一种语义感知的安全检查检测方法

方宇彤, 刘洪毅, 李经纬, 文伟平

北京大学软件与微电子学院 北京 中国 102600

摘要 安全检查是 OS 中最常见的安全增强方式, 是漏洞检测的重要基石。检测安全检查必然要用到异常处理函数, 本论文从检测异常处理函数入手, 实现了基于异常处理函数的安全检查检测方法。论文提出了一种语义感知的安全检查检测方法(Sased), 通过基于自然语义、程序语义的异常处理函数检测方法对 Linux 系统的函数控制流进行静态分析。同时, Sased 可以对异常处理函数和安全检查进行回溯过滤从而降低其误报率。通过实验, Sased 共报告异常处理函数 795 个, 安全检查 41519 个, 二者都达到了 90%以上的检测准确率。这其中, 有 208 个异常处理函数是之前的工作中从未发现的。同时,我们结合已有的漏洞检测方法, 发现了 Linux 内核的 9 个新漏洞。实验表明 Sased 可以非常高效地检测 OS 中的异常处理函数及安全检查, 从而为操作系统漏洞检测提供有益的参考。

关键词 异常处理; 安全检查; 系统安全
中图法分类号 TP319 **DOI 号** 10.19363/J.cnki.cn10-1380/tn.2020.09.02

Semantic-aware Security Check Detection Method

FANG Yutong, LIU Hongyi, LI Jingwei, WEN Weiping

School of Software and Microelectronics, Peking University, Beijing 102600, China

Abstract The security check is the most common security enhancement method in OS, and it is an essential cornerstone of vulnerability detection. Detecting security checks has to use the functions of exception handling. This paper starts with the detection of the exception handling function and implements a method of security check detection based on the exception handling function. This paper proposes a Semantic-aware Security Check Detection Method (Sased), which performs static analysis on the function control flow of the Linux system via an exception handling function detection method based on natural and program semantics. At the same time, Sased can retrospectively filter exception handling functions and security checks to reduce its false-positive rate. Through experiments, Sased reports a total of 795 exception handling functions and 41519 security checks, both of which have achieved a detection accuracy of more than 90%. Among them, 208 exception handling functions have never been found in previous work. At the same time, we combined the existing vulnerability detection methods and found 9 new vulnerabilities in the Linux kernel. Experiments show that Sased can detect exception handling functions and security checks in the OS very efficiently, thus providing a useful reference for operating system vulnerability detection.

Key words exception handling; security check; system security

1 介绍

操作系统是管理计算机硬件与软件资源的计算机程序, 其功能复杂, 体型庞大, 仅 Linux 内核就有 25 000 000 以上的代码行数。由于其复杂的逻辑, 操作系统经常可能发生错误等异常情况, 例如来自软件的意外行为、操作者的不正常输入、硬件的物理故障等。以上都是可以被视为异常的行为, 如果操作系统没有强大的异常恢复能力或异常处理能力, 其就会在运行的过程中受到恶意代码的攻击(例如, 栈溢出等)或者产生致命的错误(例如, 系统崩溃等)。因此, 良好的异常处理能力, 是操作系统健壮性的体现, 是系统可靠性的基石[1]。

异常处理是系统进行安全检查后需要执行的一部分路径, 而安全检查是操作系统中最常见的安全增强方式, 其通过对于临界变量的检查来确认系统目前的运行状态, 以确保系统的安全。良好的安全检查设计可以使得我们发现程序中的错误, 并对其进行相应的处理。经过数十年来计算机科学的发展, 操作系统中有了很多安全检查。例如, 指

通讯作者: 文伟平, 博士, 教授, Email:weipingwen@ss.pku.edu.cn。
收稿日期: 2019-11-30; 修改日期: 2020-02-17; 定稿日期: 2020-08-25

针使用前应先确定其不为空，表示字符串长度的变量不能被篡改等。而当临界变量出现可能的错误或者需要遵守的约束条件不能满足的时候，系统将执行异常处理函数，对当前程序执行进行处理或修正。为了更好地对异常处理函数、安全检查等进行解释，我们给出如下说明:

异常处理函数: 当程序运行出现异常情况，为了保持正常运行或系统不受破坏而执行的处理异常情况的函数(例如，图 1 的 27 行)。

安全检查: 检查系统会发生哪些特定错误或使用变量或函数时应该强制执行的约束条件，通常是条件语句(例如，图 1 的 26 行)。

临界变量(critical variables): 代表系统运行状态的变量，通常是安全检查中检查的变量(如图 1 中的 result)。

```
20 //Linux 5.4: /drivers/net/caif/caif_serial.c
21 static int register_ldisc(void)
22 {
23     int result;
24
25     result = tty_register_ldisc(N_CAIF, &caif_ldisc);
26     if (result < 0) {
27         pr_err("cannot register CAIF ldisc=%d err=%d\n", N_CAIF,
28             result);
29         return result;
30     }
31     return result;
32 }
```

图 1　异常处理函数、安全检查、和临界变量的实例

Figure 1　Examples of exception handlers, security checks, and critical variables

在理想情况下，每一个临界变量的每一次使用都应该进行相应的安全检查。但实际上，由于操作系统的代码量庞大，结构复杂，或是因为开发者的疏忽，开发者很难全部确定自己使用的变量是否需要安全检查。因而安全规则并不能被很好地遵守，因此可能危害整个操作系统的安全性和完整性[2]。

现有的安全检查机制并不能完美的解决系统中的存在的问题，安全检查的缺失或错误一直是操作系统中的十分普遍的情况。这种安全检查的错误包含两种情况，(1)一种是对于某个规则，系统没有为其设置安全检查，如图 2 所示; (2)另一种情况是，对于某个规则，程序中存在安全检查，但是不完全，存在漏查的问题。缺少安全检查的错误可能会导致各种各样严重的安全后果，这其中包括越界访问、信息泄露、权限绕过，甚至整个系统崩溃。

不幸的是，即使缺少安全检查的问题如此普遍和严重，但是针对安全检查的研究项目却很少，出现这样的情况的内在原因可能是: (1)关于安全检查的系统漏洞还未受到人们的注意; (2)操作系统源代码体量巨大，调用关系复杂，详细分析源码是不可能完成的任务; (3)目前针对识别异常处理的研究也尚少。针对以上三点，我们对 Linux 操作系统的异常处理情况展开深入研究，以便通过异常处理实现对安全检查的精确识别。

```
1  // Linux 5.2.13: /net/smc/smc_ib.c
2  static void smc_ib_remove_dev(struct ib_device *ibdev, void *client_data)
3  {
4      struct smc_ib_device *smcibdev;
5
6      smcibdev = ib_get_client_data(ibdev, &smc_ib_client);
7      ib_set_client_data(ibdev, &smc_ib_client, NULL);
8      spin_lock(&smc_ib_devices.lock);
9      list_del_init(&smcibdev->list); /* remove from smc_ib_devices */
10     spin_unlock(&smc_ib_devices.lock);
11     smc_ib_cleanup_per_ibdev(smcibdev);
12     ib_unregister_event_handler(&smcibdev->event_handler);
13     kfree(smcibdev);
14 }
```

图 2　缺失安全检查的实例

Figure 2　Examples of missing security checks

(注: *smcibdev* 变量在第 9 行进行调用的时候没有检查它是否为空)

Linux 内核结合了大量的安全检查来验证变量和函数的返回值[3]。安全检查通常包括检查条件和执行路径两个组成部分。执行路径会分为异常处理执行路径和正常执行路径。安全检查在检查条件失败的时候，执行异常处理路径的检查程序，如图 1 所示。Linux 内核中，在条件语句发生错误时，异常处理路径将调用一个异常处理函数(图 1 的 26 行)对当前状态进行修正或处理。由此可见，异常处理函数的检测可以为安全检查的识别提供依据。

对于安全检查的识别，一个非常直观的认识是，安全检查机制通常包含异常处理函数(例如，异常报告函数)，所以可以通过全面检测异常处理函数对安全检查进行初步识别，再根据安全检查通用的特性对其进行精确识别。据我们所知，之前针对异常处理函数的检测，主要分为两种方法: (1)人工识别: 这种方法可以针对操作系统内核，但费时费力，并且会出现比较严重的遗漏; (2)静态分析: 这种方法实现了自动化，但是缺点是现有基于数据流的异常处理函数研究都是关于具体应用程序的，其为了提高检测精度而牺牲了分析范围，无法适用于庞大复杂的操作系统内核。

在本文中，之所以对异常处理函数的检测展开研究，是因为:

(1) 绝大多数的异常处理函数都用于系统的安全检查，可以通过对异常处理路径和正常处理路径的检测实现对安全检查的识别。

(2) 异常处理函数的精确检测是一个巨大挑战，主要表现在，系统中有标准的异常处理函数

(如 BUG()、panic()等), 也有越来越多的自定义的异常处理函数, 然而没有一个官方的手册对 Linux 中的异常处理函数进行说明, 之前也没有专门的工作针对异常处理函数的检测。

安全检查的识别的思想非常直观, 但是它的实现仍然需要克服多种挑战:

(1) 如何获取尽量全面的异常处理函数。只有获取尽量全面的异常处理函数, 才能找到更多的安全检查。找到全面的异常处理函数是一个前提条件。

(2) 如何降低异常处理函数的误报率。作为识别安全检查的基石, 其本身应该具有较高的可信度。人工校验去除误报是一个费时费力的方法, 应该将人工校验的流程提取出来, 实现自动化。

(3) 如何对安全检查进行精确的识别, 如何优化识别结果也是一个挑战。

为了解决以上挑战, 我们提出了一种语义感知的安全检查检测方法(Sased), 其具体的技术分别解决以上几个挑战。为了解决挑战(1), 我们提出了一种基于自然语义的可疑异常处理函数识别方法。它的主要思想是, 通过对异常处理函数名进行分词, 与设定的负向词库进行比对, 来初步确定异常处理函数样本集。我们还提出了一种基于程序语义的异常处理函数样本集扩充方法, 通过判断异常处理函数的父子函数的传参类型以及对象, 进一步对样本集进行扩充。为了解决挑战(2), 我们提出了回溯过滤技术来自动化确认异常处理函数。为了解决挑战(3), 我们提出了基于异常处理函数的安全检查识别方法, 根据安全检查的组成特性, 对操作系统内核中的安全检查进行了较为精确的识别。

Sased 结合自然语义感知和程序语义感知的异常处理函数检测方法以及回溯过滤方法, 实现了对操作系统中异常处理函数的精确检测, 并根据异常处理函数对安全检查进行了有效的识别。

同时 Sased 也具有很高的可扩展性, 能够轻松添加新的检测规则, 从而实现对异常处理函数和安全检查样本集的有效扩充。此外, 在后边的章节中, 我们讨论了 Sased 的可移植性, Sased 可以比较容易地扩展到检测其他系统软件和程序的异常处理函数检测当中。最后, 我们在 Linux5.5-rc6 中找到了 9 个安全检查缺失漏洞。

我们在本文做出了以下贡献:

(1) 我们提出了一种异常处理函数和安全检查检测方法。一种语义感知的安全检查检测方法, 通过这种方法来对操作系统内核中的异常处理函数和安全检查进行详细而精确的探测, 为漏洞检测提供依据。由此产生的系统 Sased 是开源的(项目链接: https://github.com/lhysgithub/Sased)。

(2) 我们在 Sased 中提出了几种新的技术, 这将有利于其他的研究项目。尤其是回溯过滤异常处理函数和安全检查的方法, 这将大大减少类似检测系统的误报率。

(3) 使用 Sased, 我们检测到 Linux 内核中存在的大量异常处理函数和安全检查, 扩充了现有的异常处理函数集和安全检查集, 并找到了 9 个安全检查缺失漏洞, 为漏洞检测、漏洞修补提供了更多的参考。

本文的其余部分安排如下:

本文的第二章将介绍本文的背景和动机; 第三章介绍异常处理函数检测、安全检查检测、以及基于其的漏洞检测的相关工作; 第四章将从较高层次上介绍 Sased 的组成结构; 第五章将分别介绍文章提出的几种新的技术, 以及 Sased 的详细设计; 第六章将介绍 Sased 的实验过程及其发现的安全检查缺失漏洞; 第七章中进一步讨论 Sased 以及未来的发展方向,并对全文进行总结。

2 背景和动机

2.1 严重的安全影响

我们可以知道, 安全检查的检测对于漏洞发现、漏洞挖掘具有非常重大的意义, 而操作系统程序中缺少安全检查或者安全检查不完全, 则极有可能造成非常严重的后果, 甚至导致系统崩溃。而异常处理作为检测安全检查的最重要的指标, 对系统安全的影响也是至关重要的。

为了调查说明安全检查的重大影响, 我们在文献[4]的基础上, 在以下几个方面进行了补充调研: (1)有多少安全漏洞是由缺少安全检查引起的? (2)这些安全漏洞的严重安全影响。

针对第一个问题, 我们在美国国家漏洞数据库[5]中, 从 2017 到 2019 年的安全漏洞中随机抽取了 300 个进行调查。在这些漏洞中, 我们选择了通过添加安全检查进行修复的漏洞作为缺少安全检查漏洞。统计结果表明, 共有 184 个漏洞是关于安全检查的漏洞。在 2017 年和 2018 年, 大多数(59.5%)的安全漏洞都是与缺少安全检查相关的, 而在 2019 年, 这个比例高达 65%。

针对第二个问题, 通过调查得知, 2017 年到 2018 年最严重的的漏洞, 也就是 CVSS(通用漏洞

评分系统)得分为 10(最高严重级别)的漏洞中, 有 11 个是关于缺少安全检查所引起的。而据我们的统计, 在 2019 年, 由我们抽样得到的 65 个安全检查漏洞中, 有 7 个漏洞的 CVSS 被认定为 Critical, 有 28 个被认定为 High, 也就是说, 大约 53.9%的缺少安全检查漏洞是比较严重的。

这些安全漏洞中缺少检查的目标包括返回值大小和类型, 权限、指针以及缓冲区的长度, 相应的, 缺少的检查会导致严重的不良影响, 包括内存损坏、信息泄露、缓冲区溢出, 最终甚至会导致整个系统的崩溃。

2.2 缺少相关研究工作

异常处理函数通常分为两种, 一种是带有异常返回代码的异常处理函数, 这种函数比较容易被检测到, 另一种是不带返回值的异常处理函数。目前对于这种类型的异常处理函数的研究很少。

而在安全检查方面, 据我们所知, 这方面的研究工作主要在 Kangjie Lu 的 CRIX[4]中有所体现。CRIX 在源码层根据可疑函数名或参数名来识别可疑异常报告函数, 并通过人工确认找到的可疑函数是否为异常处理函数, 据此识别安全检查和安全临界变量。但是我们发现, 仅通过函数名和参数名对异常处理函数进行识别会出现大量的漏报, 例如图 3 所示的, CRIX 将 AA_ERROR 识别为异常处理函数, 而第 14 行中的 pr_err_ratelimited()函数按照语义, 它实际上也是异常处理函数, 却因为扫描对象的粗糙而被漏报, 由此可知, 仅通过函数名称对异常处理函数进行识别是不完善的。即使这种识别方法存在一定的误报和漏报, 但仍然对于基于安全检查的漏洞挖掘提供了非常多的支持。

```
1  // Linux 5.2.13: /security/apparmor/include/lib.h
2  #define AA_ERROR(fmt, args...)                  \
3      pr_err_ratelimited("AppArmor: " fmt, ##args)
4
5  // Linux 5.2.13: /security/apparmor/apparmorfs.c aa_create_aafs()
6  if (aa_sfs_entry.dentry) { // Father function is a error handler
7      AA_ERROR("%s: AppArmor securityfs already exists\n", __func__);
8      return -EEXIST;
9  }
10 // Linux 5.2.13: /arch/s390/kvm/kvm-s390.c kvm_arch_vcpu_ioctl_run()
11 if (!kvm_s390_user_cpu_state_ctrl(vcpu->kvm)) { // Child function is also a error handler
12     kvm_s390_vcpu_start(vcpu);
13 } else if (is_vcpu_stopped(vcpu)) {
14     pr_err_ratelimited("can't run stopped vcpu %d\n",vcpu->vcpu_id);
15     rc = -EINVAL;
16     goto out;
17 }
```

图 3 CRIX 对于一些异常处理函数的漏报

Figure 3 CRIX's Missing case for exception handling functions

作为基于安全检查的漏洞挖掘的基石, 异常处理函数得到的数据支持越多, 其挖掘效果将会越好, 而现有的工作, 在对于异常函数识别和安全检查识别方面的不足, 是我们进行研究的最大的动机。

3 相关工作

异常处理和安全检查在计算机领域是非常普遍的现象, 但是只有很少的研究工作着眼于操作系统内核, 这可能是由于操作系统内核过于庞杂, 很难理清内部的逻辑。大部分工作都对分析范围进行了限制, 以提高分析的准确性, 减少复杂度。例如:

APEx[6]总结出调用函数中的错误处理代码比实现正常功能的代码更有可能具有更少的分支点、程序语句和函数调用的思想, 根据这样的路径特征对异常处理代码进行识别, 完成了对 C 语言 API 的检测。

Rubio-González 等人[7]和 Gunawi 等人[8]创建的用于静态检测 Linux 文件系统代码中异常处理错误的查找程序; Lawall 等人[9]构建并评估了一个用于在安全套接字层(SSL)中查找异常处理错误的静态异常查找工具, 他们使用 Coccinelle(一个程序匹配和转换引擎)查找 OpenSSL 中缺少的安全检查; Weimer 等人[10-11]开发的用于查找 Java 程序中异常处理错误的 bug 查找工具; Robillard 等人[12]构建的简化和可视化异常处理流的工具, 可以帮助开发人员最小化异常处理代码中的错误的发生; Acharya 等人[13]通过挖掘 API 运行时行为的静态跟踪, 自动推断 API 的错误处理规范。正如 Acharya 等人所观察到的, 缺乏广泛的数据依赖性支持(例如指针分析、别名等)会导致结果不精确; EPEx[14]实现了自动检测 C 语言程序中的不同类型异常处理函数发生的错误; Gunawi 和 Rubio Gonzalez 等人[7-8,14-15]使用静态数据流分析在文件和存储系统中发现了异常处理函数检测缺失和错误,他们的方法使用符号执行来检测异常处理路径, 然后通过检查是否传播了适当的错误值来检测安全检查缺失错误; Son S[16]开发了一种健壮的方法来查找 Web 应用程序中丢失的安全检查。

以上都是针对具体应用程序范围内的对安全检查错误, 对异常处理函数的检测方法, 接下来一部分是在操作系统内核范围内的相关工作内容。

K-Miner[17]是一个操作系统内核的静态分析框架, 其核心思想是以系统调用接口为起点, 沿着不同的执行路径对内核代码进行分区。可以允许在复杂的操作系统内核中进行实际的过程间数据流分析。DR. CHECKER[18]是一个用于识别 Linux 内核

驱动程序错误的工具, 它能够有效地检测驱动程序中漏洞的分类, 但是在某些情况下, DR. CHECKER 选择以可靠性为代价(放弃跟踪核心内核代码的调用)来保证分析的精确度。以上两种方法都采用的传统的静态检测技术。

Kint[19]使用污点分析来查找 Linux 内核中的整数错误。UniSan[20]使用静态污点分析来搜索可能的从内核空间复制到用户空间的未初始化字节。

Lrsan[21]结合了自动安全检查识别和临界变量推理等新的分析技术, 实现了对操作系统内核中的缺失再检查(lacking recheck, LRC)漏洞的检测。

Kangjie Lu 的 CRIX[4]在 Linux 内核源码级别根据函数名来识别异常处理函数, 据此识别安全检查和安全临界变量, 根据条件语句构造约束条件, 并根据语义(条件类型等)进行建模, 从而检测安全检查缺失错误。

CHEQ[22]采用和 CRIX[4]中一致的异常处理函数和安全检查检测方法, 对 Linux 内核中的异常处理情况和安全检查进行了一系列形式化定义, 并对空指针解引用、缺失异常处理、和双重获取漏洞, 三种关键的内核语义错误进行了检测。

Jia-Ju Bai 等人[23]提出了一种错误注入测试方法 EH-Test, 它设计了一种基于模式的异常处理代码特征的提取策略, 并根据异常处理代码特性对 Linux 设备驱动程序中的异常处理代码进行了有效测试。EH-Test 在 15 个 Linux 设备驱动程序中发现了 50 个新的漏洞。

文献[4,22-23]都是基于异常处理以及安全检查来实现对操作系统内核错误的检测的, 由此可见, 对于系统内异常处理情况和安全检查的识别对于漏洞检测具有重大意义, 而此前并没有专门的工作是针对异常处理函数的识别和检测的。

此外, 相关工作中对于安全检查以及异常处理函数检测方法的单一性, 导致结果中存在大量的漏报和误报, 所以提高异常处理函数和安全检查的检测精度, 将是本文关注的重点内容。

4 概述

本章节主要通过介绍 Sased 的顶层设计, 概述如何检测操作系统中的异常处理函数以及安全检查, 为漏洞检测、漏洞修补提供更多的参考。

Sased 的目标是检测 Linux 内核中的异常处理函数和安全检查。为了做到这一点, 首先需要讨论几个问题: (1)安全检查的目标是什么? (2)安全检查由什么组成? (3)异常处理函数如何进行识别?

安全检查一般是一个分支语句, 其由三部分组成: (1)安全检查条件; (2)异常处理执行路径; (3)正常执行路径。异常处理路径往往调用了一个异常处理函数, 这个异常处理函数分为两种, 一种是遵循 errno.h 的带有异常返回代码的异常处理函数, 这类异常往往比较严重, 但却很容易被检测; 第二种就是不带返回值的异常处理函数, 其中异常报告函数居多。

异常处理函数作为基于安全检查的漏洞挖掘方法的基石, 安全检查的数据支持越多, 漏洞检测效果必然越好。本着这样的思想, 我们对异常处理函数的识别进行了更深的研究。

现有的对于异常处理函数和安全检查检测的方法存在一些问题:

(1) 通过检测可疑关键字的方式筛选可疑函数, 其对于可疑关键字的定义具有相当严重的依赖性。可疑关键字集设置的好坏直接决定了异常处理函数集的上限, 这种设置要求安全专家对于 Linux 内核异常处理有着极深的理解。

(2) 安全检查的确定方式一般是先确定异常处理函数, 再根据安全检查的特性去识别安全检查。但是, 即便可以通过函数名的可疑关键字帮助人们进行可疑函数的筛选, 人工的方式去确认异常处理函数仍是一件费时费力的方法。

为了解决问题(1), 我们提出了基于自然语义的异常函数检测技术。程序会对异常处理函数名进行分词, 并根据负向词分词得到新的可疑关键字集, 实现智能化的可疑关键字集构建, 并找到更多的异常处理函数。

为了解决问题(2), 我们提出了回溯过滤技术来自动化地确认异常处理函数。对于每一个可疑函数, 查看其调用环境是否在一个分支当中, 并且部分分支没有调用可疑函数, 这样便认定一个可疑函数为一个异常处理函数。

此外, 通过我们对于 Linux 内核源码的分析, 异常处理函数的父子函数仍有可能为异常处理函数, 并且传参类型基本相同, 如图 5 所示, 父函数 acpi_print_osc_error 调用子函数 acpi_handle_debug, 父子函数又分别当做异常处理函数使用。为了尽可能多地发现可用的异常处理函数, 我们提出了基于程序语义的异常处理函数检测技术。其将确认的异常处理函数的父子函数作为可疑函数, 通过参数类型和对象检测其是否也为异常处理函数。

最后即为基于异常处理函数的安全检查识别。其根据安全检查的组成特性, 全局检测安全检查

语句。

总的来说，Sased 的主要由四个部分组成，如图 4 所示，基于自然语义的异常函数检测模块自动学习可疑关键字集并生成可疑函数集，基于程序语义的异常函数检测模块可以找到更多类似的异常处理函数，回溯过滤模块可以精简并确认异常函数集，最后基于异常处理函数的安全检查模块通过安全检查的特性识别安全检查。

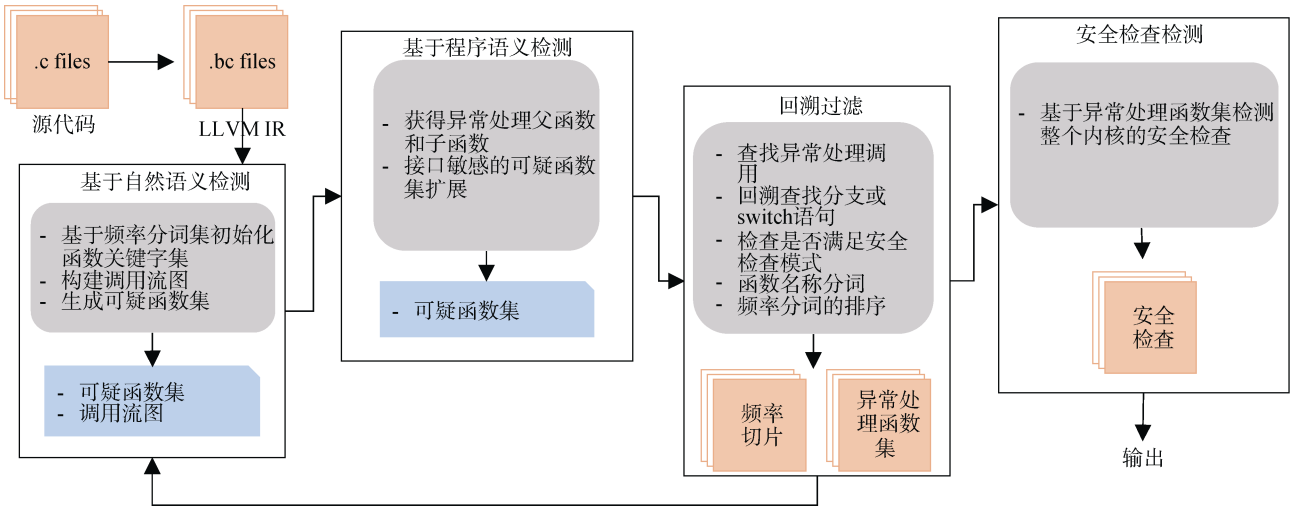

图 4　Sased 框架图

Figure 4　The overview of Sased

```c
// Linux 5.2.10: /drivers/acpi/bus.c
static void acpi_print_osc_error(acpi_handle handle,
                struct acpi_osc_context *context, char *error)
{
    int i;

    acpi_handle_debug(handle, "(%s): %s\n", context->uuid_str, error);

    pr_debug("_OSC request data:");
    for (i = 0; i < context->cap.length; i += sizeof(u32))
        pr_debug(" %x", *((u32 *)(context->cap.pointer + i)));

    pr_debug("\n");
}

// Linux 5.2.10: /drivers/acpi/bus.c
    /* Need to ignore the bit0 in result code */
    errors = *((u32 *)out_obj->buffer.pointer) & ~(1 << 0);
    if (errors) {
        if (errors & OSC_REQUEST_ERROR)
            acpi_print_osc_error(handle, context,
                "_OSC request failed");
        if (errors & OSC_INVALID_UUID_ERROR)
            acpi_print_osc_error(handle, context,
                "_OSC invalid UUID");
        if (errors & OSC_INVALID_REVISION_ERROR)
            acpi_print_osc_error(handle, context,
                "_OSC invalid revision");
        if (errors & OSC_CAPABILITIES_MASK_ERROR) {
            if (((u32 *)context->cap.pointer)[OSC_QUERY_DWORD]
                & OSC_QUERY_ENABLE)
                goto out_success;
            status = AE_SUPPORT;
            goto out_kfree;
        }
        status = AE_ERROR;

// Linux 5.2.10: /drivers/acpi/bus.c
int acpi_bus_get_private_data(acpi_handle handle, void **data)
{
    acpi_status status;

    if (!*data)
        return -EINVAL;

    status = acpi_get_data(handle, acpi_bus_private_data_handler, data);
    if (ACPI_FAILURE(status)) {
        acpi_handle_debug(handle, "No context for object\n");
        return -ENODEV;
    }

    return 0;
}
```

图 5　父子函数均为异常处理函数

Figure 5　Parent and child functions both are exception handling functions

5　设计

Sased 是一个基于异常处理函数的安全检查识别方法，其主要通过增强异常处理函数的识别来增强安全检查检测的效果。识别异常处理函数的一般策略是，首先找到可能是异常处理函数的可疑函数，再通过异常处理函数的特性来进行确认。Sased 首先采用基于自然语义和程序语义的异常处理函数检测方法来找到可疑函数，再通过对异常处理函数的回溯过滤方法去掉误报项，最终通过识别异常处理函数的所有安全检查模式实现基于异常处理函数的安全检查检测方法。

5.1　自然语义感知的异常处理函数检测

由于可疑关键字集很难人为去定义完善，因此仅通过函数名来识别可疑函数容易出现大量误报和漏报，为了解决这个问题，我们将能确定的异常处理函数的函数名进行分词，根据函数名片段在异常处理函数集中的频率选取异常处理函数特征字段，进而实现可疑关键字集的自动生成。对于如何确定一个可疑函数是否是异常处理函数，我们将在 5.3 中描述。为了实现上述想法，如何分词，以及如何选择分词片段将成为主要问题。

根据我们对于 Linux 内核的观察与实验，Linux 内核函数大多使用“_”来分割一个函数名内的单

词或单词缩写。因此针对 Linux 内核, 我们指定的分词分割方式是通过“_”进行分割。根据实验的结果, 上述分词方式可能会产生大量无意义却频率很高的片段, 这是由于函数命名中常用一些通用的单词或单词缩写(例如,“to”、“get”等)。根据我们的观察, 异常处理函数往往带有负向语义(例如,“error”、“remove”等)的片段, 因此在选择分词片段时, 应尽量选取带有负向语义的高频词汇。为了设定何为高频片段, 我们设定了一个频率阈值。

$$\theta = P / S \tag{1}$$

其中, S 为异常处理函数集的大小, P 为包含该片段的异常处理函数的数目。频率阈值对于整个实验效果的影响将在 6.2 部分讨论。

还有一个比较重要的问题是, 第一批异常处理函数如何获取? 我们可以参考以往的方式, 首先预定义一个可疑关键字集(例如, “err”、“warn”等)来筛选可疑函数集, 再通过 5.3 的方法筛掉非异常处理函数, 得到第一批异常处理函数。

自然语义感知的异常处理函数检测流程是, (1)由已确定的异常处理函数集, 根据片段的频率阈值生成可疑关键字集, 或直接初始化可疑关键字集; (2)根据可疑关键字集筛选所有函数, 以获得可疑函数集。如何确认一个可疑函数为异常处理函数的方法, 将在 5.3 中讨论。

5.2 程序语义感知的异常处理函数检测

自然语义感知的异常处理函数检测方法存在着一个缺陷, 即无法检测函数名中不包含负向片段的异常处理函数。根据我们对于 Linux 内核的观察, 一个异常处理函数的父函数与子函数都有可能还是异常处理函数, 即可能存在一个异常处理函数通过另一个异常处理函数实现的情况, 父异常处理函数仅对子异常处理函数进行参数上的传递。基于以上经验, 我们提出了一种基于程序语义的异常处理函数扩展检测方法。规定若一个异常处理函数的接口(即参数类型与参数名)与其调用的子函数相似, 则认为其子函数为可疑函数; 同理, 若一个异常处理函数的接口与其父函数相似, 则认为其父函数也为可疑函数。

需要注意的是, 对于异常处理函数的父子函数扩展问题上, 一定是对于确定的异常处理函数。可疑函数的父子函数不应该被认定为可疑函数。

5.3 异常处理函数的回溯过滤方法

在先前的工作中, 需要人工来对可疑函数集进行确认, 为了解决这个问题, 我们提取人工识别异常处理函数的流程, 设计了异常处理函数的回溯过滤方法, 以降低最后的误报率。该方法的核心思想是, 异常处理函数的确认与安全检查相结合, 即确定可疑函数的调用情况, 分析其调用环境是否用于安全检查中。本节将首先给异常处理函数的回溯过滤方法的形式化定义, 再给出其设计的相关细节。

5.3.1 形式化定义

本节首先将异常处理函数以及安全检查的模式进行形式化定义。设语句为 S, 分支语句为 BS, 安全检查语句为 SC, 异常处理函数为 e, 可疑函数为 s, 函数 $\delta(BS)$表示分支语句 BS 所能达到的分支, 分支 P 由异常处理分支 EP 和正常分支 NP 组成, 即 $P \in \{EP, NP\}$, $\phi(P)$表示分支中的所有语句。若 $e \in \phi(P)$则该分支 P 为异常处理分支 EP, 相反, 若 $e \notin \varphi(P)$则该 P 为 NP。若分支语句 BS 为安全检查, 则一定存在部分 BS 能到达的分支为异常处理分支 EP, 同时部分 BS 能到达的分支为正常分支 NP。我们称这种模式是安全检查模式, 如公式(2)所示。并称满足安全检查模式的异常处理函数 e 满足安全检查特性, 即 e 服从安全检查模式 $e \sim M(SC)$, 如公式(3)所示。

$$BS = SC \Leftrightarrow \exists EP \in \delta(BS) \wedge \exists NP \in \delta(BS) \tag{2}$$

$$(\exists e \in \phi(P_1) \wedge P_1 \in \delta(SC)) \wedge (\forall e \notin \phi(P_2) \wedge P_2 \in \delta(SC)) \Leftrightarrow e \sim M(SC) \tag{3}$$

同时, 反过来讲, 若有部分分支为异常分支 EP, 同时部分分支为正常分支 NP, 则有该分支语句 BS 为安全检查语句 SC, 如公式(2)所示, 该公式从右往左看可以用来判断一个分支语句是否为安全检查语句。

值得注意的是, 由于我们在一开始不能确认哪条分支语句是安全检查语句, 以及哪个可疑函数是异常处理函数。因此我们不能识别形如公式(3)的安全检查模式。因此我们定义一种“类安全检查模式”, 使得安全检查模式是类安全检查模式的一个子集。

类安全检查模式: 若语句 S 为分支语句, 并且部分分支包含可疑函数 s, 部分分支不包含可疑函数 s, 则可疑函数 s 服从类安全检查模式 $s \sim M(BS)$, 如公式(4)所示。

$$(\exists s \in \phi(P_1) \wedge P_1 \in \delta(BS)) \wedge (\forall s \notin \phi(P_2) \wedge P_2 \in \delta(BS)) \Leftrightarrow s \sim M(BS) \tag{4}$$

若一个可疑函数 s, 不满足类安全检查模式, 其不满足安全检查模式, 即一定不是异常处理函

数, 如公式(5)所示; 若一个可疑函数 s, 满足类安全检查模式, 其不一定是异常处理函数。

$$\neg((\exists s \in \phi(P_1) \wedge P_1 \in \delta(S)) \wedge (\forall s \notin \phi(P_2) \wedge P_2 \in \delta(S))) \Rightarrow S \neq SC \quad (5)$$

公式(5)不能用来确定一个可疑函数是异常处理函数, 但可以用来确定一个可疑函数不是异常处理函数, 即通过公式(5)我们可以筛掉很多误报。

5.3.2 回溯过滤异常处理函数

一个安全检查模式如图 6 所示: 第 3 行调用了可疑函数 pr_err(), 回溯到可疑函数的上一层条件语句, 对第 2 行所示的条件语句分支进行检查, 分析其两条分支分别调用可疑函数 pr_err(); 和正常函数 omap1_clk_disable();满足类安全检查模式,则可疑函数 pr_err()有更高的概率为异常处理函数。

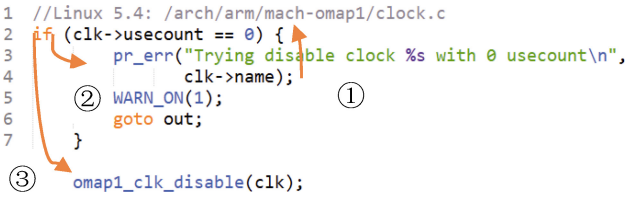

图 6 回溯过滤示意图

Figure 6 Back filtering diagram

若要确定一个可疑函数的调用环境, 就需要获取程序的控制流图。为此, 我们采用 LLVM 框架来分析 Linux 编译的中间代码。由于确定一个可疑函数是否为异常处理函数只需要找到其用于一处类安全检查模式, 而确定全部的安全检查需要确定异常处理函数的全部调用情况并判断其是否为满足安全检查模式。因此结合安全检查的异常处理函数检测与直接检测异常处理函数有着本质的区别。

C 语言中的分支语句主要包括两种, 即 IF...ELSE 语句和 SWITCH...CASE 语句。因此我们将函数调用环境分为三类, IF...ELSE 语句块中, SWITCH...CASE 语句块中, 以及其他语句块中。回溯过滤的主要目的是筛掉非异常处理函数, 保留可能是异常处理函数的可疑函数。具体来说, 针对每一个可疑函数, 检查其是否满足公式(4)。具体步骤如下:

(1) 如果在一个运行环境下调用了**可疑函数**, 则回溯找到父基本块。

(2) 确认父基本块以 IF 语句或者 SWITCH 语句等条件语句结尾, 确保可疑函数在 IF...ELSE 语句块中或 SWITCH...CASE 语句块中;

(3) 对条件语句的每一个分支进行检查, 分析其是否满足类安全检查模式, 即公式(4);

(4) 若满足类安全检查模式, 则保留该可疑函数; 否则, 筛除该可疑函数;

(5) 继续分析下一条可疑函数调用情况。

值得注意的是, 如果条件语句的全部分支都调用了可疑函数, 则不能确定该可疑函数是否为异常处理函数, 因为其本身为可疑函数这件事情就可能就是误报。

回溯过滤的具体过程可以参考如下伪代码:

算法 1.回溯过滤.

输入: *CI* 调用可疑函数的指令的指针; *SEH* 可疑函数集; *CEH* 异常处理函数集

输出: 扩充后的 *CEH* 异常处理函数集

1: *PBB*←*CI->getParent*() // 获得父基本块指针

2: *PI*←*PBB->end*() // 获得父基本块结尾语句指针

3: IF *PIB*←(*BranchInst*)*PI*: // 判父基本块尾语句是否为分支语句

4: *i*←0;*CountFlag*←0

5: DO:

6: *tmpBlock*←*PIB->getSuccessor*(*i*) // 获取第 i 个分支

7: FOR (*I*：*tmpBlock*): //获取分支中的每一条语句

8: IF (*call*=(*CallInst*)*I* && *call*→*getCalledFunction*() IN *SEH*): // 判断是否存在调用可疑函数的情况

9: *CountFlag*+=1

10: BREAK

11: END IF

12: END FOR

13: WHILE *i*<*PIB->getNumsuccessors*()

14: IF (0<*CountFlag*<*PIB->getNumsuccessors*()): // 判断是否满足类安全检查模式

15: *CEH.insert*(*CI->getCalledFunction*()

16: END IF

17: return *CEH*

同时, 为了能够辅助 5.1 自然语义感知的检测, 需要对异常处理的函数进行函数名的分词, 并计算其在整个异常处理函数集出现的频率。

5.4 基于异常处理函数的安全检查检测

基于异常处理函数检测安全检查的过程与5.3.2 节所述的回溯过滤方法基本相同, 不过其检查的对象不是可疑函数而是异常处理函数。检测时应该对所有调用异常处理函数的位置进行确认, 确认其函数调用情况是否满足为安全检查模式。

具体步骤如下:

(1) 如果在一个运行环境下调用了**异常处理函数**, 则回溯找到父基本块;

(2) 确认父基本块以 IF 语句或者 SWITCH 语句等条件语句结尾, 确保异常处理函数在 IF...ELSE 语句块中或 SWITCH...CASE 语句块中;

(3) 对条件语句的每一个分支进行检查, 分析其是否满足安全检查模式, 即公式(3);

(4) 若满足安全检查模式, 则保留该分支语句为安全检查语句; 否则继续执行;

(5) 继续分析下一条异常处理函数调用情况。

6 实验与评估

6.1 可移植性

由于 LLVM[24]支持对于中间代码的分析, 其可移植性高于针对高级语言的分析方法, 因此我们选取 LLVM 来实现我们的设计。针对其他软件检测异常处理函数仅需修改函数名分词规则即可。根据函数名变量名命名规则, 常见的分词规则有下划线分词法和驼峰分词法。

同时, LLVM[25]是专门用来分析中间代码的开源工具, 其对控制流及调用流的分析提供非常好的支持, 这对我们在 5.2 节回溯过滤以及 5.3 节基于程序语义的异常处理函数检测中提供了非常大的帮助。

6.2 实验结果

我们对实验结果进行抽样统计评估, 以此获得实验的过滤百分比、准确率。实验设计具体如下, 对于实验得到的异常处理函数随机抽样 20 个, 然后进行人工确认, 统计得出 Sased 在不同阶段的过滤百分比与准确率。

Sased 的超参数只有一个, 即 5.1 节中的频率阈值 θ。本节主要讨论不同的 θ 对于 Sased 的影响。如图 7 和表 1 所示, 随着 θ 的降低, Sased 的能发现的异常处理函数越来越多, 但其准确率与过滤百分比越来越低, 这是数量与精度的矛盾。同时, 当 θ 高于 0.04 后, Sased 的效果基本不变, 这是因为基本没有未加入可疑关键词集的分词片段的频率度能达到 4%。

表 1 Sased 在不同 θ 值下的表现

Table 1 Sased's performance under different θ values

θ	可疑函数	异常处理函数	准确率/%	过滤百分比/%
0.05	2070	795	94.06	38.41
0.04	2070	795	94.06	38.41
0.03	3092	1039	80.60	33.60
0.02	3492	1146	69.78	32.82
0.01	4075	1312	58.54	32.20
0	4450	1402	60.24	31.51

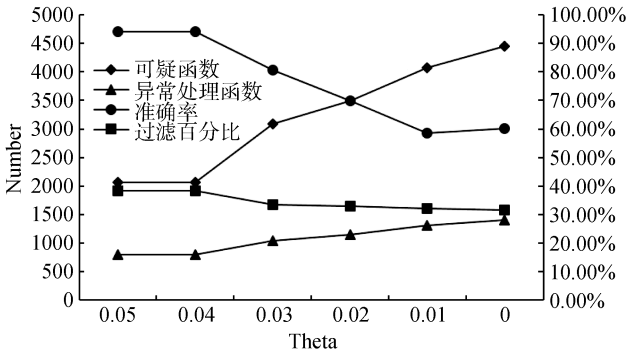

图 7 θ 对于 Sased 的影响

Figure 7 The influence of θ on Sased

由于 Sased 在不同 θ 值下会有不同的准确率与过滤百分比, 我们选取准确率最高的情况作为我们的最终实验结果。Sased 共发现安全检查实例 41519 个, 异常处理函数 795 个。对于这 795 个异常处理函数, 人工确认得到 743 个真实异常处理函数。我们的检测仍然存在着误报, 其原因是虽然可疑函数与非可疑函数之间属于不同类别函数是高概率的, 但仍存在可疑函数与非可疑函数都是正常处理函数或都是异常处理函数的可能。即便如此, Sased 找到异常处理函数的总量已经远超其他相关方法(例如 6.3 节中的 CRIX 和 CHEQ)。

6.3 结果对比

在之前的工作中, 只有 CRIX 和 CHEQ 进行了与本论文类似的工作, 为了评估 Sased 在异常处理函数以及安全检查检测的数量以及准确率方面的优势, 将 Sased 与二者进行对比实验。

首先, 通过对 CRIX 和 CHEQ 源码的分析, 我们发现 CHEQ 使用与 CRIX 基本一致的检测方法, 为了与这二者进行对比, 我们在 Linux5.3.0 版本上分别实现了 Sased、CRIX 和 CHEQ, 其中, CRIX 共报告异常处理函数 572 个, 准确率为 93.5%, 共报告安全检查 39777 个; CHEQ 共报告异常处理函数

572 个, 准确率为 86.5%, 共报告安全检查 38043 个, 准确率为 98.2%, 而 Sased 共报告异常处理函数 795 个, 准确率为 94.06%, 共报告安全检查 41519 个, 准确率为 92.5%。需要说明的是, CRIX 与 CHEQ 检测的安全检查数量小于其论文中的数量, 这是由于我们使用的数据集不同, 本数据集共有条件语句 324842 条。在这其中, 有 208 个异常处理函数是 CRIX 和 CHEQ 从未检测到的, 具体新增的异常处理函数见附录 A。Sased 与相关工作的实验对比结果如表 2 所示。

表 2 Sased、CRIX 和 CHEQ 对比表

Table 2 The comparison table of Sased, CRIX and CHEQ

方法	异常处理函数	准确率/%	安全检查	准确率/%
Sased	795	94.06	41519	92.5
CRIX	572	93.5	39777	-
CHEQ	572	86.53	38042	98.2

需要说明的是, 由于 Sased、CRIX 和 CHEQ 三种方法获得的安全检查的数量十分庞大, 进行完全的人工确认将是十分巨大的工程, 所以在本文中采用了抽样检测的方法, 随机抽取 300 个安全检查, 对其进行人工确认, 统计准确率。同时由于 CRIX 并没有生成安全检查的中间结果, 本文并未对其检测安全检查的准确率进行统计。

6.4 漏洞检测

为了验证异常处理函数与安全检查对于漏洞检测的重要性, 我们进行基于异常处理函数和安全检查的漏洞检测。

与 6.3 的实验设置相同, 针对 Linux5.3.0 内核, 使用 wllvm 生成 LLVM 中间代码, 然后针对 LLVM 中间代码进行漏洞检测。实验与 CRIX 的区别是, 我们输入 Sased 检测出来的范围更广的异常处理函数和安全检查。实验结果表明, 基于 Sased 的漏洞检测比原本的 CRIX 多报告了 196 个可疑 bug, 目前经过人工确认了 9 个真实漏洞, 并上报给了 Linux 社区。这 9 个漏洞也同时存在于当前的最新版本内核 Linux5.5-rc6 中, 漏洞具体情况如表 3 所示。

表 3 基于 Sased 找到的安全检查缺失漏洞

Table 3 Missing-check vulnerability found based on Sased

漏洞位置	函数	变量	缺失检查类型	漏洞类型
/arch/x86/xen/p2m.c 232 行	xen_build_mfn_list_list	p2m_top_mfn_p	返回值 alloc_p2m_page	空指针解引用
/arch/x86/xen/p2m.c 235 行	xen_build_mfn_list_list	p2m_top_mfn	返回值 alloc_p2m_page	空指针解引用
/drivers/usb/host/xhci-pci.c 276 行	xhci_pme_acpi_rtd3_enable	obj	返回值 acpi_evaluate_dsm	释放空指针
/lib/dynamic_debug.c 1066 行	dynamic_debug_init	cmdline	返回值 kstrdup	释放空指针
/mm/page_alloc.c 6174 行	setup_zone_pageset	zone	返回值 alloc_percpu	空指针解引用
/kernel/trace/ring_buffer.c 1959 行	rb_inc_iter	head_page	返回值 rb_set_head_page	空指针解引用
/kernel/bpf/btf.c 2082 行	btf_array_seq_show	elem_type	返回值 btf_type_id_size	空指针解引用
/fs/ecryptfs/keystore.c 1663 行	decrypt_passphrase_encrypted_session_key	tfm	参数 ecryptfs_get_tfm_and_mutex_for_cipher_name	空指针解引用
/drivers/usb/host/xhci.c 3216 行	xhci_endpoint_reset	ctrl_ctx	返回值 xhci_get_input_control_ctx	空指针解引用

7 讨论与总结

7.1 方法的局限性

尽管 Sased 在一定程度上扩充了操作系统的异常处理函数与安全检查的检测集, 但仍存在一些问题:

(1) Sased 不能发现那些既没有负向词片段又不是异常处理函数的父子函数的函数。这是由于 Sased 在可疑函数判断上直接采取一个负向词库匹配的方法实现, 该方法对负向词的质量要求较高。为了解决这个问题, 日后可以考虑结合情感分析(emotion analysis)技术。

(2) 由于 Sased 是结合安全检查对异常处理函数进行确认的, 所以对于不参与安全检查的可疑异常处理函数无法进行确认。

7.2 未来的工作

(1) 目前基于安全检查的漏洞检测技术, 其主要原理是通过安全检查确定临界变量, 根据同一个临界变量应该具有大致相同的安全检查的特性来检测安全检查漏洞。但根据我们的经验发现, 内核安全检查的其实不是临界变量, 而是危险操作或者函数。在未来的工作中, 我们考虑通过安全检查确定危险操作或函数, 再反过来确定对于其是否存在安全检查相关的漏洞。

(2) 仅针对异常处理和安全检查的检测来说, 在安全检查路径中, 对于异常情况的处理不仅仅

包括异常处理函数, 还包括对异常情况的修复等等, 今后可以将对异常的其他处理情况加入我们的工作中。

7.3 总结

良好的异常处理能力, 是操作系统健壮性的体现, 是系统可靠性的基石。本文从异常处理函数入手, 提出了基于自然语义和程序语义的异常函数检测技术、异常处理函数过滤方法以及基于异常处理函数的安全检查检测技术, 所提出的 Sased 方法共报告异常处理函数 795 个, 安全检查 41519 个, 都达到了 90%以上的检测准确率。这其中, 有 208 个异常处理函数是之前的工作中从未发现的。同时, 我们结合已有的漏洞检测方法, 发现了 Linux 内核的 9 个缺失安全检查漏洞。这也为之后的漏洞检测工作提供了坚实的基础。

参考文献

[1] Melliar-Smith P M, Randell B. Software reliability: The role of programmed exception handling[C]. *1977 ACM conference on Language design for reliable software*, 1977: 95-100.

[2] Wang W, Lu K, Yew P C. Check it again: Detecting lacking-recheck bugs in os kernels[C]. *2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018: 1899-1913.

[3] Ashcraft K, Engler D. Using programmer-written compiler extensions to catch security holes[C]. *2002 IEEE Symposium on Security and Privacy*, 2002: 143-159.

[4] Lu K, Pakki A, Wu Q. Detecting missing-check bugs via semantic-and context-aware criticalness and constraints inferences[C]. *28th {USENIX} Security Symposium ({USENIX} Security 19)*, 2019: 1769-1786.

[5] NVD NIST Search. https://nvd.nist.gov/. Nov. 2019.

[6] Kang Y, Ray B, Jana S. Apex: Automated inference of error specifications for c apis[C]. *31st IEEE/ACM International Conference on Automated Software Engineering*, 2016: 472-482.

[7] Rubio-González C, Gunawi H S, Liblit B, et al. Error propagation analysis for file systems[C]. *ACM sigplan notices*, 2009, 44(6): 270-280.

[8] Gunawi H S, Rubio-González C, Arpaci-Dusseau A C, et al. EIO: Error Handling is Occasionally Correct[C]. *FAST*, 2008, 8: 1-16.

[9] Lawall J, Laurie B, Hansen R R, et al. Finding error handling bugs in openssl using coccinelle[C]. *2010 European Dependable Computing Conference*, 2010: 191-196.

[10] Weimer W, Necula G C. Finding and preventing run-time error handling mistakes[C]. *ACM SIGPLAN Notices*, 2004, 39(10): 419-431.

[11] Weimer W, Necula G C. Exceptional situations and program reliability[J]. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 2008, 30(2): 8.

[12] Robillard M P, Murphy G C. Designing robust Java programs with exceptions[C]. *ACM SIGSOFT Software Engineering Notes*, 2000, 25(6): 2-10.

[13] Acharya M, Xie T. Mining API error-handling specifications from source code[C]. *International Conference on Fundamental Approaches to Software Engineering*, 2009: 370-384.

[14] Jana S, Kang Y J, Roth S, et al. Automatically detecting error handling bugs using error specifications[C]. *25th {USENIX} Security Symposium ({USENIX} Security 16)*, 2016: 345-362.

[15] Kang Y, Ray B, Jana S. Apex: Automated inference of error specifications for c apis[C]. *31st IEEE/ACM International Conference on Automated Software Engineering*, 2016: 472-482.

[16] Son S, McKinley K S, Shmatikov V. Rolecast: finding missing security checks when you do not know what checks are[C]. *ACM Sigplan Notices*, 2011, 46(10): 1069-1084.

[17] Gens D, Schmitt S, Davi L, et al. K-Miner: Uncovering Memory Corruption in Linux[C]. *NDSS*, 2018.

[18] Machiry A, Spensky C, Corina J, et al. {DR}.{CHECKER}: A Soundy Analysis for Linux Kernel Drivers[C]. *26th {USENIX} Security Symposium ({USENIX} Security 17)*, 2017: 1007-1024.

[19] Wang X, Chen H, Jia Z, et al. Improving Integer Security for Systems with {KINT}[C]. *10th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 12)*, 2012: 163-177.

[20] Lu K, Song C, Kim T, et al. UniSan: Proactive kernel memory initialization to eliminate data leakages[C]. *2016 ACM SIGSAC Conference on Computer and Communications Security*, 2016: 920-932.

[21] Wang W, Lu K, Yew P C. Check It Again: Detecting Lacking-Recheck Bugs in OS Kernels[C]. *2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018: 1899-1913.

[22] Lu K, Pakki A, Wu Q. Automatically Identifying Security Checks for Detecting Kernel Semantic Bugs[C]. *European Symposium on Research in Computer Security*, 2019: 3-25.

[23] Bai J J, Wang Y P, Yin J, et al. Testing error handling code in device drivers using characteristic fault injection[C]. *2016 {USENIX} Annual Technical Conference ({USENIX}{ATC} 16)*, 2016: 635-647.

[24] LLVM Overview. https://llvm.org/. Nov. 2019

[25] Li S F, An J X, Liu Y, et al. C++ source code coverage analysis instrumentation using Clang/LLVM [J]. *Computer science*, 2017, 44(11):191-194.
(李树芳, 安金霞, 刘洋, et al. 采用 Clang/LLVM 的 C++源代码覆盖率分析插装方法[J]. *计算机科学*, 2017, 44(11):191-194.)

附录

附录 A　新增的 208 个异常处理函数

Appendix 1　208 new exception handling functions

__acpi_handle_debug	ata_scsi_cmd_error_handler	filemap_fdatawait_keep_errors	quota_send_warning
__aer_print_error	ata_scsi_port_error_handler	filter_cpuid_features	raw6_icmp_error
__bad_area_nosemaphore	ata_sff_error_handler	fix_erratum_688	refcount_error_report
__blk_add_trace	ata_to_sense_error	fixup_exception	reset_control_assert
__blk_mq_end_request	bad_area	flush_end_io	reset_control_deassert
__die	bad_area_access_error	fpregs_assert_state_consistent	restore_reserve_on_error
__do_page_fault	bad_area_nosemaphore	fuse_aio_complete	rio_clr_err_stopped
__do_sys_exit	blk_status_to_errno	ghes_edac_report_mem_error	rt6_age_exceptions
__do_sys_exit_group	bpf_warn_invalid_xdp_action	handle_critical_trips	rt6_exception_hash
__ext4_error	check_mpx_erratum	icmpv6_err_convert	rt6_flush_exceptions
__ext4_error_inode	check_pcc_chan	init_emergency_isa_pool	rt6_insert_exception
__ext4_std_error	cper_mem_err_location	io_check_error	rt6_remove_exception
__ext4_warning	cpu_emergency_stop_pt	jbd2_journal_abort	rt6_remove_exception_rt
__ext4_warning_inode	crypto_alg_tested	kauditd_send_queue	rtnl_set_sk_err
__filemap_set_wb_err	crypto_hash_walk_done	keyctl_reject_key	rwsem_down_read_failed
__journal_abort_soft	dax_iomap_fault	kgdb_arch_handle_exception	rwsem_down_read_failed_killable
__log_error	dax_iomap_pte_fault	kgdb_arch_pc	rwsem_down_write_failed
__quota_error	df_debug	kgdb_cpu_enter	rwsem_down_write_failed_killable
__save_error_info	die	kgdb_skipexception	sched_numa_warn
__se_sys_exit	dio_warn_stale_pagecache	libata_trace_parse_eh_err_mask	scsi_end_request
__se_sys_exit_group	disable_err_thresholding	lockdep_assert_cpus_held	scsi_ioctl_block_when_processing_errors
__set_page_dirty	do_coprocessor_segment_overrun	log_arm_hw_error	sd_suspend_common
__trace_probe_log_err	do_device_not_available	log_error_deferred	search_exception_tables
__udp4_lib_err	do_divide_error	log_error_thresholding	segment_complete
__udp6_lib_err	do_double_fault	log_non_standard_event	selinux_netlbl_err
__warn	do_general_protection	math_error	set_one_prio
__warn_printk	do_int3	md_error	skb_recv_datagram
aa_audit_file	do_invalid_TSS	memory_failure	skb_tx_error
acpi_bios_error	do_invalid_op	memory_failure_hugetlb	skb_warn_bad_offload
acpi_bios_exception	do_kern_addr_fault	memory_failure_queue	slab_err
acpi_bios_warning	do_machine_check	mext_check_coverage	sock_alloc_send_pskb
acpi_check_address_range	do_overflow	microcode_sanity_check	sock_alloc_send_skb
acpi_error	do_page_fault	mm_fault_error	sock_queue_err_skb
acpi_exception	do_segment_not_present	mtrr_state_warn	sock_recv_errqueue
acpi_format_exception	do_trap	netlbl_skbuff_err	spurious_kernel_fault
acpi_print_osc_error	do_user_addr_fault	no_context	swap_writer_finish
acpi_ut_check_address_range	edac_pci_clear_parity_errors	notify_die	sysctl_err
acpi_ut_method_error	edac_pci_get_check_errors	of_i2c_setup_smbus_alert	sysfs_warn_dup
acpi_ut_predefined_bios_error	edac_raw_mc_handle_error	page_counter_try_charge	tcp_fastretrans_alert
acpi_ut_predefined_warning	efi_status_to_err	page_endio	tcp_req_err
acpi_ut_prefixed_namespace_error	emergency_restart	pci_cleanup_aer_error_status_regs	tomoyo_warn_oom
acpi_warning	emergency_sync	pci_disable_pcie_error_reporting	trace_kprobe_error_injectable
add_bytestring_header	err_pos	pci_enable_pcie_error_reporting	truncate_error_page
addrconf_dad_stop	errno_to_blk_status	pgtable_bad	unknown_nmi_error
aer_get_device_error_info	errseq_check	pkcs1pad_decrypt_complete	usercopy_warn
aer_print_error	errseq_check_and_advance	pkcs1pad_decrypt_complete_cb	vt_do_diacrit
aer_process_err_devices	errseq_sample	pkcs1pad_encrypt_sign_complete	warn_alloc
amd_mce_is_memory_error	errseq_set	pkcs1pad_encrypt_sign_complete_c	warn_bad_vsyscall
apei_mce_report_mem_error	ext4_clear_journal_err	pkcs1pad_verify_complete	wb_stat_error
append_filter_err	ext4_handle_error	pkcs1pad_verify_complete_cb	within_error_injection_list
arch_apei_report_mem_error	file_check_and_advance_wb_err	populate_error_injection_list	xfrm_local_error
ata_eh_analyze_ncq_error	filemap_check_errors	qdisc_create	

方宇彤　于 2018 年在河北大学网络工程专业获得学士学位。现在北京大学软件工程专业攻读硕士学位。研究领域为网络与系统安全。研究兴趣包括：信息安全。Email: 1801210820@pku.edu.cn

刘洪毅　于 2018 年在北京理工大学计算机科学与技术专业获得工学学士学位。现在北京大学计算机技术专业攻读工程硕士学位。研究领域为操作系统、漏洞挖掘。研究兴趣包括：机器学习、网络攻防。Email: hongyiliu@pku.edu.cn

李经纬 于 2016 年在东北大学大学软件工程专业获得工学学士学位。现在北京大学软件工程专业攻读工学硕士学位。研究领域为移动应用安全、机器学习。 Email: lijingwei@pku.edu.cn

文伟平 于 2004 年在中国科学院软件研究所获得信息安全方向博士学位。现任北京大学软件与微电子学院教授, 博士生导师。研究领域为系统与网络安全、大数据与云安全及智能计算安全研究。 Email: weipingwen@ss.pku.edu.cn