误处理缺失缺陷. ErrDoc[14] 将故障处理行为相关的缺陷分为 4 种表现形式，并基于这 4 种形式构建了检测工具. CHEQ[15] 提出了错误处理路径与安全检查之间的关系，通过安全检查识别错误处理代码，进而识别错误处理缺失等缺陷.

目前尚未有工作通过错误处理路径信息来检测引用计数缺陷. 与已有工作关注的错误处理缺失、内存释放缺失缺陷相比，引用计数缺陷的成因和特征更复杂，因此已有工作的技术和方案并不能直接应用

于引用计数缺陷检测场景.本文工作将错误路径信息引入引用计数缺陷检测,并实现了可行的检测方案.

2 本文检测方案

2.1 错误路径上的引用相关行为一致性

Linux 内核中的函数往往具有非常复杂的逻辑.并且在函数执行时,只有当内核状态(如全局标志位,全局对象等)满足特定条件,功能才能正常执行;反之,则会触发故障错误处理逻辑,根据情况进行中断执行、尝试修复故障或恢复执行等不同的处理.在一个函数中,发现函数执行错误并对错误故障进行处理的路径被称为错误路径.

对于错误路径上引用计数对象及其相应行为,本文有 2 点观察:

1)当一个函数发生错误时,其错误故障处理发生的位置具有一致性.具体来说,当某个函数在执行过程中遭遇错误故障后,相对应的故障处理行为可以在该函数内进行,也可以由它们的调用函数依据该函数的返回值在外部进行相应处理.但是,对于同一个函数的不同错误路径,其处理错误故障的位置基本相同.也就是说,即使一个函数触发了不同的错误路径,这些路径要么均在该函数内进行相应的故障处理,要么均不在函数内进行任何故障处理,而是留给调用者统一处理,即一个函数错误故障处理发生的位置具有一致性.从开发者的编程习惯来说,该性质也较为合理.因为,若一个函数内部的错误路径部分进行了故障处理,而另一部分没有进行,那么,该函数的所有调用者均需要了解函数内部的实现,在函数外部精心地区分内部执行的路径,并有针对性地进行相应的错误故障处理,这极大地增加了函数调用者的负担,且提高了编程错误出现的可能性.

2)基于 1)中性质,在同一个函数内的各错误路径上执行的引用相关行为具有一致性,即要么各路径都执行引用相关行为对错误故障进行处理,要么均不进行任何处理.错误路径上与引用相关的行为主要有 2 种:引用计数的减操作和引用逃逸.对大部分内核业务函数而言,其正常的业务逻辑是先获得一些重要对象的引用(伴随引用计数的增操作),然后再使用该对象进行一些业务处理.如果在后续处理过程中遭遇错误故障,则要进行相应的故障处理.如果错误路径上要执行类似“回滚”的故障处理行为,则通常会进行引用计数减操作以平衡之前的引用计数加操作,使对象的引用计数数值回滚到进入该函数之前的初态;如果错误路径要进行中断操作并退出,一般该对象会存在引用逃逸,以方便外部代码通过对逃逸的引用使用该对象进行后续操作.

2.2 基于一致性分析的缺陷检测方案

基于 2.1 节中的 2 点观察,本文提出基于错误路径引用相关行为的一致性分析来检测引用计数缺陷的方案.该方案的原理如图 1 所示.对于目标函数,首先,确定其中存在引用计数增的主要对象,称之为被计数对象.其次,将函数中的路径分为正常路径和错误路径 2 类.本方案的检测对象主要是错误路径.对于每条错误路径,定位错误路径上被计数对象的引用计数行为和引用逃逸行为.然后,依据各路径上的具体行为,将各路径划分为 2 类:一类是存在引用相关行为的路径,即存在引用计数减操作和引用逃逸;另一类是不存在引用相关行为(即不处理引用)的路径,也就是既不存在引用计数减操作,也不存在引用逃逸.划分完成后,确定数量较多的一类所具有的特征为主流行为,另一类数量较少的为异常行为.正常来说,无缺陷的函数所有的错误路径应该都会属于同一个类别,即同时属于存在引用相关行为或同时属于不存在引用相关行为,而不会存在异常行为类.一旦存在异常行为类,这些类所包含的路径都与主流类存在不一致,有可能包含缺陷,即本方案检测出的引用计数缺陷.

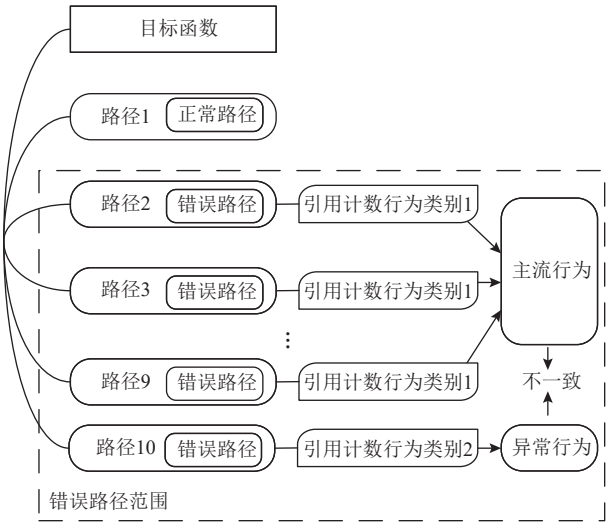


Fig. 1 Illustration of our proposed refcount bug detection scheme

图 1 本文引用计数缺陷检测方案示意图

以函数 *siw_fastreg_mr* 为例,其源代码如图 2 所示.函数在第 5 行框中通过调用函数 *siw_mem_id2obj* 进行了对象 *mem* 的引用计数增加行为,*mem* 为该函数的被计数对象.在第 8～35 行进行了一系列操作,最后在第 37 行框进行引用计数减少行为,并在第 38

行返回记录状态的变量 *rv*. 函数在第 10～28 行之间对某些内核状态和执行中间情况进行了安全检查. 安全检查失败即意味着函数执行遭遇错误，此时需要进行故障处理. 具体来说，第 11, 15, 20, 25 行相关的代码块均在执行故障处理，涉及的路径均为错误路径，具体的路径分析情况如表 1 所示.

```
1   static int siw_fastreg_mr(struct ib_pd *pd, struct siw_sqe *sqe)
2   {
3       struct ib_mr *base_mr = (struct ib_mr*)(uintptr_t)sqe→base_mr;

4       struct siw_device *sdev = to_siw_dev(pd→device);
5   ①   struct siw_mem *mem = siw_mem_id2obj(sdev,
            sqe→rkey >> 8);
6       int rv = 0;
7
8       siw_dbg_pd(pd, "STag 0x%08x\n", sqe→rkey);
9
10      if (unlikely(!mem || !base_mr)) {
11          pr_warn("siw: fastreg: STag 0x%08x unknown\n",
                sqe→rkey);
12  ②      return -EINVAL;
13      }
14      if (unlikely(base_mr→rkey >> 8 != sqe→rkey >> 8)) {
15          pr_warn("siw: fastreg: STag 0x%08x: bad MR\n",
                sqe→rkey);
16          rv = -EINVAL;
17  ③      goto out;
18      }
19      if (unlikely(mem→pd != pd)) {
20          pr_warn("siw: fastreg: PD mismatch\n");
21          rv = -EINVAL;
22  ④      goto out;
23      }
24      if (unlikely(mem→stag_valid)) {
25          pr_warn("siw: fastreg: STag 0x%08x already valid",
                sqe→rkey);
26          rv = -EINVAL;
27  ⑤      goto out;
28      }
29      /* Refresh STag since user may have changed key part */
30      mem→stag = sqe→rkey;
31      mem→perms = sqe→access;
32
33  ⑥   siw_dbg_mem(mem, "STag 0x%08x now valid\n", sqe→rkey);
34      mem→va = base_mr→iova;
35      mem→stag_valid = 1;
36  out:
37      siw_mem_put(mem);
38  ⑦   return rv;
39  }
```

Fig. 2　Code of function *siw_fastreg_mr*

图 2　函数 *siw_fastreg_mr* 代码

Table 1　Paths in Function *siw_fastreg_mr*

表 1　函数 *siw_fastreg_mr* 中的路径

路径	路径分类	错误路径行为
①⇒②	错误路径	无引用计数减，也无引用逃逸
①⇒③⇒⑦	错误路径	引用计数减
①⇒④⇒⑦	错误路径	引用计数减
①⇒⑤⇒⑦	错误路径	引用计数减
①⇒⑥⇒⑦	正常路径	

注：路径中的序号与图 2 左侧标出的带圈序号同义，用于标注代码块.

路径①～④都属于错误路径. 进一步分析这些路径上的引用相关行为可以看到，路径②～④上最终都会跳转到函数尾部进行引用计数减的操作. 只有路径①在第 12 行没有进行引用计数操作的情况下退出当前函数，且也不存在引用逃逸的情况. 显然，路径②～④会被划分为存在引用相关行为，即该函数的主流情况；而路径①是函数中的异常情况，极有可能存在引用计数缺陷.

3　系统设计

基于第 2 节所述的检测方案，本文设计并实现了一套针对内核引用计数缺陷的检测系统. 本节首先介绍该系统的架构，然后逐一介绍系统的各主要模块.

3.1　本文系统架构

本文流程图如图 3 所示. 本系统以内核源代码以及由源码编译而成的 LLVM IR[17] 文件作为输入.

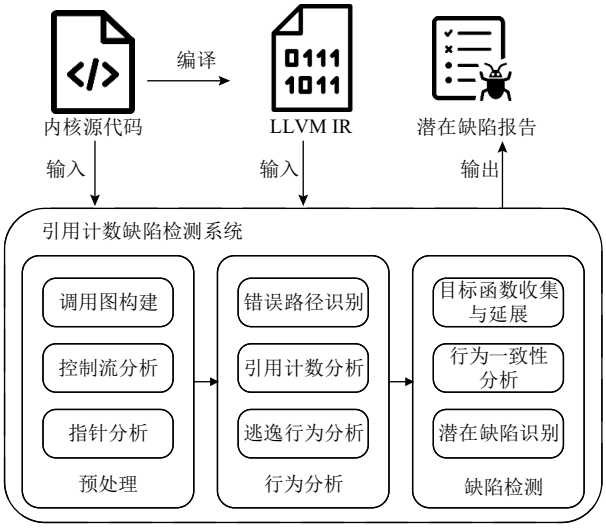


Fig. 3　Workflow of our proposed detecting system

图 3　本文检测系统流程图

分析工具主要基于 LLVM IR 进行静态程序分析，以源代码文件作为辅助信息，总共经过预处理、行为分析、缺陷检测 3 个步骤进行处理，最终产出潜在缺陷的分析报告.

1）预处理. 在预处理阶段，系统主要对输入的 IR 进行基础的静态分析，包括内核调用图构建、函数内控制流分析和函数内指针分析，为后续的分析步骤提供控制流和数据流信息支持.

2）行为分析. 在行为分析阶段，系统对目标函数内各条错误路径上的引用计数行为进行采集和分析，并传递给后续缺陷检测阶段. 具体来说，系统首先在内核中识别出所有符合引用计数域特征的结构域，即识别内核结构体中的引用计数字段，并定位所有操作这些字段的代码行为. 其次，对于涉及引用计数

操作的函数，系统开展错误路径识别，收集函数中所有的错误路径，并标记路径上的引用计数操作. 最后，进行逃逸行为分析，标记各错误路径上的引用逃逸行为.

3)缺陷检测. 在缺陷检测环节，系统将包含引用计数操作的函数作为目标函数，并基于行为分析环节对目标函数的分析结果开展行为一致性分析. 具体来说，系统对各错误路径上的具体行为进行分类，然后计算目标函数的行为一致性分数. 若分数高于阈值，则生成对应的缺陷报告. 此外，系统还迭代地对目标函数进行延展，以扩大目标函数的范围.

3.2 预处理

预处理阶段主要是进行一些基本的静态程序分析以获取控制流和数据流的信息，作为后续分析和缺陷检测的基础. 具体来说，预处理阶段主要包含调用图构建、函数内控制流分析和指针分析 3 部分.

本系统的调用图构建和函数内控制流分析主要基于 CID[10] 使用的静态分析框架实现. 在构建调用图时，对于直接函数调用，连接直接函数调用指令所在的函数和目标函数. 与此同时，内核中还存在大量的间接函数调用，本系统使用 MLTA(multi-layer type analysis)算法[18] 对间接函数调用的目的地进行识别. 对于函数内的控制流，LLVM 框架[19] 支持函数内控制流分析.

特别地，本文发现目前对 MLTA 算法的处理还不够完备. 例如，MLTA 仅能识别函数指针在结构域初始化或直接赋值的情况下被提供的情况. 然而，本文发现在内核代码中亦存在函数指针直接作为函数调用参数被提供的情况. 因此，在现有 MLTA 框架的基础上，本文针对函数指针作为函数参数被传递的情况进行建模，类似的改进还包括为函数指针逃逸至全局变量的情况提供支持等，较为完善的建模降低了原有算法引入的误报.

本系统的指针分析主要基于 SVF 框架[20]. SVF 框架在 LLVM IR 层面基于内建的内存模型进行函数间指针分析，并最终生成稀疏数据流图(sparse value-flow graph, SVFG). SVFG 以图形式表达了丰富全面的数据流信息，在后续的分析过程中提供数据流信息的支撑.

3.3 行为分析

如 2.2 节所述，本文检测系统关注错误路径上的引用计数操作和引用逃逸. 在行为分析阶段，需要对每个函数中的引用计数操作、错误路径和引用逃逸逐一进行分析.

1)引用计数分析. 引用计数分析具体分为 2 个部分，即引用计数字段的识别和字段上的相关操作识别. 通常来说，在需要通过引用计数机制来管理的对象结构体中均存在一个专门的引用计数字段. 在 Linux 内核中有 2 个专门为引用计数设计的数据类型[21]，即 refcount_t 和 kref 类型. 但是，本文发现并非所有开发者在使用引用计数时都选择这 2 种数据类型，他们也可能使用其他的数据类型(比如 atomic_t 类型)来作为引用计数字段，导致仅通过数据类型分析难以识别内核中所有的引用计数字段.

因此，本文主要基于 CID 提出的基于代码行为特征的检测方案来识别引用计数字段. 具体来说，引用计数字段上的代码行为主要有 3 个特点：

①引用计数字段的生命周期中将经历引用计数值设定、引用计数增加和引用计数减少 3 种行为.

②对于引用计数值设定行为，所有引用计数值只能被设定为 0 或 1，且至少有一个行为将引用计数值设定为 1.

③每个引用计数字段上应包含至少 1 个引用计数增加 1 和引用计数减少 1 的行为.

基于这 3 个特点，可以通过 3 个步骤识别内核中的引用计数字段. 首先，收集所有内核结构体中数据类型为 kref, refcount_t, atomic_t 等可用于引用计数目的的字段作为候选域，并且收集所有候选域上的代码行为. 然后，对于每一个候选字段，将字段上的代码行为分为设定数值、增加数值、减少数值 3 类. 接着，基于引用计数字段上的代码行为的 3 个特点对每个候选域上的行为进行审计，以筛选出符合特征的字段作为引用计数域. 此外，在 CID 的基础上，本文对一些额外情况进行了特殊建模. 本文工具还优化了部分实现，提高了对源代码进行代码行为分析的效率与准确性. 最后，收集被识别为引用计数字段为对象的操作作为引用计数操作.

2)错误路径识别. 本文主要依据路径上的错误故障处理行为来识别错误路径. 根据文献 [15]，故障处理行为大致可以分为 4 种：①返回故障码；②停止当前执行；③修复故障状态；④发送故障信息. 这 4 种行为一般是通过调用对应的处理函数或宏来实现的.

因此，本文主要通过 2 种方式来识别故障处理行为：一是定位路径上的函数返回值，通过后向数据流分析溯源返回值是否被赋值为故障码. 如图 4 所示，在第 6 行，代码判断了第 4 行该函数主要业务的返回值的状态. 如果返回值为空指针，则说明函数主要业

务执行失败，其将在第 7 行返回对应的故障码. 如果识别到了第 7 行的故障码，则可以判断经过第 7 行代码行的路径一定为错误路径. 二是收集路径上所有的函数调用和宏，通过白名单匹配的方式判断分析路径是否使用了停止当前执行/修复故障状态/发送故障信息功能的函数调用或宏. 如图 5 所示，该函数返回值为 void，因此无法通过返回值是否为故障码来识别错误路径. 但是，该函数在第 6 行检查了该函数之前的执行结果. 如果变量 *ret*<0，说明之前的执行遇到了错误，而后通过调用特定的故障信息发送函数 *pr_err* 打印故障信息. 因此，如果识别出第 7 行的故障信息发送函数，即可判断出经过第 7 行的路径一定为错误路径.

```
1  static ssize_t message_store(struct kobject *kobj,
       struct kobj_attribute *attr, const char *buf,
2        size_t count)
3  {
4      struct msg_group_t *group =
         spk_find_msg_group(attr→attr.name);
5
6      if (WARN_ON(!group))
7        return−EINVAL;
8
9      return message_store_helper(buf, count, group);
10 }
```

Fig. 4 Example of returning error code

图 4 返回故障码示例

```
1  void __init rpoc_init_cdev(void)
2  {
3      int ret;
4
5      ret = alloc_chrdev_region(&rproc_major, 0,
           NUM_PROC_DEVICES, “remoteproc”);
6      if (ret < 0)
7        pr_err(“Failed to alloc rproc_cdev region,
           err %d\n”, ret);
8  }
```

Fig. 5 Example of sending error message

图 5 发送故障信息示例

本系统的错误路径识别功能主要基于 CRIX[16] 的框架来实现，并进行了一定的额外扩展. 对于故障码，CRIX 主要支持了内核中的一些常见故障码，如 EINVAL（参数错误故障码）等. 对于故障处理函数，CRIX 使用了一个人工构建的具有故障处理功能的函数名单.

对于故障码识别，CRIX 只处理了内核通用的故障码，而本文工具实现了对模块自定义故障码的支持. 具体来说，模块自定义故障码均以宏定义或枚举类型的形式被定义. 本文借助 libclang[22] 扫描代码中所有的枚举类型，记录它们在内核中的值映射，将它们作为故障码候选. 然后，根据候选枚举类型的常量名来判断其可能为故障码的倾向程度. 如果常量名包含类似 error 等错误故障相关的词，则枚举类型更有可能是故障码. 由此，系统将生成一个映射表，映射表中包括枚举量名称、枚举量值、故障码倾向等信息. 由于将源代码编译为 IR 会丢失枚举类型常量名，因此在 IR 中分析函数的返回值以识别错误路径时，除了获取其 IR 上的具体数值以外，还会从源代码中获取其对应的字面量，通过查询上述映射表来判断该返回值是否与故障码有关.

错误路径识别伪代码如图 6 所示. 错误路径识别阶段以函数的初始静态分析结果（module）为输入，输出函数中识别出的错误处理基本块和路径，在缺陷检测阶段将利用该结果再进行具体分析. 本文工具在进行错误处理枚举类型分析后，从返回值（第 7～14 行）和函数调用（第 15～25 行）2 个维度分析所有

```
1  procedure ErrorPathIdentification(Module)
2    /*收集枚举类型元素量的名称和数值的映射，并根据常量名
       判断错误倾向，创建分析上下文*/
3    ErrorPathIdentificationContext ← ProcessEnumMap(Module);
4  /*储存识别结果*/
5  ErrorBlockOrEdge ← NewErrorSet();
6    for Func in Module:
7      ReturnValue ← getReturnValue(Func);
8      /*从函数返回值推断*/
9      if ReturnValue.ReturnType.isInt()
             or ReturnValue.ReturnType.isPointer()
             or ReturnValue.ReturnType.isEnum():
10       for ReturnValue in ReturnValueSet:
11       /*递归判断返回值是否为特定值（int）、NULL（指针类型）
           或具有错误倾向的枚举量（枚举类型）*/
12         if isErrorCode(ReturnValue):
13         /*借助数据流和控制流信息对返回值进行数据流分析，
             标注并保存结果*/
14           MarkErrorBlockOrEdgeFromReturnValue (ReturnValueInt);
15     /*从函数调用推断*/
16     for Inst in Func:
17       if Inst.isCall():
18       /*如果调用了错误处理函数，则对Func中函数调用对应的
           块和边进行标注*/
19         if isErrorHandlingFunction(Inst):
20           MarkErrorBlockOrEdgeFromErrHandlingFunction(Inst);
21         /*如果该函数调用可能返回特定值（同上），则对相应的
             块和边进行标注*/
22         if Inst.ReturnType.isInt() or Inst.ReturnType.isPointer()
               or Inst.ReturnType.isEnum():
23           if mayReturnErrorCode(Inst):
24           /*相比MarkErrorBlockOrEdgeFromReturnValue
               策略将更加保守*/
25             MarkErrorBlockOrEdgeFromCalleeReturnValue (Inst);
26    return ErrorBlockOrEdge;
27 end procedure
```

Fig. 6 Pseudocode of error path identification

图 6 错误路径识别伪代码

的函数：检测故障处理行为，并对故障发生点进行标注，其中故障发生点以基本块或基本块之间的边的形式存储. 特别地，在分析函数调用时本文也会尝试分析其返回值，但分析策略更加保守以降低误报.

3)逃逸行为分析. 在一个函数内，如果在函数退出时，一个对象的引用被传递到了函数的外部，本文称之为引用逃逸. 本系统主要关注引用被传递给函数参数、函数返回值和全局变量 3 种形式的引用逃逸. 由于内核代码的复杂性，一个函数中引用的传递关系可能极为复杂，涉及到大量的中间变量. 因此本系统主要收集函数返回值赋值、全局变量赋值、结构体类型的函数参数赋值这 3 种语句为源，开展反向数据流分析，借助 SVF[20] 提供的丰富数据流分析结果，判断在错误路径上是否存在引用逃逸行为.

3.4 缺陷检测

缺陷检测部分的伪代码如图 7 所示. 本文定义路径由基本块组成，基本块由指令组成. 缺陷检测阶段以潜在缺陷位点(〈函数调用指令，调用函数〉对)集合为输入. 对于每一个潜在缺陷位点，收集调用函数内以函数调用指令所在基本块为起点的所有路径(第 6 行). 对于每一条路径，以潜在缺陷位点为锚点，根据 3.3 节中的分析方案进行引用计数行为分析、错误路径识别以及逃逸分析，并对每条路径进行标注(第 15～22 行). 最后，评估错误路径上的行为一致性(第 26 行). 如果分数大于阈值，则进行缺陷报告(第 26～27 行)；否则，考虑延展该函数作为目标函数(第 28～29 行)，评估整个函数行为是否符合预设规则.

```
1  procedure BugDetection(CandidatePairsOfThisEpoch)
2   /*CandidatePairsOfThisEpoch存储本轮要分析的目标函数
    （指令TargetFunctionCallsite）和其调用者
    （函数CandidateCaller）*/
3   for all <TargetFunctionCallsite, CandidateCaller> in
      CandidatePairsOfThisEpoch
4   do
5    /*对控制流图进行BFS遍历收集函数内所有以调用指令为
      起点的路径*/
6    AllPathSet← CollectPathViaBFS(CandidateCaller,
      TargetFunctionCallsite);
7    ErrorPathSet← NewPathSet();  /*存放错误路径的容器*/
8    PathActionMap← NewPathActionMap();
     /*存放路径与行为的对应关系*/
9    for Path in AllPathSet
10   do
11    /*尽可能排除在基本块级别用BFS算法收集路径引入的噪声，
       即控制流图上可达但实际不可达路径*/
12    if isInfeasiblePath(Path);
13        then continue;
14    /*与3.3节1）收集的引用计数减少操作配对，若配对成功则
       对路径进行标注*/
15    if hasRefcountOperation(Path,
       RefcountOperationDetectionContext)
16      then markAction(Path, DEC, PathActionMap);
17      /*利用3.3节3）收集到的信息进行逃逸检测，若检测到逃逸
        行为则对路径进行标注 */
18    if doEscaped(Path, EscapeDetectionContext)
19        then markAction(Path, ESCAPE, PathActionMap);
20    /*利用3.3节2）中的方案判断该路径是否为错误路径，若是则
       收集该路径*/
21    if isErrorPath(Path, ErrorPathIdentificationContext)
22        then insertPath(Path, ErrorPathSet);
23   end for
24   /*评估CandidateCaller关于TargetFunctionCallsite的错误
      路径行为一致性.若一致性分数较低，则进行缺陷报告，
      若不存在缺陷，评估该函数是否可被延展为目标函数*/
25   ConsistencyScore←CalculateScoreBasedOnActions(ErrorPathSet,
      PathActionMap);
26   if ConsistencyScore < Threshold
27    then BugReport(TargetFunctionCallsite, ConsistencyScore);
28    elseif canBeExtended(CandidateCaller)
29        then insertCallPairByTargetFunction(CandidateCaller,
              CandidatePairsOfNextEpoch);
30   end for
31   return CandidatePairsOfNextEpoch;
32  end procedure
```

Fig. 7 Pseudocode of bug detection

图 7 缺陷检测伪代码

1)目标函数收集与延展. 只要一个函数中对某个在 3.3 节中被标识为引用计数的字段有引用计数增加的操作，该函数即可被作为本检测工具的目标函数. 然而此策略仅能检测直接操作引用计数字段的函数，分析范围比较小. 因此本文考虑在某些情况下，对目标函数进行迭代延展. 具体来说，在满足下面①～③条件的情况下，本文会将调用目标函数的函数作为新的目标函数进行分析.

①目标函数中不应包含与引用计数增加行为相对应的引用计数减少行为，即在目标函数中不能出现引用计数平衡的情况.

②目标函数中的引用计数对象一定会以函数参数或返回值的形式逃逸至调用函数.

③目标函数中与引用计数对象相关的数据流应该直观简单，否则会使得结果不可信.

当满足这 3 个条件时，可以对原目标函数的调用视作是一个引用计数增加的操作，将调用函数视为新的目标函数. 进一步地，将逃逸出去的引用视为新目标函数的分析对象，从而大大扩增本系统的分析范围.

2)行为一致性分析. 对于每一个目标函数，本系统都会根据 3.3 节所述对目标函数进行行为分析，然后基于分析结果进行行为一致性分析. 具体来说，系统会收集 3.3 节中所识别出的错误路径. 每一条路径会围绕目标函数中的被计数对象. 基于 3.3 节中引用计数分析和逃逸行为分析的结果，定位路径上与被计数对象有关的引用计数减操作和引用逃逸操作. 如果一条路径上含有引用计数减操作或引用逃逸，

则将该路径标识为含有引用行为的错误路径；如果路径上不包含这 2 种操作，则该路径被分类为不含引用行为的路径. 如果一个目标函数中包含 2 类路径，则说明该函数的不同路径上的行为并不一致；反之，如果一个函数中只包含同一类路径，则说明函数的行为一致.

3）潜在缺陷识别. 如果一个函数中的各条错误路径上的引用行为不一致，则该函数有可能存在潜在缺陷. 然而，由于静态程序分析的结果并不一定准确，对部分路径的引用行为分析可能存在错误，因此，如果把所有程序分析识别出的不一致行为的函数全部作为缺陷，有可能导致产生大量的缺陷报告，并且会含有大量的误报. 为了降低误报，本文引入一致性分数来量化行为的一致性程度，即函数中采取主流行为的路径比例. 具体来说，通过 3.3 节的行为分析，本文将一个函数中错误路径上的行为划分为存在引用行为和不存在引用行为 2 类，并计算这 2 类行为的比例分别为

$$Ratio_{\text{Action}} = \frac{ErrorPath_{\text{dec}} \bigcup ErrorPath_{\text{escape}}}{ErrorPath_{\text{all}}},$$

$$Ratio_{\text{noAction}} = 1 - Ratio_{\text{Action}}.$$

这 2 个比例中，比例更大的行为即为函数中的主流行为，对应的比例即为主流行为占比，也就是一致性分数：

$$Score_{\text{consistency}} = \max(Ratio_{\text{action}}, Ratio_{\text{noAction}}).$$

一致性分数越高，说明一个函数内的行为策略越一致，与主流行为背离的路径行为越可疑，越有可能存在缺陷. 因此，本文筛选出高于阈值的情况，视之为潜在缺陷位点. 阈值越高，报告数量越少，需要人工核验的成本越低，但必然会遗漏掉一些真正的缺陷；阈值越低，报告数量越多，需要大量的人工成本对结果进行核验，但能覆盖一些分数很低的缺陷. 阈值可以根据实际生成的报告分数和报告数量来进行选择. 此外,在某些特殊情况下，确实可能出现不符合本文检测策略的路径，例如在涉及异步的函数中. 因此，本文还会对通过分数筛选出的潜在缺陷位点进行简单分析，以排除这类特殊情况.

4 实验与结果分析

4.1 实验设置

本文的实验在 Ubuntu 18.04 系统上进行，所使用的 LLVM 版本为 12.0.0. 该机器具有 Intel® Xeon® Gold 6242 处理器（2.80 GHz, 32 核）. 本文分别编译了版本 5.6-rc2 和版本 5.17 的 Linux 内核，编译时所使用的配置为 allyesconfig. 编译版本 5.6-rc2 的内核得到 18 868 个 IR 文件作为系统的输入，编译版本 5.17 的内核则得到 21 188 个 IR 文件.

4.2 缺陷检测结果

如 3.4 节所述，本系统的缺陷检测需要设置一个阈值，该阈值用以筛选检测函数的一致性分数，标注潜在的引用计数缺陷. 在实验过程中，本文发现错误路径行为一致性分数在不同的内核版本上的分布存在一定差别，其累计分布图（CDF）如图 8 和图 9 所示. 在 Linux 内核版本 5.6-rc2 和版本 5.17 上，均有大量的报告聚集在 0.7 分附近，超过 60% 的缺陷报告的分数在 0.6 分以上，在 Linux 内核版本 5.17 上运行的结果分数则相对分散，且相对集中于低分段. 如果分数设置过低，可能会导致需要检查的缺陷报告数量过多. 最终，本文选择 0.625 作为共同的分数阈值，以尽可能覆盖到更多缺陷报告，同时确保缺陷报告的总数在合理范围内. 在 Linux 内核版本 5.6-rc2 上，工具一共识别到 790 个引用计数域，报告了 66 个潜在缺陷.

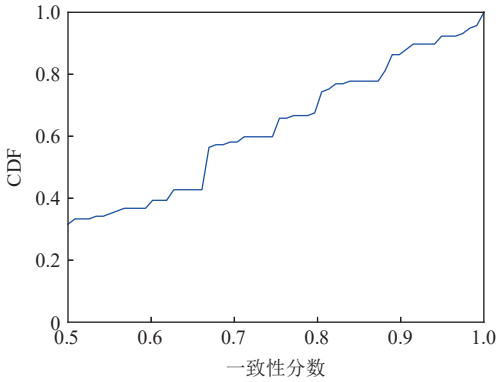


Fig. 8 CDF of reported bugs' scores on Linux kernel 5.6-rc2 version

图 8 Linux 内核 5.6-rc2 版本缺陷报告分数累计分布图

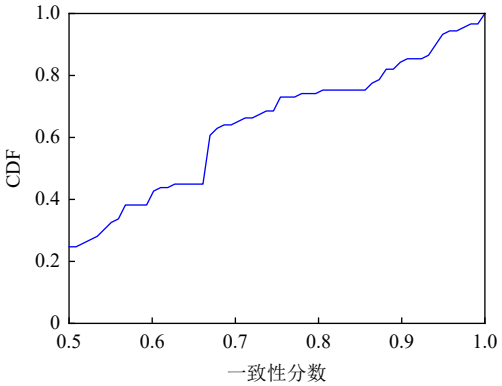


Fig. 9 CDF of reported bugs' scores on Linux kernel 5.17 version

图 9 Linux 内核 5.17 版本缺陷报告分数累计分布图

在 Linux 内核版本 5.17 上, 工具一共识别到 790 个引用计数域, 报告了 46 个潜在缺陷.

本文进一步对所有的缺陷报告进行了人工分析, 最终结果如表 2 所示. 在这 2 个版本上, 分别确认了 21 和 9 个引用计数缺陷. 对于这些缺陷, 本文也通过邮件向 Linux 内核开发者进行了汇报, 并提交了相应的补丁. 目前, 在 2 个版本上, 开发者已经确认了其中的 17 个和 7 个, 其余的缺陷尚在等待开发者的反馈.

Table 2 Results of Bug Detection

表 2 缺陷检测结果

检测版本	工具报告缺陷数量	人工确认缺陷数量	开发者确认缺陷数量
5.6-rc2	66	21	17
5.17	46	9	7

值得注意的是, 在这 2 个版本上工具汇报的结果存在一定的交集. 一方面, 对于一些工具的误报情况, 相关代码在 2 个内核版本上同时存在; 另一方面, 对于工具检测出的真实缺陷, 由于 Linux 的缺陷修复可能存在较长的时间延误, 即从缺陷被发现到对应的补丁被真正应用于新版内核之间的时间间隔较长, 本文在 2 个版本上发现了一些相同的代码缺陷(可能仅仅改变了行号). 具体来说, 在 2 个版本上, 被开发者确认的 17 个和 7 个缺陷中, 有 4 个是相同的. 此外, 虽然新版本的开发和新代码的引入会带来一些新的缺陷, 但是由于在老版本上有相当数量的缺陷被汇报并修复, 总的来说, 工具在版本 5.17 的误报率要大大高于版本 5.6-rc2.

4.3 误报原因分析

根据 3.2 节的结果可以发现, 本文提出的系统在应用过程中存在一定数量的误报. 本文进一步人工分析了版本 5.6-rc2 上 45 个误报的具体原因, 并对误报原因进行了归类. 分析结果表明, 误报主要是由三大类原因导致的:

1)由于内核代码的复杂性, 本工具使用的静态程序分析在控制流分析和数据流分析上都存在不准确的情况. 控制流分析的不准确会导致其无法准确识别错误路径; 数据流分析的不准确会导致对引用计数对象的引用计数行为分析错误. 这两者都会导致基于错误路径行为一致性分析所分析的行为对象错误, 进而造成误报, 共有 29 个误报是由于该原因导致的.

2)如 3.4 节所述, 为了扩大检测的目标函数范围, 工具在部分情况下会采用一种基于人工经验的策略对目标函数进行延展. 但是, 这种策略并不总是合理的. 当不恰当地将一个函数作为目标函数时, 也会造成误报, 共有 12 个误报是由于该原因导致的.

3)本文使用的检测策略并不一定在所有情况下都适用. 在部分特殊场景下, 开发者出于特殊考虑, 可能会故意在同一个函数中的不同错误路径上采取不同的引用计数行为, 共有 4 个误报是由于该原因导致的.

总体来看, 由于本文提出的新检测策略导致的误报, 占总体误报的比例较低(4/45), 策略本身较为可靠. 大部分误报是由于静态分析部分的分析不够准确导致的.

4.4 已有工作比较

为了研究本文提出的基于错误路径信息的检测方案对引用计数缺陷的检测能力以及与已有方案的检测能力之间的关系, 本文将 3.2 节中的检测结果与 CID[10] 进行了比较. CID 是近几年比较有价值的引用计数缺陷检测方案之一, 且同样也在 Linux 内核版本 5.6-rc2 上进行了缺陷检测. CID 在 Linux 内核版本 5.6-rc2 上共检测到 792 个引用计数域, 报告了 149 个引用计数缺陷, 其中有 44 个得到人工确认. 本文的实验结果在 Linux 内核版本 5.6-rc2 上识别出 790 个引用计数域, 总共报告了 66 个引用计数缺陷, 其中 21 个得到人工确认.

本文进一步比较了 CID 和本文所发现的引用计数缺陷, 结果如图 10 所示. 两者发现的缺陷交集有 12 个; 有 9 个缺陷只有本文工具能检测到, 而 CID 无法检测到; 有 32 个缺陷只有 CID 能检测到, 而本文工具无法检测到. 这主要是由于 2 个工具采用了截然不同的检测方案, 2 个方案适用于不同的场景, 都存在一定的局限, 因此两者的检测范围既存在重叠又有所不同. 这说明本文工作可以和已有检测工具对内核引用计数缺陷形成检测能力的互补.

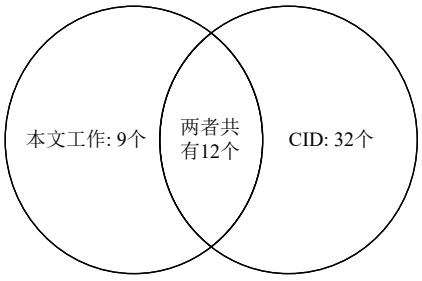


Fig. 10 Results comparison of bugs detected by our method and CID

图 10 本文与 CID 检测出的缺陷结果对比

4.5 错误路径识别效果

本文的检测方案首次将内核代码错误故障的语

义引入到引用计数检测领域. 如 3.3 节所述，错误路径的识别是后续检测模块的重要输入，直接影响了缺陷识别的效果. 因此，本文对错误路径识别模块的效果进行了评估.

本文系统在 Linux 内核版本 5.6-rc2 上分析了 12 812 个函数，其中 4 639 个函数识别到了错误路径（正样本类），8 173 个函数上没有识别到错误路径（负样本类）. 本文工作从这 2 类中分别随机选取 40 个函数进行人工分析，以验证这些函数中错误路径的分析结果. 对于每一个函数，本文工作会检查其所有路径的识别情况，只要有一条路径的识别结果是错误的，该函数就会被归类为分析错误的类. 具体结果如表 3 所示. 在被取样的未识别到错误路径的 40 个函数中，有 31 个人工确认分析正确；在被取样的未识别到错误路径的 40 个函数中，有 32 个人工确认分析正确.

Table 3 Effectiveness of Error Path Identification

表 3 错误路径识别效果

分类	函数总数	采样数量	人工分析正确数量	人工分析错误数量	准确率/%
识别到错误路径的函数	4 639	40	31	9	77.5
未识别到错误路径的函数	8 173	40	32	8	80

本文也对分析结果不正确的情况进一步深入分析. 对于误报来说，最主要的原因是静态分析技术的不准确性. 有一些不可达的路径被错误地分析为错误处理路径，或是数据流分析的错误导致工具误识别了一些错误状态码，进而错误地标记了部分路径的分类. 对于漏报来说，最主要的原因是本文对错误路径的识别策略还不够健全. 在 3.3 节中，处理了 4 种错误故障处理的情况，然而内核的错误故障处理方式多样，有许多情况无法被 3.3 节的识别策略覆盖. 但是，如何进一步提升错误路径的识别策略并不是本文工作的主要研究工作.

5 总 结

引用计数缺陷在内核中广泛存在，对内核安全构成严重威胁. 本文提出了一个新的思路，即从错误路径上的引用计数操作行为入手. 本文观察到，同一个函数中的不同错误路径对引用计数对象有相似的故障处理行为. 基于该观察，本文提出了基于错误路径行为一致性分析的引用计数缺陷检测方案，并实现了一个引用计数缺陷检测系统. 对每一个目标函数，系统首先识别出其中的错误路径，然后分析路径上的引用计数行为，最终基于一致性分析识别出引用计数相关行为背离主流倾向的路径潜在缺陷报告. 在实验中，检测工具在 Linux 内核版本 5.6-rc2 上和版本 5.17 上分别报告了 66 个和 46 个缺陷，人工确认了其中的 21 个和 9 个. 此外，与工具 CID 的比较实验表明，本文工具可以发现部分已有工具识别范围外的缺陷，形成检测能力的互补.

作者贡献声明：熊忻负责论文相关的方案设计、代码实现、实验测试以及论文撰写；谈心参与了方案设计与论文撰写；张源提出指导意见并修改论文.

参 考 文 献

[1] Zhang Jing, Huang Zhiqiu, Shen Guohua, et al. Memory leak mechanism analysis and detection of C programs[J]. Computer Engineering and Science, 2020, 42(5): 776−787 (in Chinese)
(张静, 黄志球, 沈国华, 等. C程序中的内存泄漏机制分析与检测方法设计[J]. 计算机工程与科学, 2020, 42(5): 776−787)

[2] Feng Zhen, Nie Sen, Wang Yijun, et al. Use-after-free vulnerabilities detection scheme based on S2E[J]. Computer Applications and Software, 2016, 33(4): 273−276 (in Chinese)
(冯震, 聂森, 王轶骏, 等. 基于S2E的Use-After-Free漏洞检测方案[J]. 计算机应用与软件, 2016, 33(4): 273−276)

[3] Mitre. CVE-2022−28356 [DB/OL]. 2022[2022-11-23].https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2022−28356

[4] Mitre. CVE-2021−20226 [DB/OL]. 2021[2022-11-23].https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021−20226

[5] Mitre. CVE-2022−29581 [DB/OL]. 2022[2022-11-23].https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2022−29581

[6] Firefox. Refcount tracing and balancing [DB/OL]. 2022[2022-11-23].https://firefox-source-docs.mozilla.org/performance/memory/refcount_tracing_and_balancing.html

[7] Emmi M, Jhala R, Kohler E, et al. Verifying reference counting implementations [C]//Proc of the 15th Tools and Algorithms for the Construction and Analysis of Systems. Berlin: Springer, 2009: 352−367

[8] Li Siliang, Tan Gang. Finding reference-counting errors in Python/C programs with affine analysis [C]//Proc of the 28th European Conf on Object-Oriented Programming. Berlin: Springer, 2014: 80−104

[9] Mao Junjie, Chen Yu, Xiao Qixue, et al. RID: Finding reference count bugs with inconsistent path pair checking [C]//Proc of the 21st Int Conf on Architectural Support for Programming Languages and Operating Systems. New York: ACM , 2016: 531−544

[10] Tan Xin, Zhang Yuan, Yang Xiyu, et al. Detecting kernel refcount bugs with two-dimensional consistency checking [C]//Proc of the 30th USENIX Security Symp (USENIX Security 21). Berkeley, CA: USENIX Association, 2021: 2471−2488

[11] Liu Jian, Yi Lin, Chen Weiteng, et al. LinKRID: Vetting imbalance

reference counting in Linux kernel with symbolic execution [C]//Proc of the 31st USENIX Security Symp (USENIX Security 22). Berkeley, CA: USENIX Association, 2022: 125−142

[12] Saha S, Lozi J P, Thomas G, et al. Hector: Detecting resource-release omission faults in error-handling code for systems software [C]//Proc of the 43rd Annual IEEE/IFIP Int Conf on Dependable Systems and Networks (DSN). Piscataway, NJ: IEEE, 2013: 1−12

[13] Kang Yuan, Ray B, Jana S. APEx: Automated inference of error specifications for C APIs [C]// Proc of the 31st IEEE/ACM Int Conf on Automated Software Engineering. New York: ACM, 2016: 472−482

[14] Tian Yuchi, Ray B. Automatically diagnosing and repairing error handling bugs in C [C]// Proc of the 11th Joint Meeting on Foundations of Software Engineering. New York: ACM, 2017: 752−762

[15] Lu Kangjie, Pakki A, Wu Qiushi. Automatically identifying security checks for detecting kernel semantic bugs [C]// Proc of the 24th European Symp on Research in Computer Security. Berlin: Springer, 2019: 3−25

[16] Lu Kangjie, Pakki A, Wu Qiushi. Detecting missing-check bugs via semantic-and context-aware criticalness and constraints inferences [C]//Proc of the 28th USENIX Security Symp (USENIX Security 19). Berkeley, CA: USENIX Association, 2019: 1769−1786

[17] Project LLVM. LLVM language reference manual [DB/OL]. 2022[2022-11-23].https://llvm.org/docs/LangRef.html

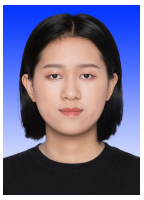
[18] Lu Kangjie, Hu Hong. Where does it go? Refining indirect-call targets with multi-layer type analysis [C]// Proc of the 26th ACM SIGSAC Conf on Computer and Communications Security. New York: ACM , 2019: 1867−1881

[19] Project LLVM. The LLVM compiler infrastructure [DB/OL]. 2022[2022-11-23].https://llvm.org/

[20] Sui Yulei, Xue Jingling. SVF: Interprocedural static value-flow analysis in LLVM [C]//Proc of the 25th Int Conf on Compiler Construction. New York: ACM, 2016: 265−266

[21] Linux kernel. Adding reference counters (krefs) to kernel objects [DB/OL]. 2018[2022-11-23].https://www.kernel.org/doc/html/v5.17/core-api/kref.html

[22] Clang. libclang: C interface to clang [DB/OL]. 2021[2022-11-23].https://clang.llvm.org/doxygen/group__CINDEX.html



Xiong Xin, born in 2000. Master candidate. Her main research interests include vulnerability detection and program analysis.

熊　忻, 2000 年生. 硕士研究生. 主要研究方向为漏洞检测和程序分析.

Tan Xin, born in 1996. PhD candidate. His main research interests include vulnerability discovery and program analysis.

谈　心, 1996 年生. 博士研究生. 主要研究方向为漏洞挖掘和程序分析.

Zhang Yuan, born in 1987. PhD, associate professor, PhD supervisor. Member of CCF. His main research interests include software security and program analysis.

张　源, 1987 年生. 博士, 副教授, 博士生导师. CCF 会员. 主要研究方向为软件安全和程序分析.