软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software, 2022,33(6):2030−2046 [doi: 10.13328/j.cnki.jos.006567]

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

基于深度学习的 Linux 内核引用计数字段识别方法*

谈 心, 杨悉瑜, 曹家俊, 张 源

(复旦大学 计算机科学技术学院, 上海 201203)

通信作者: 张源, E-mail: yuanxzhang@fudan.edu.cn

摘 要: 引用计数机制是现代软件中一种常见的内存管理技术. 引用计数错误往往会导致内存泄露、释放后使用(use after free)等严重的安全问题. 现有致力于提高引用计数安全性的工作都依赖于对引用计数的字段进行识别. 然而, 由于类似于 Linux 等软件系统的代码十分复杂, 在代码中识别出引用计数字段是一项十分困难的工作. 传统的基于代码模式匹配的引用计数字段识别方法一方面存在需要专家经验总结规则, 人工开销大的问题; 另一方面存在总结的模式无法覆盖所有情况, 召回率较低等局限. 针对这些问题, 发现与字段有关的代码行为以及字段的名称可以用来表征这个字段的特征, 帮助识别引用计数字段. 基于这两个层面的特征, 设计了一种基于多模态深度学习的引用计数字段识别方法, 并面向 Linux 内核实现原型系统. 测试数据表明: 该原型系统的精确率、召回率分别为 96.98%和 93.54%, 而传统的基于代码模式匹配的方法没有识别出任何引用计数字段. 此外, 在 Linux 内核上发现 61 个引用计数字段使用不安全的数据类型, 并对其中 21 个向 Linux 内核社区提交数据类型转换补丁以提高引用计数字段的安全性, 其中 6 个已经被合并到 Linux 内核代码主分支.

关键词: 引用计数识别; 静态程序分析; 多模态深度学习

中图法分类号: TP311

中文引用格式: 谈心, 杨悉瑜, 曹家俊, 张源. 基于深度学习的 Linux 内核引用计数字段识别方法. 软件学报, 2022, 33(6): 2030–2046. http://www.jos.org.cn/1000-9825/6567.htm

英文引用格式: Tan X, Yang XY, Cao JJ, Zhang Y. Refcount Field Identification for Linux Kernel Based on Deep Learning. Ruan Jian Xue Bao/Journal of Software, 2022, 33(6): 2030−2046 (in Chinese). http://www.jos.org.cn/1000-9825/6567.htm

Refcount Field Identification for Linux Kernel Based on Deep Learning

TAN Xin, YANG Xi-Yu, CAO Jia-Jun, ZHANG Yuan

(School of Computer Science, Fudan University, Shanghai 201203, China)

Abstract: Reference counting (refcount) is a common memory management technique in modern software. Refcount errors can often lead to severe memory errors such as memory leak, use-after-free, etc. Many efforts to harden refcount security rely on known refcount fields as their input. However, due to the complexity of software code, identifying refcount fields in source code is very challenging. Traditional methods of identifying refcount fields are mainly based on code pattern matching and have great limitations such as requiring expert experience to summarize patterns, which is a laborious job. Besides, the manually-summarized patterns do not cover all cases, resulting in a low recall. To address these issues, this studyproposes to characterize a field based on the field name and the code behaviour associated with the field; and designs a multimodal deep learning based approach. The study implements a prototype of the new approach for Linux kernel code. In the evaluation, the precision and recall achieved by the prototype system are 96.98% and 93.54%. In contrast, the traditional code-pattern-based identification method did not report any refcount fields on the testing set. In addition, sixty-one refcount fields are identified which are implemented with insecure data types in the latest Linux kernel. Until now, twenty-one of them are reported to the Linux community, of which six have been confirmed.

* 基金项目: 国家自然科学基金(U1836210, U1836213, U1736208, 61972099, 62172105); 上海自然科学基金(19ZR1404800); 上海市青年科技启明星计划(21QA1400700)

本文由“系统软件安全”专题特约编辑杨珉教授、张超副教授、宋富副教授、张源副教授推荐.

收稿时间: 2021-09-05; 修改时间: 2021-10-15; 采用时间: 2022-01-10; jos 在线出版时间: 2022-01-28

Key words: refcount field identification; static program analysis; multimodal deep learning

引用计数机制(reference count)作为一种常见的内存管理技术, 被广泛应用于现代软件开发中. 一方面, 由于 C 语言并不支持自动垃圾回收(garbage collection)或智能指针(smart pointer)这样的内存管理机制, 许多由 C 语言开发的重要程序, 如 Linux 内核、FreeBSD 内核、Mozilla Firefox 引入了引用计数机制来帮助程序员管理使用的内存; 另一方面, 如 PHP、Python 等支持内存自动回收的编程语言自身的内存管理就是基于引用计数机制实现的.

引用计数机制的核心想法是: 用一个整数来记录对当前内存对象的引用数量, 当被计数对象的引用计数为0时, 表明该对象的所有引用已不存在, 则可以将对象销毁. 由于引用计数机制在内存管理中扮演了重要角色, 因此引用计数机制相关的错误会对内存安全造成重大影响[1]. 与引用计数相关的安全问题主要有两类.

(1) 使用不安全的数据类型作为引用计数. 引用计数本质是一个有限长度的整数, 对整数进行运算面临着整数溢出的问题. 如果引用计数字段使用的数据类型本身的加减运算是没有进行整数溢出检查的, 那么当一个被计数对象的引用次数过多时, 就有可能造成引用计数的值溢出为 0, 使得一个仍在被使用中的对象被释放, 从而造成释放后使用(use after free)等严重安全问题, 比如 CVE-2014-2851、CVE-2016-4558、CVE-2016-0728;

(2) 对引用计数字段进行错误操作. 在 C 语言中, 需要程序员手动维护引用计数的更新. 错误地对引用计数进行更新, 会使得被计数对象被提前释放或者延后释放, 造成内存泄露、释放后使用等严重的安全问题, 对内存安全造成重大的安全危害[2,3], 比如 CVE-2020-25670、CVE-2019-20811.

为了缓解这些安全问题, Reshetova 等人[1]提出将 Linux 内核中存在溢出风险的引用计数字段全部更新为 refcout_t 类型, 这一类型封装了无整数溢出的运算 API, 使得操作该类型的数据时更安全. 此外, 为了检测错误的引用计数错误, Li 等人[4]提出了静态仿射分析(affine analysis)检测 C 和 Python 程序中的引用计数缺陷; Mao 等人提出基于不一致的路径对来检测 Linux 内核中的引用技术缺陷[5].

这些致力于提升引用计数安全性的工作都需要将引用计数字段作为输入. 然而, 从程序中识别出引用计数字段本身就是一个充满挑战的问题. 这主要是由于现代软件开发日益复杂, 参与的开发人员众多, 不同开发人员的开发习惯有所不同. 对于引用计数字段, 在同一个软件中, 虽然其使用约定是固定的, 但对引用计数字段的具体操作和代码风格往往是不同的. 此外, 源代码中也缺少足够的注释来标注结构体中每个字段的具体用途.

目前对引用计数字段识别的方法主要有两种: 人工识别和基于代码模式匹配(code pattern matching)的识别. 部分现有工作[4,5]通过人工审计代码来识别引用计数字段. 该方法需要花费大量的人工精力和时间, 短时间内无法识别出大量的字段. Reshetova 等人[1]观察到: Linux 内核在使用引用计数字段时, 存在固定的代码片段, 提出使用代码模式匹配来识别引用计数字段. 通过总结出 3 种代码模式, Reshetova 等人使用 Coccinelle[6] 静态分析工具识别出内核中所有符合这些代码模式的引用计数字段. 该方法需要专家经验来总结代码模式, 并且由于 Linux 内核中还有许多引用计数字段不符合这 3 种代码模式, 导致该方法存在大量漏报. 由于缺乏有效的引用计数字段识别手段, 引用计数缺陷检测工具[4,5]的检测范围受到了很大限制.

针对现有方法的这些问题, 本文提出一种新的引用计数字段识别方法. 该方法不依靠专家经验和代码模式, 而是基于字段的关联代码行为和字段名称进行引用计数字段识别. 该方法基于两个重要的观察.

(1) 由于引用计数字段的目的是记录字段所在的内存对象(即被计数对象)的引用情况, 与其他类型字段相比, 对引用计数字段的操作与对字段所在的内存对象的操作行为密切相关. 比如, 对引用计数字段的增操作往往伴随着其所在对象的引用增加; 对引用计数字段的减操作总是伴随着其所在对象的引用减少. 因此, 可以提取对字段的直接操作以及与之相关内存对象的引用操作等代码行为作为识别引用计数字段的重要特征;

(2) 除代码行为外, 开发者对字段的命名有时也表达了字段的用途, 如表示引用计数字段的名称经常是

reference count 的某种缩写(比如 ref、refcount、ref_count 等), 表示长度的字段名称经常是 length 或其缩写(比如 len 等). 因此, 字段的名称也可以作为引用计数字段识别的一个重要特征.

本文提出的两个特征, 即字段的关联代码行为和字段名称, 相比于依据从固定代码片段总结出的代码模式, 能更灵活、更通用地表征与字段有关的业务逻辑和使用目的, 从而更好地识别引用计数字段.

考虑到 Linux 内核的重要性及其使用引用计数机制的普遍性, 本文面向 Linux 内核设计了一个基于多模态深度学习的引用计数字段自动化识别系统. 具体来说, 该系统首先通过静态程序分析, 提取待分析字段的关联行为和字段名, 作为一个字段的特征——本文称之为字段签名(signature), 用于后续判断是否是引用计数字段. 其次, 考虑到一个字段签名是由多个层面的信息(行为和字段名)构成, 本文引入深度学习中的多模态融合技术[7−9]用于构建引用计数字段识别模型. 该技术可以利用不同的子神经网络捕获不同模态下的数据信息, 并将捕获到的信息融合起来作为输入, 经分类模型完成最后的识别任务. 本文提出的系统通过深度学习技术在标注数据上进行训练, 自动学习引用计数字段签名的特征, 避免了已有工作需要人工总结识别规则的缺陷.

本文在 5.9-rc1 版本的 Linux 内核上进行了评估实验. 通过人工标注 1 100 个字段用于模型的训练和测试, 实验结果表明: 本文提出的系统在测试集上取得的精确率、召回率、*F*1 值分别为 96.98%、93.54%和 95.21%, 而传统的基于代码模式匹配的方法在测试集上未能发现任何引用计数字段(*F*1 值为 0%). 此外, 本文还将系统应用于 Linux 内核中未标注字段的分析, 在 2 309 个样本中识别到 468 个引用计数字段, 为后续开展引用计数的安全分析和加固工作提供了基础.

为了说明本工作对提升引用计数安全性研究的意义, 本文选取了一个后续任务——缓解使用不安全数据类型的引用计数字段, 并进行了实验. 基于本文提出的工具, 我们在最新版本的 Linux 内核(5.14-rc1)上识别出 61 个引用计数字段使用了不安全的数据类型. 这些字段面临引用计数溢出的风险[1], 应当被更新为安全的专门为引用计数设计的数据类型. 针对其中 21 个采用了不安全数据类型的引用计数字段, 本文向 Linux 内核社区提交数据类型转换补丁以提高它们的安全性, 截至目前, 其中 6 个已经被开发者确认并且合并到 Linux 内核代码主分支中.

本文的主要工作及贡献如下:

(1) 针对引用计数字段识别问题, 本文首次提出了基于多模态深度学习的引用计数字段识别方法. 该方法以字段的关联代码行为和字段名称为特征, 利用多模态深度学习在训练集上自动学习引用计数字段识别模型. 本文的识别方法一方面能够较好地反映字段的行为特点, 另一方面, 比人工规则匹配/代码模式匹配方法更全面而高效;

(2) 本文面向 Linux 内核开发引用计数字段识别系统并进行了评估实验. 该系统在测试集上取得的精确率、召回率、*F*1 值分别为 96.98%、93.54%和 95.21%, 而传统的基于代码模式匹配的方案在测试集上未能识别出任何引用计数字段;

(3) 基于本文设计的引用计数字段识别系统, 我们在最新版本的 Linux 内核代码中发现 61 个使用了不安全数据类型的引用计数字段. 为了缓解这些引用计数字段面临的安全风险, 本文对其中 21 个进行了修补, 截至目前, Linux 社区已经接受了我们提交的 6 个安全补丁.

本文第 1 节对引用计数机制、命名实体识别、多模态深度学习等背景知识进行介绍. 第 2 节阐述本文提出的引用计数字段识别方法的设计思路和工作流程. 第 3 节对本文提出的方案进行实验评估并分析实验结果. 第 4 节介绍引用计数字段识别和安全性检测的相关工作. 第 5 节对本文的内容进行总结, 并展望后续的工作方向.

1 背景知识

1.1 Linux内核中的引用计数

引用计数本质上是一个记录资源对象引用的整数, 当一个结构体对象的引用增加时, 引用计数加 1; 当其引用减少时, 引用计数减 1; 当对象的引用计数为 0 时, 表明该对象的所有引用已不存在, 应该销毁这个对象,

释放该对象所占用的内存. 在 Linux 系统中, 引用计数器被广泛用于各种子系统来管理各种各样的资源, 如动态分配的内存块[10]、设备驱动器[11]等.

根据 Linux 内核文档[10,12−14], 为了避免并发和性能问题, 引用计数必须被定义为一个原子整数, 以支持原子操作. atomic_t、atomic_long_t 和 atomic64_t 是通用的原子类型, 可被用于任何需要保证操作原子性的场景, 不仅仅局限于引用计数. 为了将引用计数字段与其他类型的字段区别开来, Linux 内核于 2004 年引用了 kref 类型, 专门用于引用计数机制[13]. 但是使用这些数据类型作为引用计数字段均存在引用计数溢出的安全风险: 这些数据类型提供的运算 API 内部是没有进行整数溢出检查的, 如果一个攻击者一直恶意地增加一个对象的引用, 就有可能造成其引用计数的值溢出为 0 从而被提前释放, 造成释放后使用错误. 因此, Linux 内核于 2015 年引入了 refcount_t 类型. refcount_t 类型增加了额外的代码支持, 以防止引用计数器的溢出, 从而有效减少释放后的使用错误[14,15]. 在 refcount_t 被引入之后, kref 类型也迁移为基于 refcount_t 类型的实现. 值得一提的是: 虽然 refcount_t 类型和改进后的 kref 类型更安全, 但它们会产生明显的性能开销. 综上, 目前的内核中, 有 5 种数据类型可以被用于定义引用计数字段: atomic_t、atomic_long_t 以及 atomic64_t 是通用的原子类型, 用于引用计数是存在安全风险的; refcount_t 和改进后的 kref 是专门为引用计数设计的数据类型, 用于引用计数不存在溢出风险.

除了数据类型, Linux 内核还提供了原始的 API 接口来对引用计数字段进行操作. 根据 Linux 开发者手册[12−14], 有 3 类原始的 API 接口: 设置、增和减. 设置 API 将新分配对象的引用计数初始化, 如图 1(a)中第 5 行所示; 当一个新的引用被分配给被计数的对象时, 可以用增 API 让引用计数增加 1, 如图 1(c)中第 4 行所示; 而减 API 则会在被计数对象的引用变得无效时将引用计数减少 1, 如图 1(b)中第 5 行所示. 值得一提的是: 尽管增减 API 允许对字段增加或减少任何值, 但 Linux 社区建议[16], 当该字段是引用计数字段时, 每次 API 调用都应该确保只增加或减少 1. 图 1(a)−图 1(c)分别展示了设置、增、减这 3 类 API 是如何使用的.

```
/*ttm_object_file_init in drviers/gnu/drm/vmwgfx/ttm_object.c*/
1.  //分配一个tmm_object_file对象
2.  struct
ttm_object_file*tfile=kmalloc(sizeof(*tfile),GFP_KERNEL);
3.  ...
4.  //调用kref_init将其引用计数初始化为1
5.  kref_init(& tfile→refcount);
6.  …
7.  return tfile;
```

(a)

```
/*ttm_object_file_unref in drviers/gnu/drm/vmwgfx/ttm_object.c*/
1.  struct ttm_object_file*tfile=*p_tfile;
2.  //引用ttm_object_file的结构体的指针被置空
3.  *p_tfile=NULL;
4.  //调用kref_put对修改结构体的引用计数
5.  kref_put(& tfile→refcount, ttm_object_file_destroy);
```

(b)

```
/*ttm_object_file_ref in drviers/gnu/drm/vmwgfx/ttm_object.c*/
1.  static inline struct ttm_object_file*ttm_object_file_ref(struct ttm_object_file*tfile)
2.  {
3.    //调用kref_get修改引用计数
4.    kref_get(& tfile→refcount);
5.    return tfile;
6.  }
```

(c)

图 1 kref 类型 API 的使用

1.2 命名实体识别

命名实体识别(named entity recognition, NER), 是指识别文本中表示命名实体的部分, 是信息提取、句法分析、机器翻译等众多自然语言处理任务的基础工具. 字段识别任务也属于命名实体识别任务的范畴, 使用的方法和其他命名实体识别任务使用的方法具有很强的关联性. 现有的命名实体识别技术主要采取以下 3 种类型的方法.

(1) 基于规则的方法: 早期的命名实体识别研究主要依赖人工构建有限规则, 之后, 从文本中匹配符合规则的字符串作为命名实体. 之后, 一些研究者尝试利用算法自动地发现和生成规则, 比较著名的有 Collins 和 Singer[17]提出的 DL-GoTrain 方法, 其对于人名、地名和机构名这 3 种类别的分类准确率超过 91%. 传统的通过静态分析识别引用计数字段的工作同样也是基于规则. Reshetova 等人[1]总结出 3 种常见的引用计数代码模式, 使用静态分析工具在源代码中识别符合这些模式的代码片段, 再从中提取出引用计数字段;

(2) 基于传统机器学习的方法: 随着机器学习在自然语言处理领域的兴起, 许多研究者将机器学习中的常用方法和模型应用于命名实体识别任务中. Szarvas 等人[18]使用 AdaBoostM1 和 C4.5 决策树算法开发了一个多语言命名实体识别系统. McNamee 和 Mayfield[19]基于 SVM 分类器实现了 SNOOD 命名实体标签系统. 该系统只需要较少的语言学知识作为输入, 因此可以快速部署于新的目标语言. Krishnan 和 Manning[20]提出了一个两阶段的基于条件随机场(conditional random fields, CRF)的命名实体识别方法. 第 1 个 CRF 使用局部特征进行预测, 另一个 CRF 使用局部信息和第 1 个 CRF 中输出的特征进行预测;

(3) 基于深度学习的方法: 近年来, 深度学习成为了机器学习的新潮流. 深度学习将词向量作为特征, 来表示词语的含义. 这种表示方法在诸多自然语言处理中取得了显著成效, 并具有很强的普适性. 受此启发, 有许多研究将深度学习技术应用到命名实体识别任务中. Collobert 等人[21]提出了基于卷积神经网络的句子方法网络, 使用卷积层提取整个句子的特征, 将其视为具有全局结构的序列, 以提升识别效果. Huang 等人[22]首先提出将双向 LSTM-CRF 网络结构利用到序列标注和命名实体识别任务上, 该网络结构可以更好地捕获到过去和未来输入的特征以提升识别效果.

本文受到基于深度学习进行命名实体识别等工作的启发, 设计了基于多模态深度学习的方法来识别引用计数字段, 与传统的基于代码模式匹配的方法相比, 取得了更好的识别效果.

1.3 多模态深度学习

多模态深度学习(multimodal deep learning)指的是将多种模态的信息进行转换和融合, 建立统一的深度学习模型表示, 从而能够处理和关联多种模态的信息. 传统的单模态深度学习利用应用任务(如情绪分析、视频分类任务等)中单一的信息源提取信息的高维表示来完成模型学习, 而多模态深度学习关注应用任务存在多个信息来源的情况, 将多个信息要素进行融合来提升模型的表现. 这种融合技术称之为多模态融合技术. 按照融合方法的演化历程划分, 有以下 3 类融合方法.

(1) 基于特征的融合方法: 早期融合方法都是基于特征的, 其从每种模态提取了特征表示后立即进行融合. 此外, 由于部分深度学习计数会涉及到从原始数据中学习特征的表示, 有时也需要在未提取特征之前就对原始数据直接进行融合. 在多模态学习场景中, 模态之间是高度相关的, 但这种相关性很难在特征或数据层面被捕获到. Hinton 和 Salakhutdinov[23]指出: 不同数据源只有在高维表示时才能显示出一定的相关性, 基于特征的融合方法存在一定的局限性;

(2) 基于决策的融合方法: 为了克服不同数据在低维度关联性弱的问题, 后续的融合技术利用深度学习模型分别处理不同的数据源, 得到各个不同的高维表示之后再进行融合, 也称为晚期融合方法. 晚期融合方法主要采用规则来组合不同模型输出的结果, 即决策融合. Kahou 等人[24]提出了一系列规则, 如最大值融合、平均值融合等规则来组合不同模型输出的结果, 得到所有数据综合的高维表示,

再用于后续应用任务;

(3) 混合融合方法: 混合融合方法综合了前两种融合方法的优点, 可以灵活地调整模型的结构来处理复杂的多模态问题, 因此被广泛地应用在多媒体、视觉问答等领域[25].

在本文的场景中, 要处理的信息主要来自代码行为和字段名称两个维度, 两个维度原始数据的关联性较弱, 不适用于早期融合方法. 因此, 本文使用了晚期融合方法来对这两个维度的信息进行融合.

2 方案设计

本节首先介绍我们提出的引用计数字段识别方案的核心思路和设计上的考虑, 其次介绍方案的总体识别流程, 以及识别流程中的 4 个主要步骤.

2.1 核心设计思路

引用计数字段识别任务的核心问题在于, 应该提取何种信息作为字段特征用于识别任务. 对于该问题, 本文有以下两个观察.

- 观察 1: 字段关联的代码行为对引用计数字段的识别非常重要.

由于引用计数字段用于内存对象的管理, 与引用计数字段相关联的代码包含两部分: 对字段操作的相关代码; 被计数对象的引用相关代码. 图 2 展示了 *xlog_ticket* 结构体中引用计数字段 *t_ref* 的相关操作. 其中, 第 8 行和第 16 行分别进行了引用计数增减操作. 但是第 8 行和第 16 行的操作不是孤立的, 而是与函数中被计数对象的引用情况密切相关. 主要有以下两种关联.

(1) 对引用计数字段的操作由被计数对象引用的行为(尤其是引用逃逸行为)决定. 如图 2 中的 *xlog_ticket_alloc* 函数, 在第 5 行创建了一个 *xlog_ticket* 结构体 *tic*(即被计数对象), 并且在第 11 行将 *tic* 的引用通过 return 语句传递到了函数的外部, 因此需要在第 8 行将其引用设置为 1;

(2) 对引用计数字段的操作决定引用计数对象的行为. 如图 2 第 16 行所示进行了减操作, 并且检查减操作之后的值是否为 0: 如果为 0, 会将 *ticket* 对象的引用传入一个 *free* 函数(第 18 行), 将 *ticket* 函数占用的内存回收.

```
/*File: fs/xfs/xfs_log.c*/
1.  static xlog_ticket*xlog_ticket_alloc(struct xlog*log,int unit_bytes,int
2.  cnt, char client, bool permanent){
3.      static xlog_ticket*tic
4.      …
5.      tic=kmem_cache_zalloc((xfs_log_ticket_zone,GFP_NOFS_GFP_NOFAIL);)
6.      …
7.      //初始化引用计数
8.          atomic_set(&tic→t_ref,1);
9.      …
10.     将tic的引用返回到函数外部
11.     return tic;
12. }
13. void xfs_log_ticket_put(xlog_ticket_t*ticket){
14.     ASSERT(atomic_read(&ticket→t_ref)>0);
15.     //减少引用计数, 并检查是否为0
16.     if (atomic_dec_and_test(&ticket→t_ref)){
17.     //根据检查结果, 执行free操作
18.     kmem_cache_free(xfs_log_ticket_zone,ticket);
19.     }
20. }
```

图 2 引用计数字段行为示例

此外, 值得一提的是: 开发者可能会编写引用计数相关的包装函数, 通过包装函数来对引用计数操作来进行操作, 如图 1 中的 *ttm_object_file_ref*. 通过包装函数对引用计数字段进行的操作与被计数对象之间依旧存在上述两种关联.

- 观察 2: 字段的名称也在一定程度上包含这个字段的使用目的, 可以用于引用计数字段的识别.

软件开发者在开发时为了方便其他开发人员理解代码的语义以进行后续的开发, 在命名结构体的字段时, 往往会使用带有实际含义的单词或是单词的缩写来表达字段在程序中的作用, 这也可以帮助我们识别引用计数字段. 如图 2 中的引用计数字段的名称为 *t_ref*, 含有 reference count 的缩写.

综合以上两个观察, 本文提出以字段的关联代码行为和字段名称两个维度的信息为特征来表征一个字段, 称为字段签名(signature), 并通过分析字段签名来识别引用计数字段. 除此以外, 本文在方案设计时还有两点重要考虑, 以弥补之前工作的局限性.

(1) 综合考虑字段所有关联代码行为而不是根据单一代码片段作为识别标准: 对于一个待识别字段, 传统的基于代码模式匹配的方法认为只要有一处代码片段符合引用计数字段的代码特征就认为该字段是引用计数字段, 这种孤立的识别方法可能会存在一定的误报; 而本文则是将对一个字段的所有关联代码行为进行分析和合并, 根据合并后的行为特征进行识别, 是一种更为全面的识别方法. 具体来说, 本文以函数为粒度对字段行为进行分析, 对每个操作了该字段的函数进行逐一分析, 将一个函数内的关联代码行为序列作为一条记录, 最终将所有的记录合并构成字段的行为特征;

(2) 基于深度学习来学习规则而不是基于人工经验总结规则: 已有的识别方法都需要基于专家经验来总结识别规则, 这种基于人工的方法不但开销大, 而且无法覆盖所有的使用情况; 为了解决这一问题, 本文引入了深度学习技术, 通过在标注数据上进行训练, 自动学习引用计数字段的签名的特征, 具有较好的普适性. 具体来说, 本文针对上文提出的字段签名, 设计了一种基于多模态深度学习技术的多模态神经网络, 可以将关联代码行为和字段名称两个维度的特征融合起来, 再由分类模型完成最后的识别任务.

2.2 系统工作流程

基于第 2.1 节的设计思路, 本文提出了基于多模态深度学习的引用计数字段识别系统, 该系统的流程如图 3 所示. 目前, 系统的主要识别对象是 Linux 内核中的引用计数字段. 由于系统的静态分析主要是基于 LLVM 中间代码(LLVM intermediate representation)的, 因此整个系统以 Linux 内核的源代码以及源代码对应的 Bitcode 文件(即 LLVM 中间代码的二进制形式)作为输入, 输出则是识别出的引用计数字段.

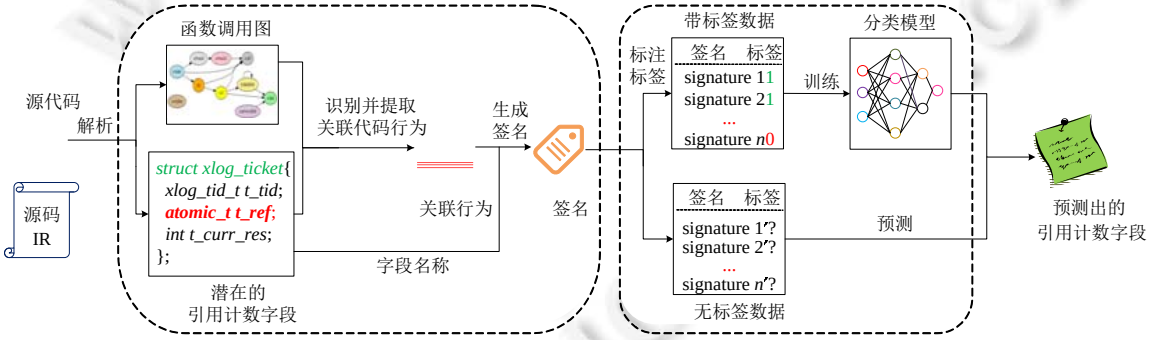

图 3 基于多模态深度学习的引用计数字段识别系统

整个系统共包括 4 个阶段.

(1) 预处理阶段: 分析内核代码, 构建函数调用图(call graph); 遍历内核中所有的结构体, 根据数据类型收集潜在的引用计数字段及字段名称;

(2) 关联行为的识别与提取: 对于每一个潜在的引用计数字段, 定位其在内核中的所有相关代码, 通过程序分析, 定位关联代码行为, 并提取关联代码行为对应的源代码;

(3) 字段签名的生成: 对关联行为对应的源代码进行文本处理(包括分词等), 并将一个字段所有的关联

行为代码进行合并, 结合字段名称构造该字段的签名;

(4) 分类模型的训练与预测: 搭建基于多模态深度学习的二分类神经网络, 对字段签名进行向量化作为模型的输入, 在标注数据集上进行训练, 得到最终的引用计数字段识别模型.

2.3 预处理阶段

在该阶段, 系统对内核代码进行初步的分析, 收集所有潜在的引用计数字段作为后续分析的对象; 同时生成函数调用图用于后续的行为分析和提取. 根据第 1.1 节, Linux 内核中共有 5 种数据类型(atomic_t, atomic_long_t, atomic64_t, kref 和 refcount_t)用于引用计数字段, 因此系统扫描内核代码中所有结构体定义, 将符合这 5 种数据类型的字段作为潜在的引用计数字段, 并记录这些字段的字段名称. 在构建函数调用图时, 本系统识别出所有的直接调用(direct call), 在调用函数(caller)和被调用函数(callee)之间添加调用关系. 对于间接调用(indirect call), 本文暂时不作处理并将其作为后续工作的改进方向.

2.4 关联行为的识别与提取

基于第 2.1 节所述, 字段关联代码行为包含了两部分: (1) 对字段进行操作的代码; (2) 被计数对象的引用相关的代码. 第 1 种代码行为主要包含对字段的增删读写. 对于第 2 种代码行为, 本文主要考虑第 2.1 节所述的两种关联情况: 被计数对象的引用行为决定引用计数操作; 引用计数操作决定被计数对象的引用行为. 为了捕获第 1 种情况, 本文分析并识别被计数对象的引用逃逸行为. 为了捕获第 2 种关联情况, 本文首先判断是否将对字段的操作的返回值作为 if 条件; 其次, 如果存在这样的判断, 后续对被计数对象进行了何种操作, 尤其是否将其传入了其他函数做进一步操作. 综上, 本文识别并提取以下 4 类关联行为.

(1) 对字段的增减、读写操作;

(2) 将字段操作的返回值作为 if 语句的条件;

(3) 被计数对象的引用被传递到函数外部, 包括通过 return 语句传递给 caller, 或是通过赋值语句传递给外部的全局变量;

(4) 对于行为(2)中的 if 语句, 当条件为真时, 对被计数对象的操作.

对于预处理阶段收集到的每一个潜在引用计数字段, 系统通过静态分析, 收集所有对该目标字段的操作, 然后对上述 4 种关联行为进行识别和提取. 如第 1.1 节所述, Linux 内核一般通过相应的 API 来操作 atomic_t、atomic_long_t、atomic64_t、kref 和 refcount_t 这 5 种数据类型的字段. 根据这一特点, 系统首先识别出代码中所有相关的 API 函数调用; 然后通过分析 API 操作的对象, 来收集所有对于该引用计数字段的操作. 除此之外, 由于内核代码的模块化, 有许多模块都基于原始的 API 进行了包装, 然后在模块内使用包装后的函数进行操作. 本文将这些包装函数也作为是对待分析字段的直接操作.

完成收集后, 对于每一个包含对目标字段操作的函数, 系统进行函数粒度的分析, 提取上述 4 类关联行为, 将行为序列作为对该函数的分析结果. 具体而言, 对于第(1)类行为, 系统根据对字段的 API 操作函数名, 可以直接识别出对应的操作类型(如 atomic_read 为读操作), 并保留出其中的增、减、读、写操作; 对于第(2)类行为, 系统对函数的指令序列进行扫描, 提取出其中所有的 if 判断, 并检查其判断的条件是否与引用计数字段有关, 从而提取出与引用计数字段有关的 if 判断.

完成了与引用计数字段相关的行为提取后, 系统进一步提取与被计数对象引用传递情况相关的代码行为. 首先通过静态分析, 系统定位出目标字段所属内存对象(即被计数对象); 然后进行函数内的 use-define 分析, 定位出所有对被计数内存对象的使用操作; 然后遍历这些使用操作, 识别并提取出第(3)类、第(4)类行为. 具体而言, 对于第(3)类行为, 系统检查使用操作是否为 return 操作, 或是对全局变量的赋值; 对于第(4)类行为, 系统定位所有将被计数对象作为参数传递给其他函数的操作, 定位到具体的函数调用位置, 并检查该函数调用所在的基本块的执行是否是由一个与目标字段相关的 if 条件决定的(如图 2 中的第 16 行–第 19 行所示); 最后, 对于一个函数内识别到的关联代码行为序列, 依次提取对应的源代码, 作为这个函数的关联行为特征.

2.5 字段签名的生成

根据第 2.1 节所述, 本文使用字段名称和字段关联代码行为表征字段特征, 称为字段签名, 并用于后续的引用计数字段识别. 图 4 展示了图 1 中 *ttm_object_file* 结构体 *refcount* 字段的字段签名. 为了生成签名, 字段名称可通过预处理阶段(第 2.3 节)直接获得, 而字段关联代码行为特征需要对第 2.4 节中识别与提取的代码作进一步的处理.

(1) 文本处理: 为了使第 2.4 节中提取到的代码更适合于作为深度学习模型的输入, 需要进行两步文本处理: 去除特殊符号和分词.

- 首先, 与自然语言文本不同, 代码中包含大量的特殊符号, 如";[{"等. 与带有操作含义的"+–"号等操作符不同, "{["这些符号并没有特殊的含义, 因此本系统会直接将这些符号删去, 替换为空格符. 而";"在代码中起到分隔语句的作用, 表明语句的终止, 因此需要保留;
- 其次, 由于代码中大量的函数名和变量名是由下划线连接的多个词组成, 这些词应该被分开理解. 因此, 系统根据下划线对这些函数名和变量名进行划分. 如图 5 中第 6 行的 kmem_cache_free 将被切分为 kmem、cache、ree 这 3 个单词;

(2) 合并为向量: 第 2.4 节对每一个操作了待分析字段的函数进行了单独的分析, 分别提取出关联行为代码. 为了综合考虑一个字段的所有关联行为, 需要将这些行为代码合并起来. 如图 4 所示, 系统将这些代码合并在一个向量中, 向量的每一个元素分别对应从一个函数中提取出的关联行为代码, 合并后的向量作为关联代码行为特征. 这些关联行为中, 包含了字段自身(即图 4 中的 *refcount*)、字段所在的内存对象(图 4 中下划线标出部分)、字段操作以及内存对象引用行为对应的动词(图 4 中斜体标出部分). 这些词与词之间的关系, 反映了字段的行为特征, 可以被用来识别引用计数字段.

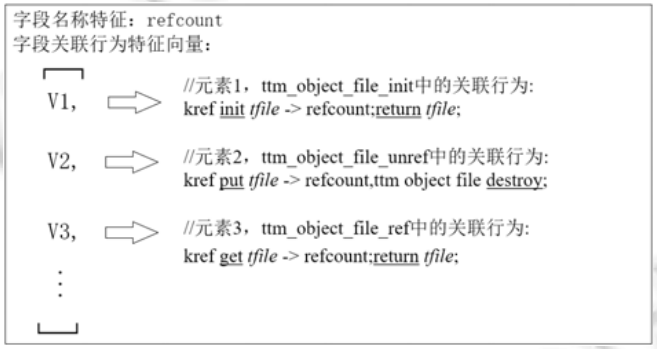

图 4 字段签名示例

经过处理后的关联代码行为特征向量和字段名称的二元组, 构成字段的签名.

2.6 多模态深度学习模型的搭建

考虑到本文提出的字段签名实际包含字段名称和字段关联代码行为两个层面的信息: 字段名称更多反映的是自然语言文本上的特征, 而字段关联代码行为反映的是代码的特征. 由于两者表示的信息维度是不同的, 单一的编码和模型难以同时在两种维度的信息上取得较好的效果. 基于这一考虑, 本文采用了多模态融合神经网络技术[7,8]. 该技术的核心思想是: 使用多个不同的子网络捕获不同维度的信息, 然后将多维度的信息融合起来.

具体来说, 本文设计的神经网络结构如图 5 所示. 首先, 对于字段行为和字段名都要先使用词嵌入技术, 将其向量化. 对于字段关联代码行为, 由于其本身的长度是不固定的, 故此本文使用长短期记忆网络(LSTM)[26,27]对其进行编码. 对于字段名, 其本身的长度较短, 分词后构成一个单词的字母间存在一定的关联关系, 因此本文使用了卷积神经网络(CNN)[28,29]对其进行编码. 经过这两个网络的编码后, 可以得到对原始

向量编码之后对应的高维向量表示. 将两个输出向量拼接, 就可以把两个不同层面的信息特征融合, 得到字段签名的高维向量表示, 作为后续分类模型的输入. 最后, 本文使用了一个全连接网络(DNN)进行分类任务. 模型在有标注的数据集上进行训练, 训练收敛后得到的模型用于后续的引用计数字段识别. 对于每一个待识别字段, 模型最终将输出一个二维向量(x,y): x 为模型认为该字段是引用计数字段的概率, y 为模型认为该字段不是引用计数字段的概率. x 和 y 二者的和为 1. 如果 x 的值大于系统设置的阈值α, 则系统认为该字段为引用计数字段.

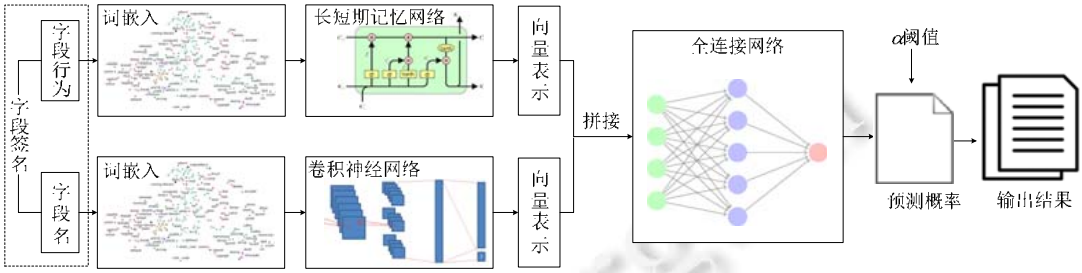

图 5 多模态神经网络结构

3 实验评估

本节以 5.9-rc1 版本的 Linux 内核代码为目标, 通过一系列实验, 评估本文提出方法在识别引用计数字段的有效性. 本节首先介绍了实验数据集的标定以及实验的主要评估指标(第 3.1 节), 然后围绕以下 5 个问题开展实验验证, 并与相关方法进行对比(第 3.2 节–第 3.6 节).

- RQ1: 模型的识别阈值α对识别效果有何影响?
- RQ2: 本文提出的方法识别效果如何, 是否优于现有方法?
- RQ3: 本文提出的方法是否存在过拟合问题?
- RQ4: 本文提出的方法在训练集以外的真实程序数据上效果如何? 是否与测试集上的效果一致?
- RQ5: 本文提出的方法如何帮助提升 Linux 内核的引用计数安全?

3.1 实验数据集和评估指标

系统在预处理阶段(第 2.3 节)从内核中提取到的待识别的引用计数字段共有 3 409 条, 其具体的数据类型分布见表 1. 本文从中随机抽取 800 条记录进行人工标注. 为确保人工标注的准确性, 本文会审计字段对应的所有代码, 根据以下原则判断一个字段是否是引用计数字段.

(1) 开发者注释中明确表明该字段为引用计数, 这些包括结构体前的注释、字段旁的注释以及其调用函数前面的注释, 其中明确出现“reference count”“refcount”等字样表明该字段为引用计数;

(2) 对于直接的字段名为“refcount”“ref_count”的字段认定为引用计数;

(3) 通过其调用函数的代码进行分析, 当涉及该字段的操作时, 结构体本身是否发生了变化, 当该字段加 1 时, 结构体是否建立了一个新的引用; 当该字段减 1 时, 结构体是否减少了一个引用; 当该字段减为 0 时, 结构体是否正确地被释放.

表 1 各类型的字段在数据集中的分布情况

数据类型	数量
atomic_t	2 101
atomic64_t	376
atomic_long_t	164
kref	436
refcount_t	332
合计	3 409

在随机抽取的 800 个字段中, 经过人工标注最终有 552 个字段为非引用计数字段, 248 个字段为引用计数字段. 由于正样本的数量大大少于负样本的数量, 考虑到非平衡的数据集可能导致模型学习效果较差, 本文又通过人工分析字段补充了一些正样本(增加了 300 个引用计数字段). 最终构建的实验数据集中, 共包括 1 100 条人工标注的字段. 对于人工标注的数据集, 我们采取随机抽取的方法以 7:3 的比例构建训练集和测试集, 得到的训练集大小为 770, 测试集大小为 330.

对引用计数字段的识别相当于一个二分类任务, 因此我们采用准确率、精确率、召回率和 $F1$ 值这 4 个常用的评估指标来对识别结果进行评估, 并与 Reshetova 等人[1]提出的基于代码模式匹配的识别方法进行比较. 计算方法分别见公式(1)–公式(4). TP 是本身为引用计数字段且被识别为引用计数字段的数量, FN 是本身是引用计数字段但识别结果不为引用计数字段的数量, FP 是本身不是引用计数字段但是被识别为引用计数字段的数量, TN 为本身不是引用计数字段且识别结果也不为引用计数字段的数量. 准确率、精确率、召回率、$F1$-score 的计算方式分别为

$$\text{准确率} = \frac{TP + TN}{TP + TN + FP + FN} \tag{1}$$

$$\text{精确率} = \frac{TP}{TP + FP} \tag{2}$$

$$\text{召回率} = \frac{TP}{TP + FN} \tag{3}$$

$$F1 = \frac{2TP}{2TP + + FP + FN} \tag{4}$$

3.2 阈值影响(RQ1)

如第 2.6 节所述, 深度学习模型最终会输出一个字段属于引用计数字段的概率, 如果该概率大于阈值α, 则该字段为引用计数字段, 反之则不是. 很明显, 阈值越小越接近于 0, 一个字段更容易被认为是引用计数字段, 反之则越不会被认为是引用计数字段; 阈值显然会影响识别的最终效果, 因此需要确定使模型效果最佳的阈值. 为了确定阈值, 本文进行了如下实验: 将模型的判定阈值设定为 0.1–0.9、步长为 0.1 分别训练模型; 对于每个阈值, 使用同样的训练集独立地训练 10 个模型, 然后在测试集上进行测试, 最后取 10 个模型的平均值作为评估结果.

实验结果见表 2. 正如前文所说, 阈值越高, 字段更容易被认为是引用计数字段. 如表 2 所示: 随着阈值由 0.9 减少到 0.1, FP 数量增多, 同时 FN 数量减少. 本文的识别任务希望误报和漏报都要尽可能地少, 需要兼顾精确率和召回率, 因此本文使用 $F1$-score 作为评判阈值的标准. 根据实验结果, 当阈值α为 0.7 时, $F1$-score 最大, 因此本文选取阈值α为 0.7 作为最终阈值. 确定阈值之后, 本文取α为 0.7 时效果最好的模型为最终模型进行后续实验(RQ2–RQ4).

表 2 不同阈值的实验结果对比

阈值α	FP	FN	准确率(%)	精确率(%)	召回率(%)	$F1$ (%)
0.9	0.40	22.60	93.03	99.73	86.22	92.44
0.8	2.20	15.80	94.55	98.54	90.37	94.25
0.7	4.80	10.60	95.33	96.98	93.54	95.21
0.6	8.40	8.00	95.03	94.93	95.12	95.01
0.5	11.60	6.20	94.61	93.24	96.22	94.69
0.4	15.80	4.20	93.94	91.16	97.44	94.15
0.3	24.20	3.00	91.76	87.23	98.17	92.29
0.2	34.00	1.60	89.21	83.17	99.02	90.26
0.1	49.60	1.00	84.67	77.31	99.39	86.78

3.3 识别有效性对比(RQ2)

为了评估本文提出方法的有效性, 本文还与现有的基于代码模式匹配的识别方法[1]进行了对比实验. Reshetova 等人[1]基于人工代码审计的经验, 总结了 3 种引用计数字段的代码模式, 并使用 Coccinelle[6]静态分

析工具扫描 Linux 内核中所有符合这些模式的代码片段, 再从中提取出引用计数字段. 本文收集了 Reshetova 等人[1]所使用的 Coccinelle Pattern, 将其作为 Coccinelle[6]的输入, 重新扫描了 5.9-rc1 版本的 Linux 内核, 并在第 3.1 节得到的测试集上与本文提出的方法进行比较.

实验评估结果见表 3. 首先, 由于 Reshetova 等人[1]总结出的代码模式是针对 atomic_t 类型的引用计数字段的, 无法适用于检测其他类型的引用计数字段; 其次, 由于代码模式是基于人工经验总结的, 无法覆盖所有的代码情况, 因此该方法存在大量的漏报, 在测试集上甚至没有检测出任何真实的引用计数字段. 与该方法相比, 本文提出的方法不局限于数据类型, 在各数据类型上都取得了良好的效果, 在测试集上的准确率、精确率、召回率和 *F*1 值分别为 95.33%、96.98%、93.54%和 95.21%, 效果远超已有的识别方案. 需要注意的是: 对于 atomic_long_t 类型, 模型的精确率和召回率较差. 这是由于 atomic_long_t 类型在内核中数量较少, 见表 1, 仅有 164 个, 是 5 种数据类型中最少的, 因此在测试集中, atomic_long_t 类型的样本也很少, 仅有 13 个. 在如此少的样本上进行测试, 结果具有一定的随机性, 因此效果较差.

表 3 本文提出的方法与代码模式匹配方法的效果对比

类型	本文的方法				代码模式匹配方法***			
	准确率(%)	精确率(%)	召回率(%)	*F*1 (%)	准确率(%)	精确率(%)	召回率(%)	*F*1 (%)
atomic_t	96.98	70.21	82.50	75.86	94.24	N/A****	0	0
atomic64_t*****	100	N/A	N/A	N/A	–	–	–	–
atomic_long_t	73.58	23.08	30.00	26.09	–	–	–	–
kref	96.94	100	96.94	98.45	–	–	–	–
refcount_t	92.46	100	92.46	96.08	–	–	–	–
合计	95.33	96.98	93.54	95.21	50.30	N/A	0	0

注释: ***. 基于代码模式匹配的识别方法只支持识别 atomic_t 类型的引用计数字段; 在计算合计的指标时, 本文认为该方法对于其他类型的字段都认为不是引用计数字段;

****. 代码模式匹配方法在测试集上, 未识别出任何引用计数字段, TP 和 FP 均为 0, 因此无法计算精确率;

*****. 测试集中没有包含 atomic64_t 字段的引用计数, 因此没有 TP; 本文的方法在 atomic64_t 上没有 FP 和 FN; 因此无法计算精确率、召回率和 *F*1-score

3.4 缓解过拟合(RQ3)

本文使用多模态神经网络结构构建模型, 训练集大小为 770 个, 由于训练集较小, 可能会存在过拟合现象. 虽然第 3.2 节、第 3.3 节的实验结果表明, 模型在测试集上已经取得了不错的效果(*F*1 值超过 95%), 但是我们仍然进一步尝试探讨如何缓解模型训练中可能存在的过拟合现象. 目前有 3 种常见的缓解过拟合和增强模型鲁棒性方法: (1) 随机失活(dropout)技术在训练过程中, 会随机使一些神经元临时不参与计算, 破坏隐层神经元已经学习到的固有关系, 减少模型对局部特征的依赖, 迫使神经网络学习更鲁棒的特征, 从而缓解过拟合; (2) 批标准化(batch normalization)技术会对深度神经网络训练过程中每一层网络输入都进行标准化, 从而稳定数据分布, 增强模型的训练速度和模型的泛化能力, 从而增强模型鲁棒性; (3) 正则化(regularization)技术通过对目标函数添加惩罚项来缓解过拟合. 具体来说, 当函数的复杂程度较高时, 意味着可能出现过拟合现象. 因此, 对于复杂程度越高的函数, 正则化惩罚越大, 从而达到削减权重参数、降低其复杂程度、增强泛化能力的目的.

将随机失活、批标准化和正则化这 3 种缓解过拟合的技术分别应用于我们的基础模型中, 构建了 3 个新的模型(称为基础模型+Dropout、基础模型+Batch Normalization、基础模型+Regularization). 基于这 3 个新的模型, 我们使用与第 3.2 节相同的训练集和测试集进行训练, 并取阈值 α 为 0.7 进行测试. 为了消除随机化的影响, 对于每一个模型结构, 我们同样独立地训练了 10 个模型, 并对其评估结果取平均, 最终结果见表 4. 对比来看, 使用了这 3 个技术之后的模型, 与基础模型相比效果并没有显著提升, 可以说明, 在该训练设置下, 第 3.2 节训练得到的基础模型并没有出现明显的过拟合现象, 可以支撑该模型在大规模真实内核数据集上进行字段识别.

表 4 不同模型的实验结果对比

模型	FP	FN	准确率(%)	精确率(%)	召回率(%)	*F*1 (%)
基础模型	4.8	10.6	95.33	96.98	93.54	95.21
基础模型+Dropout	3.5	15.4	94.27	97.78	90.89	94.21
基础模型+Batch Normalization	10.9	4.1	95.45	94.24	97.57	95.74
基础模型+Regularization	2.0	14.0	95.15	98.72	91.71	95.09

3.5 在真实内核上的识别结果(RQ4)

系统在 5.9-rc1 版本的 Linux 内核代码中提取到的待识别的引用计数字段共有 3 409 条, 其中有 1 100 条被标注的数据被用于训练和测试(第 3.1 节), 剩余 2 309 个未标注字段. 本文将最终获得的模型(第 3.2 节)应用于这 2 309 个字段进行引用计数识别. 结果见表 5, 共识别到 468 个引用计数字段.

表 5 在 Linux 5.9-rc1 上识别出的引用计数字段的情况

类型	被标注的数据集中引用计数字段的数量	无标注的数据集中, 工具识别的引用计数字段的数量	内核中该类型字段总数	总数据集中正类的比例(列 2+列 3)/列 4(%)
atomic_t	28	211	2 101	11.38
atomic64_t	0	8	376	2.13
atomic_long_t	4	7	164	6.71
kref	296	137	436	99.31
refcount_t	220	105	332	97.89
合计	548	468	3 409	29.80

为了验证模型在无标注数据集上的准确性, 对于每一种数据类型识别出的正样本和负样本, 本文各随机挑选了最多 30 条数据进行人工验证, 共分析 101 条正样本数据和 97 条负样本数据. 经过人工分析, 发现存在 9 条漏报和 13 条误报, 表明本文的工具在未标记数据集上也取得了良好的效果.

如表 5 所示, 本文在整个内核代码中共识别到了 1 016 个引用计数字段. 可以看到, 引用计数字段在不同数据类型上的分布情况差异很大. 对于 atomic_t 等基本原子类型, 由于这些数据类型也可用于其他字段, 如统计数据长度或是记录系统状态, 因此只有 2.13%–11.38%的字段为引用计数字段. 此外, kref 和 refcount_t 作为专门用于引用计数的数据类型, 理想情况下, 使用了这些数据类型的字段应该都为引用计数字段. 但实验结果表明, 这些数据类型的正类比例并没有达到 100%. 这是因为, 在部分情况下, 开发者会将这些数据类型用于一些同样需要确保操作原子性的字段上. 比如在 task_struct 中存在一个 refcount_t 类型的字段 rcu_users, 该字段是用于记录代码中另一个名为 rcu 的结构体的使用者的数量的, 而不是 task_struct 的引用数量. 这些结果充分说明了引用计数字段识别的意义以及本文提出的系统在识别引用计数字段上取得了良好的效果.

3.6 将结果应用于提升引用计数安全(RQ5)

为了探讨本工作识别出的引用计数字段如何帮助提升 Linux 内核中引用计数的安全性, 本文尝试缓解使用不安全数据类型的引用计数字段这一安全问题. 如第 1.1 节所述, 使用 atomic_t 类型进行引用计数存在安全风险, 而 refcount_t 类型因为对运算操作添加了较好的安全检查, 因此被认为不存在加法溢出的问题, 更加适用于引用计数字段类型[1]. 本工作尝试检测 Linux 内核中使用了 atomic_t 类型的引用计数字段, 并开发补丁将其转换为安全的专门为引用计数设计的数据类型, 以提高引用计数字段的安全性.

在第 3.5 节, 本文在 5.9-rc1 版本的 Linux 内核代码中一共识别到了 239 个 atomic_t 类型的引用计数字段. 本文随机分析了其中的 94 个字段, 其中有 4 个字段在最新版的内核代码上不存在, 有 10 个在最新版本的 Linux 内核上已经被开发者修改为 refcount_t 类型, 有 19 个属于工具的误报. 需要注意的是, 本实验的检测对象是 atomit_t 类型的引用计数字段, 该实验中的精确率 79% (75/94)甚至优于测试集上 atomic_t 类型的精确率(70.21%, 见表 3). 针对剩余的 61 个引用计数字段, 本文认为, 它们在最新版本的 Linux 内核(5.14-rc1)上依旧存在溢出的安全风险, 因此需要考虑将其数据类型转化为 refcount_t 类型. 在进行修复的过程中, 本文有两个额外的考虑: 一是某些通用的结构体在内核代码中被广泛使用, 对其修改存在较大的风险; 二是某些原有字段代码的功能无法通过 refcount_t 类型提供的 API 来实现, 更换类型会对原功能造成破坏. 对于这两种情况,

本文决定暂缓对其进行安全性提升. 最终, 本文针对 21 个采用了不安全数据类型的引用计数字段, 向 Linux 内核社区提交数据类型转换补丁以提高它们的安全性, 截至目前, 其中 6 个已被开发者确认并且合并到 Linux 内核代码主分支中.

4 相关工作

本文的相关工作主要有两方面: 引用计数字段的识别和检测引用计数缺陷.

4.1 引用计数字段识别

致力于提升引用计数安全性的工作都需要将引用计数字段作为输入, 它们大多依赖人工识别出引用计数字段或是操作引用计数的 API[4,5]. 与本文最为接近的工作是 Reshetova[1]等人提出的基于代码模式匹配的识别方法. 基于人工代码审计的经验, Reshetova[1]等人总结出 atomic_t 类型的引用计数字段在 Linux 内核中存在 3 种常见的代码模式: (1) 以 atomic_dec_and_test 的返回值作为条件, 决定是否释放一个对象; (2) 使用 atomic_add_return, 将引用计数与–1 相加, 并将返回值与 0 比较. 这其实是 atomic_dec_and_test 操作的变种; (3) 当引用计数的值不为 1 时, 调用 atomic_add_unless 将引用计数的值减 1. 为了在 Linux 内核上识别出这 3 种代码模式, 文献[1]使用了静态代码匹配工具——Coccinelle[6]. 文献[1]将这 3 种代码模式编写为 Coccinelle pattern 形式, 作为输入由 Coccinelle 扫描 Linux 内核代码, 匹配存在这些模式的代码片段. 对于匹配到的代码片段, 人工地从中提取出代码片段中操作的字段作为引用计数字段. 在 4.10 版本的 Linux 内核上, 该方法识别出 250 个引用计数字段, 经人工验证, 其中 233 个是正确的, 剩余 27 个是误报. 第 3.3 节的对比实验表明, 本文方法的识别效果远超该方法. 该方法既无法识别 refcount_t 与 kref 两种专门为引用计数设计的数据类型的字段, 对于 atomic_t 类型的识别效果也很差(召回率为 0%).

4.2 检测引用计数缺陷

已有研究主要关注两方面的引用计数缺陷问题.

- 一是缓解使用不安全的数据类型作为引用计数. Reshetova[1]等人提出使用带有运算溢出检查的 refcount_t 类型来替代原有的面临整数溢出风险的 atomic_t 类型, 作为引用计数字段的专用数据类型; 并向 Linux 社区提交了 233 个补丁, 对已有的不安全的引用计数字段的类型进行转换, 其中 123 个被接受;
- 二是检测错误的引用计数操作. Li 和 Tan 等人提出了 Pungi[4], 该工具的检测规则是: 内存对象在一个函数中引用计数的变化应该等于从该函数逃逸的引用的数量. 基于该规则, Pungi 通过静态代码分析在 Python 和 C 程序中检测出了超过 150 个引用计数错误. Mao 等人提出了 RID[5], 他们提出一个函数中两条参数和返回值都相同的路径应该进行相同的引用计数操作; 如果两条路径上的引用计数操作不同, 则称为不一致路径对. RID 通过检测不一致的路径对来检测错误的引用计数操作, 并在 Linux 内核上发现了超过 100 个与引用计数操作相关的缺陷.

以上这些工作都以引用计数字段作为研究对象, 本文提出的引用计数字段识别方法, 为这些工作奠定了基础.

5 讨 论

本节将对本文方案设计中的一些重要问题进行讨论.

5.1 用于引用计数字段识别的特征选取

正如本文最初部分和第 4.1 节所说, 已有的引用计数字段识别工作主要依赖专家知识构造高度定制化的规则来识别引用计数字段. 这种方法一方面需要人工地对代码进行大量审计, 另一方面总结的规则可能存在遗漏导致漏报. 针对这一问题, 本文提出一种基于机器学习的引用计数字段识别方案, 通过在标注的数据集

上进行模型训练, 自动学习引用计数字段的特点. 如第 2.4 节所述, 本文主要考虑两类引用计数字段特征: (1) 字段的名称; (2) 字段关联的代码行为. 根据引用计数机制的设计, 引用计数字段的这两类特征明显区别于普通内核字段, 可以帮助模型有效识别引用计数字段. 然而, 可能还存在其他类型的特征可以帮助本方案取得更好的引用计数字段识别效果, 包括其他类型的使用这两类特征的方式, 本文将在后续工作进一步探索.

5.2 使用深度神经网络进行引用计数字段识别

在引用计数字段识别任务中, 很难构造出一个规模较大的数据集用于模型的训练. 通常来说, 在数据量较小时, 往往使用传统的机器学习方法, 如 SVM 等; 使用深度神经网络可能遇到过拟合问题. 然而, 在本文的设计方案中, 直接使用传统机器学习方法存在两方面的问题: 首先, 如第 2.6 节所述, 提取出的字段行为和字段名称都需要通过词嵌入将其向量化作为模型输入, 为了更好地得到两者的向量化表示, 在训练过程中需要通过后续网络的反馈来更新使用的词嵌入模型, 使用深度神经网络更合适; 其次, 第 2.4 节提取出的关联行为特征是不定长的, 不定长的数据通常需要使用长短期记忆网络等深度学习模型进行处理. 基于上述原因, 本文使用了深度神经网络来构建模型. 第 3 节的一系列实验结果表明: 本文设计的模型取得了不错的效果, 同时并没有出现明显的过拟合问题.

此外, 针对如何更好地表征不定长的文本信息, 除了使用长短期记忆网络模型这一传统思路外, 近期学术界还有许多新进展. 许多研究提出了更为复杂的深度神经网络模型, 如 ELMo[30]、Attention Mechanism[31]、Transformer[32]等, 用于处理文本表征问题, 以提升各自然语言处理任务的准确率和效率. ELMo[30]改进了传统的表征技术, 使得一个单词的向量表征不再是一个固定值, 而是会结合这个单词的上下文来推断出这个单词的向量表征, 提升了单词向量表征的泛化性, 从而进一步提升了下游任务的表现. 注意力机制(attention mechanism)[31]可以让模型更关注于某些重要的单词, 而忽略一部分与任务无关的单词, 以便综合提升下游任务的表现, 并能提供一定程度上的模型解释. Transformer[32]模型提供了一种处理序列化模型的新思路, 用多个 Attention 的层级结构来代替传统的循环神经网络, 纯粹依赖于注意力机制来构建各个单词之间的联系, 并在多个下游任务中都取得了出色的表现. 未来, 我们计划继续探索这些复杂网络在本文提出的多模态模型中的效果, 以进一步提升引用计数字段的识别准确率. 此外, 引入 Attention 机制也可以帮助解释我们的多模态模型中, 究竟哪些特征更重要, 更能反映出引用计数字段的特征.

5.3 与引用计数字段特征耦合的深度学习模型设计

本文在引用计数字段识别模型中, 使用字段名称和字段关联代码行为两种不同维度的特征. 为了将两种特征融合在一个模型中, 本文使用多模态深度学习模型. 如第 1.3 节所述, 基于决策融合方法的多模态深度学习模型需要对不同层面的特征作进一步处理, 得到其高维特征向量表示后再进行融合. 因此, 对于行为特征, 我们首先使用词嵌入将其向量化, 之后针对其不定长的特点, 使用长短期记忆网络进行处理; 对于字段名称特征, 同样先使用词嵌入将其向量化, 之后为了捕获其字母间的关系, 本文使用了卷积神经网络进行处理; 将获得的高维向量表示合并后, 我们使用全连接神经网络作为分类器, 使神经网络的反馈向前传导, 帮助调整词嵌入网络的权重. 可以看出, 本文的模型设计与选用的特征是高度耦合的. 后续如果考虑其他维度的特征, 或者单独使用一种特征进行引用计数字段识别, 都需要对原有的模型结构进行较大的调整. 同时, 本文提出的将特征空间与模型结构相结合的设计, 帮助我们在内核引用计数字段识别任务上取得了较好的效果.

6 总结与展望

针对 Linux 内核中引用计数字段的识别问题, 本文工作提出了基于多模态深度学习的引用计数字段识别方法. 与传统的通过代码模式匹配的方法不同, 本文方法以字段的关联代码行为和字段名称作为引用计数字段的识别特征, 从而显著提升了识别效果. 此外, 本文使用深度学习技术, 从标注样本中自动学习引用计数字段的特征, 而不用基于专家经验总结识别规则, 可以识别更广泛的引用计数字段.

本文在 5.9-rc1 版本的 Linux 内核上进行了评估实验. 在测试集上, 本文方法的精确率、召回率、*F*1 值等

指标均远超传统的基于代码模式匹配的方法. 此外, 基于引用计数字段识别结果, 本文在最新的 Linux 内核上发现了 61 个存在内存安全风险的字段, 并对其中的 21 个字段开发了相应的补丁, 将其迁移至安全的数据类型. 在后续工作中, 我们还计划基于新识别出的引用计数字段, 开展引用计数操作缺陷的检测工作, 以进一步提升 Linux 内核的引用计数安全性.

References:

[1] Reshetova E, Liljestrand H, Paverd A, Asokan N. Toward Linux kernel memory safety. Software: Practice and Experience, 2018, 48(12): 2237−2256.

[2] Feng Z, Nie S, Wang YJ, Xue Z. Use-after-free vulnerabilities detection scheme based on S2E. Computer Applications and Software, 2016, 33(4): 273−276 (in Chinese with English abstract).

[3] Zhang J, Huang ZQ, Shen GH, Yu YS, Ai L. Memory leak mechanism analysis and detection of C programs. Computer Engineering and Science, 2020, 42(5): 776−787 (in Chinese with English abstract).

[4] Li S, Tan G. Finding reference-counting errors in Python/C programs with affine analysis. In: Proc. of the European Conf. on Object-oriented Programming. 2014. 80−104.

[5] Mao J, Chen Y, Xiao Q, Shi Y. 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. 2016. 531−544.

[6] Padioleau Y, Lawall J, Hansen RR, Muller G. Documenting and automating collateral evolutions in Linux device drivers. ACM SIGOPS Operating Systems Review, 2008, 42(4): 247−260.

[7] Ngiam J, Khosla A, Kim M, Nam J, Lee H, Ng AY. Multimodal deep learning. In: Proc. of the ICML. 2011.

[8] Ramachandram D, Taylor GW. Deep multimodal learning: A survey on recent advances and trends. IEEE Signal Processing Magazine, 2017, 34(6): 96−108.

[9] He J, Zhang CQ, Li XZ, Zhang DH. Survey of research on multimodal fusion technology for deep learning. Computer Engineering, 2020, 46(5): 1−11 (in Chinese with English abstract).

[10] McKenney PE. Overview of linux-kernel reference counting. 2007. https://www.open-std.org/JTC1/SC22/WG21/docs/papers/2007/n2167.pdf

[11] Bovet DP, Cesati M. Understanding the Linux Kernel. O'Reilly Media, 2005.

[12] Atomic type documentation. https://www.kernel.org/doc/Documentation/atomic_t.txt

[13] Kref type documentation. https://www.kernel.org/doc/Documentation/kref.txt

[14] Refcount_t type documentation. https://www.kernel.org/doc/Documentation/core-api/refcount-vs-atomic.rst

[15] https://lwn.net/Articles/668724/

[16] Refcount operation documentation. https://www.kernel.org/doc/Documentation/driver-api/basics.rst

[17] Collins M, Singer Y. Unsupervised models for named entity classification. In: Proc. of the Joint SIGDAT Conf. on Empirical Methods in Natural Language Processing and Very Large Corpora. 1999.

[18] Szarvas G, Farkas R, Kocsor A. A multilingual named entity recognition system using boosting and C4.5 decision tree learning algorithms. In: Proc. of the Int'l Conf. on Discovery Science. 2006. 267−278.

[19] McNamee P, Mayfield J. Entity extraction without language-specific resources. In: Proc. of the 6th Conf. on Natural Language Learning. 2002.

[20] Krishnan V, Manning CD. An effective two-stage model for exploiting non-local dependencies in named entity recognition. In: Proc. of the 21st Int'l Conf. on Computational Linguistics and the 44th Annual Meeting of the Association for Computational Linguistics. 2006. 1121−1128.

[21] Collobert R, Weston J, Bottou L, Karlen M, Kavukcuoglu K, Kuksa P. Natural language processing (almost) from scratch. Journal of Machine Learning Research, 2011, 12: 2493−2537.

[22] Huang Z, Xu W, Yu K. Bidirectional LSTM-CRF models for sequence tagging. arXiv preprint arXiv: 1508.01991, 2015.

[23] Hinton GE, Salakhutdinov RR. Reducing the dimensionality of data with neural networks. Science, 2006, 313(5786): 504−507.

[24] Kahou SE, Pal C, Bouthillier X, *et al*. Combining modality specific deep neural networks for emotion recognition in video. In: Proc. of the 15th ACM on Int'l Conf. on Multimodal Interaction. 2013. 543−550.

[25] Wu D, Pigou L, Kindermans PJ, Le NDH, Shao L, Dambre J, Odobez JM. Deep dynamic neural networks for multimodal gesture segmentation and recognition. IEEE Trans. on Pattern Analysis and Machine Intelligence, 2016, 38(8): 1583−1597.

[26] Hochreiter S, Schmidhuber J. Long short-term memory. Neural Computation, 1997, 9(8): 1735−1780.

[27] Graves A, Schmidhuber J. Framewise phoneme classification with bidirectional LSTM and other neural network architectures. Neural Networks, 2005, 18(5-6): 602−610.

[28] LeCun Y, Boser B, Denker J, Henderson D, Howard R, Hubbard W, Jackel L. Handwritten digit recognition with a back-propagation network. In: Advances in Neural Information Processing Systems. 1989.

[29] LeCun Y, Bottou L, Bengio Y, Haffner P. Gradient-based learning applied to document recognition. Proc. of the IEEE, 1998, 2278−2324.

[30] Peters ME, Neumann M, Iyyer M, Gardner M, Clark C, Lee K, Zettlemoyer L. Deep contextualized word representations. arXiv preprint arXiv: 1802.05365, 2018.

[31] Galassi A, Lippi M, Torroni P. Attention in natural language processing. IEEE Trans. on Neural Networks and Learning Systems, 2021, 32(10): 4291−4308. [doi: 10.1109/TNNLS.2020.3019893]

[32] Vaswani A, Shazeer N, Parmar N, Uszkoreit J, Jones L, Gomez AN, Polosukhin I. Attention is all you need. In: Advances in Neural Information Processing Systems. 2017. 5998−6008.

附中文参考文献:

[2] 冯震, 聂森, 王轶骏, 薛质. 基于 S2E 的 Use-after-free 漏洞检测方案. 计算机应用与软件, 2016, 33(4): 273−276.

[3] 张静, 黄志球, 沈国华, 喻垚慎, 艾磊. C 程序中的内存泄漏机制分析与检测方法设计. 计算机工程与科学, 2020, 42(5): 776−787.

[9] 何俊, 张彩庆, 李小珍, 张德海. 面向深度学习的多模态融合技术研究综述. 计算机工程, 2020, 46(5): 1−11.

谈心(1996－), 男, 博士生, 主要研究领域为漏洞挖掘, 软件补丁研究.

曹家俊(1998－), 男, 硕士生, 主要研究领域为软件补丁研究, 软件安全性.

杨悉瑜(1997－), 女, 硕士生, 主要研究领域为静态程序分析, 内核缺陷检测.

张源(1987－), 男, 博士, 副教授, 博士生导师, CCF 专业会员, 主要研究领域为软件安全, 程序分析.