本科毕业设计（论文）

设计（论文）题目

Linux 内核的漏洞态势感知与预测

设计（论文）英文题目

Perception and Prediction of Vulnerability Situation of Linux Kernel

学　　院：计算机与信息技术学院

专　　业：信息安全

学生姓名：陈力恒

学　　号：14281055

指导教师：何永忠

北京交通大学

2018 年 6 月

学士论文版权使用授权书

本学士论文作者完全了解北京交通大学有关保留、使用学士论文的规定。特授权北京交通大学可以将学士论文的全部或部分内容编入有关数据库进行检索，提供阅览服务，并采用影印、缩印或扫描等复制手段保存、汇编以供查阅和借阅。

（保密的学位论文在解密后适用本授权说明）

学位论文作者签名：　　指导教师签名：

签字日期：　　年　　月　　日　　签字日期：　　年　　月　　日

中文摘要

摘要：随着计算机发展，软件安全备受关注。本文刚实现或发布的软件从不稳定状态到指定稳定状态（简称稳定化过程）的时长研究是漏洞态势研究的一个方向，指定稳定状态指一款软件的关键漏洞类型高危害程度的漏洞平均已被修复的状态。

本文研究 Linux 内核，其被广泛用于定制操作系统。我们认为刚实现或发布的 Linux 内核版本不够稳定，存在未知漏洞。预测稳定化时长，便能选取稳定版本定制系统，且为 Linux 内核开发人员提供新版本关键测试阶段时长的参考。

本论文编写爬虫工具从 NVD、GitHub、Kernel 等网站上获取 CVE 漏洞信息、补丁代码、内核源代码等。编写漏洞检测工具纠正 NVD“受漏洞影响版本”的不正确，编写代码定位工具纠正 Diff 代码中定位信息的不正确。通过分析历史数据得到漏洞危害等级走势图和关键漏洞类型。利用 BP 神经网络工具对历史数据训练建模，并预测高危害程度的关键漏洞类型的漏洞的最早存在到被发现时间间隔，加权平均求得 Linux 内核的稳定化时长。本文详细阐述了相关背景知识，阐述了从数据获取、纠正到预测的设计实现和实验结果，在文末提出了相关的结论和指导意见。

关键词：漏洞态势研究；Linux 内核；稳定化时长；漏洞检测工具；软件版本选择问题

ABSTRACT

ABSTRACT: With development of computer technology, people pay more and more attention to software security. Perception and prediction of duration from unstable stage to specified stable stage (referred to stabilization process) of a software that has just been implemented or released is one of directions of vulnerability situation research. The specified stable stage in this article refers to a stage that mainly critical and high-risk vulnerability has been repaired.

This paper focused on Linux kernel, because it's widely used to customize various operating systems. A version of Linux kernel that have just been implemented or released are not stable enough, and it tends to have a lot of unknown but serious vulnerabilities. By predicting duration of stabilization process, we can choose a stable Linux kernel version as a custom operating system's kernel. On the other hand, this can also provide Linux kernel developers with duration of critical testing phase of a new version of Linux kernel.

This paper uses crawler tools to get CVEs' information, patches and source trees of Linux kernel from websites including NVD, GitHub, Kernel and so on. This paper uses vulnerability detection tool to correct the fault of "Vulenrable Software and Version" of NVD and uses code location tool to correct the fault of code location in Diff code of GitHub and so on. Then get perceptual result of vulnerability situation, which including trend of vulnerability's level and cirtical vulnerability type. The historical data is trained by using BP neural network tool for modeling, and forecasts interval between exists and be discovered of critical high-level vulnerabilities, finally calculates duration of software stabilization. This paper elaborates the related background knowledge, elaborates from the design and implementation of data acquisition, correct, forecast to experimental results, and in the end puts forward some conclusions and guidance.

KEYWORDS： Vulnerability situation; Linux Kernel; Duration of Stabilization Process; Vulnerability detection tool; Software's version selection problem

目　　录

1 引言

本章讲述本毕业设计的研究背景、研究目的、研究内容、相关领域的工作、研究贡献、假设约束等，使得读者能对漏洞态势研究和本毕业设计能有一个初步的了解。

1.1 研究背景

如今，由于各类计算机、互联网的系统变得日益复杂，这些系统存在着许许多多的各种各样的危害程度不尽相同的漏洞。其中被利用的漏洞对计算机、互联网造成巨大的经济损失，而那些尚未被利用的漏洞使系统存在巨大的安全隐患。因此，漏洞研究越来越备受瞩目，这些关注和支持对漏洞研究提供了很好的经济基础和研究平台。

漏洞态势的研究作为漏洞研究的一部分，可以有效指导相关从业人员了解漏洞的总体情况和发展情况，从而使相关从业人员在决策中获得一定的理论和技术支持，使得决策更加准确和有说服力。本论文针对稳定化时长这一漏洞态势进行感知与预测。

以 Linux 内核为例，它是一款当下非常流行的开源电脑操作系统内核，被全球无数的程序员免费使用同时无数的程序员为 Linux 内核的开发提供无偿援助。Linux 内核的开源、免费等特征使其广泛地被应用于定制各类操作系统，其中非常著名的有 Android 系统、Ubuntu、CentOS 等，值得一提的是 Android 被广泛应用于智能电视、智能手机等各种面向大众的产品中，因此，我们可以看到 Linux 内核的稳定性非常大程度地影响了各类定制操作系统和相关产品的稳定性和安全性。我们认为那些新发布的软件版本由于尚未被广泛使用，很多关键漏洞还没有显现出来，因此这些新版本的软件一定程度上是不安全的或者是不稳定的。所以为了解决这个冲突，我们需要研究软件从刚实现或发布的时候的不稳定状态进入指定稳定状态（简称为稳定化过程）所需要的时间，以确定新发布的软件版本如 Linux 内核新版本需要多长时间的测试使用修复等工作才能够达到指定稳定状态，另一方面，可以确定以前发布的 Linux 内核的哪些版本已经到达了指定稳定状态，以应用于定制操作系统。

针对这一背景，在本论文中，以 Linux 内核为案例，详细讲述了 Linux 内核的稳定化时长这一漏洞态势的感知与预测的设计实现和结果，最终对实验结果进行分析，得出结论。

1.2 研究目的

本论文旨在预测一款软件的一个版本需要多长时间的漏洞发现、修补等工作才能完

成稳定化过程，从而在软件刚实现或发布的时候便能够一定程度地预测软件需要多长时间来实现稳定化过程，而并不需要等到软件已经经历了漫长的漏洞发现、修补工作之后才能确定软件已达到指定稳定状态。

首先，这一研究在软件的测试过程中能起到预测测试过程时长的作用。该研究可以帮助测试人员和需要考虑测试过程时长的决策人员来预测测试过程中需要多久才能使软件到达指定稳定状态，从而帮助决策层安排软件开发进度。

此外，这一研究可以帮助软件使用者选择合适的软件版本。在软件使用过程中，我们往往将刚刚发布的软件视为不够稳定的产品，即软件处于不稳定状态，那么该如何选择软件版本就会成为一个需要考虑的问题。一般来说，我们需要足够稳定的且最新的版本，目的在于这个版本的软件具备足够的稳定性并且具备更佳的性能和更完善的功能。只有在选择了合适的软件版本之后，才能展开后续工作。

以 Linux 内核为例，有许许多多的工业系统、零售系统、面向大众消费者的系统等均是以 Linux 内核为基础，进而实现定制的操作系统（如 Android 系统）。那么在 Linux 内核的版本选择过程中，所选择的 Linux 内核首先要满足稳定性的需求，其次要满足功能更完善、性能更佳的需求。那么通过本研究，将提取 Linux 内核中最关键的漏洞并预测它们的“最早存在到被发现的时间间隔”，从而确定 Linux 内核需要多长时间用于稳定化过程，进而可以判断哪些版本的 Linux 内核在当前时间下已处于指定稳定状态，我们便可以选取其中最新版本的 Linux 内核用于定制操作系统。

1.3 研究内容

我通过对实际问题进行分析，确定了本毕业设计的研究内容。问题分析和研究内容如下：

1. 如何研究一款软件的某个版本的稳定化过程所需要的时间？一款新版本的软件的漏洞往往是未知的，但我们可以根据该软件的历史数据确定该软件的发展过程中哪些漏洞是最致命、最核心的（称为关键漏洞），确定了这些漏洞之后，我们可以预测这些漏洞的最早存在到被发现修复的时间间隔，以此便可代表稳定化过程所需要的时间。
2. 如何获取软件的关键漏洞？获取并分析软件历年漏洞数据，如从漏洞危害程度、漏洞数量角度进行分析。
3. 如何获得一款软件历史数据？利用 NVD 数据库获得漏洞的危害程度、类型；利用补丁网站获取漏洞的补丁代码（补丁代码中的删除代码视为漏洞代码），利用软件源代码下载网站获取软件源代码。进而判断一个漏洞是否存在于一个特定版本的软件中及确定漏洞最早存在的时间，进而确定漏洞最早存在到被发现的

时间间隔。

4. 如何预测一个漏洞的最早存在到被发现的时间间隔？使用神经网络对历史数据中的特征因子（本论文将漏洞类型、漏洞危害程度、漏洞存在到被发现时间间隔等称为特征因子）进行训练，最后用于预测漏洞的最早存在到被发现的时间间隔这一特征因子。

因此，本毕业设计的研究流程如下：

1. 获取与纠正历史漏洞数据。
2. 对漏洞数据进行分析得出漏洞态势感知结果，其中包括软件的关键漏洞类型。
3. 对历史漏洞数据运用神经网络进行训练并得出较为精准的模型。
4. 利用训练出来的模型对高危害程度的关键漏洞类型的漏洞进行预测，得出高危害程度的关键漏洞类型的漏洞的最早存在到被发现时间间隔。
5. 最终确定软件稳定化过程所需时长。

1.4 相关领域

- **前人工作**

一方面，以 NVD 为例的漏洞数据库已到达一定的成熟程度，给我们在研究中获取数据这一环节提供了极大的便利。另一方面，开始出现以这些数据库为基础研究漏洞态势的论文，只是有些论文的研究深度和指导意义可能不够理想。

通过阅读漏洞态势相关的论文[1]-[14]，可以看到以往研究漏洞态势的论文主要旨在反应漏洞的趋势如何，其中有的论文提出了漏洞态势预测的思路，如通过历史数据曲线拟合的方法来实现预测。这些论文在给我提供研究思路这方面起了非常重要的作用，让我能够抓住漏洞的关键特征，并构思如何实现本毕业设计的漏洞态势的预测，从而展开并实现本毕业设计。

- **知识空白**

目前尚未出现对漏洞的最早存在到被发现的时间间隔与漏洞其他特征因子之间的关系的研究，因此也无法利用漏洞最早存在到被发现的时间间隔去指导测试人员预测软件测试过程时长，以及指导软件使用者选择软件版本。本毕业设计针对这一尚未解决的问题获取、纠正历史数据并提出了相关的预测方法和模型。

1.5 研究贡献

本节，我将阐述本毕业设计对相关领域做出的贡献、参考价值以及本毕业设计的创新点。具体贡献和创新点如下：

- 本论文编写了 Linux 内核有关的数据获取工具，获取了大量的 Linux 内核的漏洞数据和补丁等，其数量级在 1000，保证了研究的有效性。这些工具和数据为相关领域的研究提供数据支持的作用，从而极大地节省了未来研究的数据获取的成本。漏洞感知方法和对历史数据的漏洞感知结果能够为相关领域的研究提供直观的有意义的参考。
- NVD 和 GitHub 部分信息存在的错误予以纠正。第一点，在本毕业设计实验过程中发现 NVD 的受漏洞影响版本数据存在错误，因此我编写了漏洞检测工具来纠正该错误，该工具中还必须实现代码预处理工具、代码匹配工具和代码定位工具。第二点，在如 GitHub 发布的具体的补丁代码中存在一行 Diff 段的头信息（如“@@ -712,6 +712,7 @@ struct sctp_chunk {”），该信息主要为了反映漏洞的一个 Diff 段存在于漏洞文件的哪些行和哪些函数体或结构体中，但是该信息存在一定程度的错误，因此我编写了代码定位工具来解决这个问题。
- 本毕业设计首次以漏洞最早存在到被发现时间间隔为关键因子展开研究。探究漏洞最早存在到被发现时间间隔与漏洞类型、漏洞危害程度之间的关系，这是以往的研究没有涉及到的。漏洞最早存在到被发现时间间隔的研究和预测为基于 Linux 内核开发定制系统的人员和 Linux 内核测试人员提供决策支持。
- 本论文利用神经网络技术来实现稳定化时长这一漏洞态势的预测，不再纠结于传统的数学公式来拟合预测，这为相关领域试图使用神经网络进行研究提供了参考经验。
- 本论文以 Linux 内核为案例来研究探讨，其中编写的大部分工具、经验可以很好地应用于其他软件中，如漏洞检测工具、代码定位工具、BP 神经网络工具等。

1.6 假设约束

本节主要讲述本毕业设计的假设和约束，这些假设和约束是在具体实验过程中得出的，为了说明本毕业设计的结论的适用范围，并保证结论的质量。具体如下：

- 在本毕业设计中，暂不考虑 Linux 内核各版本之间的在关键漏洞类型方面的差异。
- Linux 内核版本采用 80 个左右的部分正式版，暂不考虑测试版本和其他版本。
- 本次实验考虑的是 Linux 内核的漏洞最早存在到被发现时间间隔与漏洞类型、漏洞严重程度之间的关联性。
- BP 神经网络的结构为 125-25-300（即包括输入层在内共三层的神经网络，输入层结点 125 个，隐含层结点 25 个，输出层结点 300 个）。
- 本人实现的漏洞检测工具未考虑漏洞文件名被修改这一类问题。

- 本毕业设计将高危害的 CWE-119、CWE-189、CWE-20、CWE-200、CWE-264、CWE-362、CWE-399、CWE-416 类型的漏洞视为 Linux 内核的关键漏洞，这些关键漏洞的最早存在到被发现时间间隔来代表 Linux 内核稳定化过程所需要的时间，即我们认为 Linux 某个版本内核的稳定状态是指上述类型的高危漏洞平均而言被修复的阶段。对于稳定状态的要求需根据具体情况而定。

1.7 论文结构

第 1 章：引言。讲述本毕业设计的研究背景、研究目的、研究内容、相关领域的工作、研究贡献、假设约束等，使得读者能对漏洞态势研究和本毕业设计能有一个初步的了解。
第 2 章：背景知识。讲述阅读本毕业设计可能需要用到的背景知识，以便读者更好地理解本文叙述的内容。
第 3 章：设计与实现。首先提出本毕业设计的研究方案，然后讲述本毕业设计的设计和实现，让读者能够对本毕业设计有一个详细的了解，并能够更好地理解本毕业设计的代码实现、实验结果和结论。
第 4 章：实验结果。展示利用本毕业设计最终得到的结果，使得读者对本毕业设计的工作内容能有直观的了解，并为第五章结论的得出提供基础。
第 5 章：论文结论。展示本毕业设计最终得到的结论和意义，体现本毕业设计的价值。

1.8 本章小结

本章主要阐述了稳定化时长这一漏洞态势的相关背景，并根据此背景得出目前需要解决的问题和本毕业设计的目的，并对问题分析以确定本毕业设计的研究方向和思路，同时阐述了相关领域其他人的贡献，并提出本毕业设计的创新点和贡献，最后阐述了本毕业设计所得出的结论所适用的假设和约束。

2 背景知识

本章讲述阅读本毕业设计可能需要用到的背景知识，以便读者更好地理解本文叙述的内容。

2.1 词汇说明

AM: After Modification,文中主要标志漏洞修改后代码文件
BM: Before Modification,文中主要标志漏洞修改前代码文件
BPNN: Back Propagation Neural Network
CPE: Common Platform Enumeration
CVE: Common Vulnerabilities and Exposures
CWE: Common Weakness Enumeration
NVD: National Vulnerability Database
漏洞：本文中主要指 CVE
漏洞代码：补丁代码中的删除代码
漏洞态势：本文中主要指漏洞态势中的一个方向——稳定化时长
模块：C 语言中的函数体和结构体在本文称为模块
稳定化过程：软件从不稳定状态到指定稳定状态的过程
稳定状态：这个在本文中指关键漏洞类型的高危害的漏洞平均而言被修复的状态
关键漏洞类型：根据漏洞在 Linux 内核中的重要程度确定的漏洞类型
关键漏洞：本文指高危害程度的关键漏洞类型的漏洞
Diff 段：补丁代码（Diff 代码）中的其中一段，代表了漏洞文件中每一处所需要修改的那一段代码
Module-BM 文件:存储 BM 文件的模块的关键信息的文件
Module-Diff 文件:存储 Diff 段所在模块的关键信息的文件

2.2 NVD

NVD 数据库中包含了以“CVE-年份-数字”为标题记载的许多漏洞，在官网中可以通过输入关键词来搜寻所需要的 CVE 信息。NVD 中的每一个 CVE 都具有一个 CVE 详细信息的页面，其中包括了漏洞描述、漏洞分析、漏洞类型、参考链接等诸多信息。NVD 给漏洞研究者们提供了很好的数据获取平台，极大地降低了漏洞信息获取的成本。当然

NVD 的功能远非如此，具体内容读者可以进入 NVD 官网一探究竟。

CVE 中受漏洞影响的版本信息以 CPE 形式表示，如“cpe:2.3:o:linux:linux_kernel:1.2.0:*:*:*:*:*:*:*”，其中包含了具体的名称（linux_kernel）和版本号（1.2.0）。本论文后续会介绍该信息存在一定程度的不完善和错误。

CVE 中的漏洞类型以 CWE 形式表示，如“CWE-119”，表示第 119 类漏洞——Buffer Errors。这一信息方便了本毕业设计确定每一个漏洞的漏洞类型，提供了漏洞态势预测模型的训练中至关重要的因子之一。

2.3 Linux 内核补丁相关说明

软件与漏洞的关系：每一款软件都会包含若干漏洞（在本文中特指 CVE），有的 CVE 会有补丁代码可供参考，这些补丁代码是用于对存在漏洞的软件版本进行维护升级。在本毕业设计中，这些补丁代码还用于漏洞检测，以实现对获取的历史数据进行纠正这一关键点。

补丁与补丁代码文件的关系：对于一个漏洞而言，往往会涉及多个文件，即多个文件同时存在漏洞代码并以整体表示成这个漏洞，其中每一个文件都是漏洞文件，对应的补丁也会存在若干个相应的文件，我们将其称为补丁代码文件，也就是说一个补丁会存在若干个补丁代码文件。

补丁代码文件与 Diff 段的关系：每一个补丁代码文件中包含了多个 Diff 段，也就是说在一个漏洞文件中，有多处代码需要进行修改，而这每一处代码所需要执行的修改就称为 Diff 段。

Diff 段的结构：每一个 Diff 段中包含 Diff 头、补丁代码。Diff 头是指形如“@@ -712,6 +712,7 @@ struct sctp_chunk {”的关于 Diff 段的说明，旨在说明 Diff 段所在的行号和函数体或结构体的名称。本文将函数体、结构体简称为模块。

Diff 段的补丁代码的结构：删除代码、增加代码、非增加非删除代码，其中删除代码视为漏洞代码，代表软件原版本中存在的漏洞。

2.4 BP 神经网络

BP(back propagation)神经网络是 1986 年由 Rumelhart 和 McClelland 为首的科学家提出的概念，是一种按照误差逆向传播算法训练的多层前馈神经网络，是目前应用最广泛的神经网络。

BP 神经网络通过对数据进行训练之后用于预测，训练中最为关键的成果是神经网络中各层之间的权重值。BP 神经网络的训练步骤主要为前馈和反馈过程。在前馈中，

将输入层的数据通过逐层的权重计算，得到输出层的数据，当输出层的数据不够贴近输出参考值的时候（即损失函数不够小），那么需要对输出值和参考输出值进行求差值，并通过 S 型函数确定误差，并通过权重逐层地反馈，以确定权重层需要修改的量以修改权重，当损失函数足够小便停止训练，这时的权重便是最终所需要的权重，是这一个模型的核心。另一方面，BP 神经网络的测试和预测便是对测试、预测输入进行一次前馈过程。

2.5 本章小结

本章阐明了本文所使用的某些词汇的具体含义、NVD 的内容结构和使用方法、Linux 内核补丁的内容结构以及 BP 神经网络的基本知识，这些知识将在研究中和后续论文中被频繁涉及，因此理解这些背景知识是一个必不可少的环节。

3 设计与实现

本章首先提出本毕业设计的研究方案，然后讲述本毕业设计的设计和实现，让读者能够对本毕业设计有一个详细的了解，并能够更好地理解本毕业设计的代码实现、实验结果和结论。

3.1 研究方案

通过第一章对研究问题和研究内容的探讨，可以得到实现并解决研究问题的思路。具体的研究方案如下：

1. 确定软件名称为 Linux 内核；
2. 通过 NVD 获取软件相应的 CVE 信息，如漏洞类型、漏洞危害程度；
3. 通过 GitHub 等网站获取每一个 CVE 的漏洞发现及修复时间、补丁代码；
4. 在 Linux Kernel 官网中下载所有版本的 Linux 内核源代码；
5. 利用 CVE 对应的补丁代码和所有版本的 Linux 内核源代码，通过漏洞检测工具判断 CVE 所存在的那些 Linux 内核版本；
6. 利用 CVE 漏洞最早存在的版本计算漏洞最早存在的时间；
7. 计算得出漏洞最早存在到被发现的时间间隔；
8. 所需要的漏洞特征因子获取完毕；
9. 将漏洞特征因子用于神经网络的训练；
10. 根据漏洞危害程度、漏洞数量、关注程度等确定软件的关键漏洞类型；
11. 将关键漏洞的特征因子输入神经网络得到预测的最早存在到被发现的时间间隔
12. 对预测得到的关键漏洞的最早存在到被发现的时间间隔通过加权平均等方法得到软件稳定化过程所需要的时间

据此绘制的研究方案流程图见图 3-1。

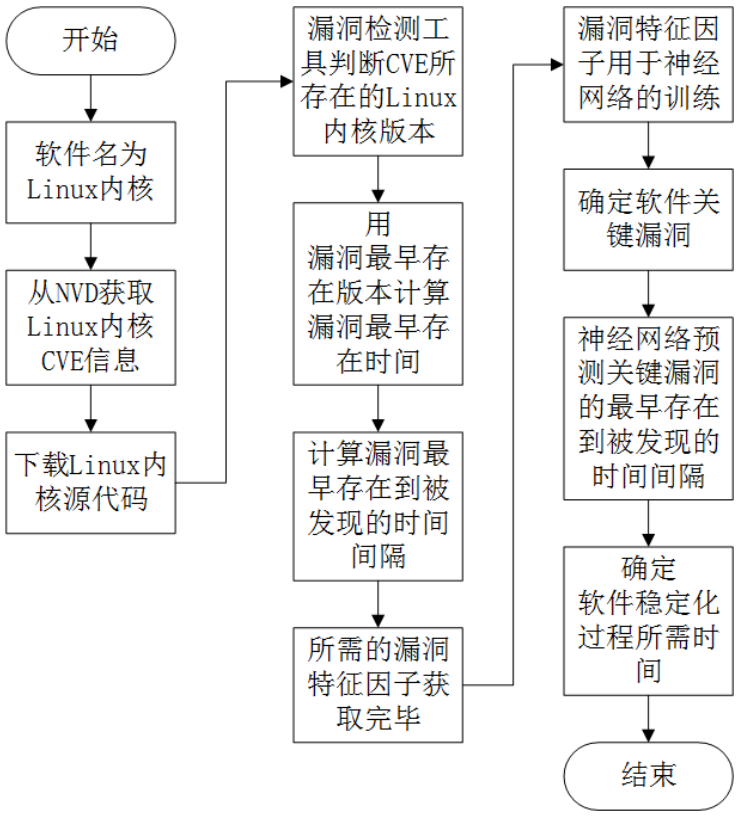


图 3-1 研究方案流程图

3.2 漏洞数据获取与筛选

本节主要讲述了稳定化时长这一漏洞态势的感知的设计实现过程中的漏洞数据获取、筛选工具的设计实现，为稳定化时长这一漏洞态势的感知结果的得出提供基础，并为稳定化时长预测模型提供可训练的样本。由于获取的部分数据不准确，因此需要下一节进行对漏洞数据的纠正。

1. **第一步。**我编写了“CVE 信息爬虫工具”用于获取 Linux 内核的所有 CVE 信息，将其保存到本地的 Excel 中，方便后续的研究分析，其中 Excel 的每一行代表一条 CVE 的所有信息（每一款软件往往有多个 CVE，即多个漏洞）。

   CVE 信息爬虫工具的设计实现：从指定 Excel 表格（如 OSSList.xls）中读取所需获取 CVE 信息的软件名称、关键词，利用 Requests 模块获取 NVD 查询结果网页（URL 的样式如 https://nvd.nist.gov/vuln/search/results?form_type=Basic&results_type=overview&query=keyword&search_type=all&startIndex=0）的源代码。在 NVD 查询结果网页的源代码中利用 CSS 选择器“strong[data-testid=vuln-matching-records-count]”定位并捕获该软件有关的 CVE 总数量，只有当 CVE 数量大于 0 时才继续后续的工作并确定 NVD 查

询结果网页 URL 中的“start index”值的范围。利用“div#row table[data-testid=vuln-results-table] tbody tr th strong a”定位到网页中所有的 CVE 编号，并为下一步确定每一个 CVE 的详细信息网页的 URL 打下基础。通过“https://nvd.nist.gov/vuln/detail/+CVE 编号”获得每一个 CVE 的详细信息网页的 URL，同样利用 CSS 选择器在 CVE 详细信息网页的源代码中获取所有的 CVE 信息。最后初始化一个 Excel 表格并往其中写入数据。

2. **第二步。**对获得的 CVE 信息中的参考链接根据其主网站站点 URL（如 git.kernel.org）进行计数，确定数量足够多且具有补丁代码的站点。

   CVE 参考链接统计工具的设计实现：在保存 CVE 信息的 Excel 中利用正则表达式“https?://[^/]*”查找所有以“http”开头的单元格并对该站点进行计数，最终保存在一个 Excel 表格中。手动确定各个站点的网页内容是否包含补丁代码，并最终确定数量足够多且具有补丁代码的站点。

3. **第三步。**从第二步的结果中确定数量足够多且具有补丁代码的站点之后（最终结果为“git.kernel.org”、“patchwork.kernel.org”、“github.com”这三个站点），将通过这些站点运用“补丁下载工具”将补丁保存到本地，用于之后的漏洞检测工具中。

   补丁下载工具：确定 NVD 查询结果网页并打开之，获取 NVD 查询结果网页的源代码，捕获所有的 CVE 编号并以此进入 CVE 详细信息网页。对每一条 CVE 的所有参考链接依序判断其是否为补丁代码链接（有补丁代码的参考链接），如果有则进入该网页将补丁代码保存至本地，之后对下一个 CVE 进行相同操作，如果没有则判断下一个参考链接（其中，每一个站点的网页样式、源代码各不相同，本毕业设计对每一个站点都需要具体处理）。最终保存的目录结构为每一个 CVE 编号均作为一个目录，其中包含补丁代码、补丁代码的来源链接、BM 文件（漏洞修改前该漏洞涉及到的文件的源代码文件）和 AM 文件（漏洞修改后漏洞涉及到的文件的源代码文件）。

4. **第四步。**这一步关于特征因子的选取。以 Linux 内核为案例，通过阅读相关论文[1]-[14]，我初步归纳了关于漏洞的那些比较重要的属性，具体如下：

   - 漏洞危害等级趋势：高、中、低
   - 漏洞影响程度：完全、部分、无
   - 漏洞类型趋势：Configuration……
   - 漏洞层次趋势：设计层、实现层、配置层
   - 漏洞来源：供应商、服务商
   - 漏洞数量：如漏洞数量在各年份的数量
   - 漏洞平均发现率：如平均一个月发现 2.9 个
   - 漏洞平均修补率

- 操作系统漏洞分布
- 漏洞危害类型：Disclosure、DOS、Theft of Service、Corruption、System Compromise
- System Compromise 原因：Run Arbitrary Code、Elevate Privileges、User Account Break-in、Root Break-in
- Corrective Actions Offered：Completely Vendor Solution、No Solution、Workaround、Average Time to Patch
- 漏洞的连接方式：网络、邻接网、本地

通过筛选，最终确定对本毕业设计而言最为重要的那些属性作为本毕业设计重点讨论的特征因子，具体如下：

- CVE Number：CVE 编号
- Linux Kernel Version Number：Linux 内核中最早存在该漏洞的版本号
- Vulnerability Exist Time：漏洞最早存在的时间
- Vulnerability Discover Time：漏洞被发现的时间
- Interval Between Exist And Discover：漏洞发现时间与最早存在时间的时间差
- Vulnerability Type：漏洞类型
- CVSS Severity V2：第二代 CVSS 评分体系
  - Base Score：基础评分
  - Attack Vector：攻击向量
  - Access Complexity：连接复杂性
  - Authentication：身份认证
  - Confidentiality Impact：机密性影响
  - Integrity Impact：完整性影响
  - Availability Impact：可用性影响

5. **第五步。**通过“特征因子爬虫工具”收集一款软件的所有漏洞的特征因子，为之后的漏洞态势预测工具提供可训练的样本。其中 NVD 发布的受漏洞影响的版本该信息被证实有错误（在第六步进行讨论）。

特征因子爬虫工具的设计实现：大致与 CVE 信息爬虫工具类似，但是捕获的信息为特征因子。特征因子中的漏洞被发现时间是优先从补丁网站搜集，如果不存在补丁网站，则以 CVE 的发布时间作为漏洞被发现时间，特征因子中的漏洞最早存在的版本是从 CVE 中的受影响版本中选取的最早发布的版本（该信息后被证实有错误），将受漏洞影响的版本通过查询“版本发布时间表”便可得到漏洞最早存在的时间。将漏洞被发现时间与漏洞最早存在的时间相减，便可得到漏洞最早存在到被发现的时间间隔。

6. **第六步。**这一步证明 NVD 中的受漏洞影响版本信息存在错误。以 CVE-2017-8069 为例，NVD 显示漏洞最早存在的版本为 Linux 内核的 4.9 版本中（见图 3-2），然而该漏洞在 Linux 内核 2.6.22 版本中已经存在，即图 3-3 中的漏洞代码存在于图 3-4 的 Linux 内核 2.6.22 版本的漏洞文件中。因此可以证明 NVD 中的受漏洞影响版本信息存在错误。

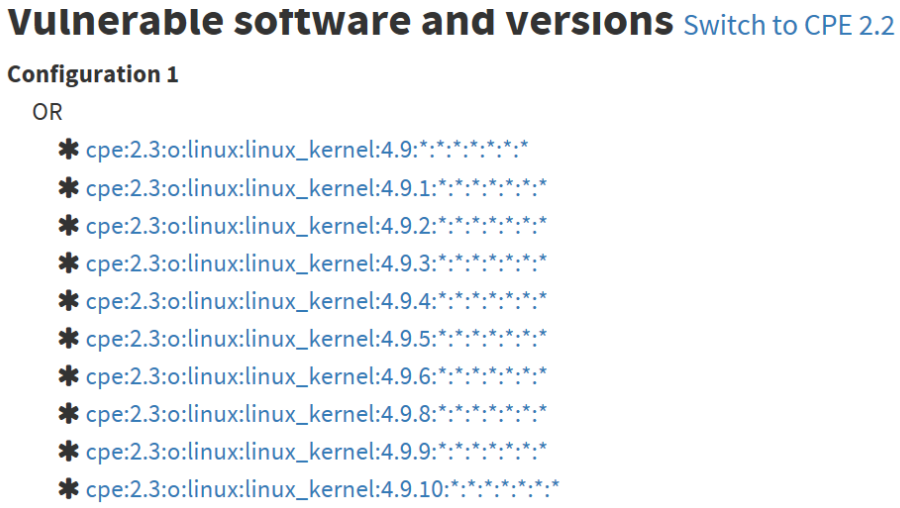


图 3-2 NVD 发布的受 CVE-2017-8069 影响的 Linux 内核版本号截图

```
diff --git a/drivers/net/usb/rtl8150.c b/drivers/net/usb/rtl8150.c
index 95b7bd0..c81c791 100644
--- a/drivers/net/usb/rtl8150.c
+++ b/drivers/net/usb/rtl8150.c
@@ -155,16 +155,36 @@ static const char driver_name [] = "rtl8150";
 */
 static int get_registers(rtl8150_t * dev, u16 indx, u16 size, void *data)
 {
-       return usb_control_msg(dev->udev, usb_rcvctrlpipe(dev->udev, 0),
-                              RTL8150_REQ_GET_REGS, RTL8150_REQT_READ,
-                              indx, 0, data, size, 500);
+       void *buf;
+       int ret;
+
+       buf = kmalloc(size, GFP_NOIO);
+       if (!buf)
+               return -ENOMEM;
+
+       ret = usb_control_msg(dev->udev, usb_rcvctrlpipe(dev->udev, 0),
+                             RTL8150_REQ_GET_REGS, RTL8150_REQT_READ,
+                             indx, 0, buf, size, 500);
+       if (ret > 0 && ret <= size)
+               memcpy(data, buf, ret);
+       kfree(buf);
+       return ret;
 }

-static int set_registers(rtl8150_t * dev, u16 indx, u16 size, void *data)
+static int set_registers(rtl8150_t * dev, u16 indx, u16 size, const void *da
 {
-       return usb_control_msg(dev->udev, usb_sndctrlpipe(dev->udev, 0),
-                              RTL8150_REQ_SET_REGS, RTL8150_REQT_WRITE,
-                              indx, 0, data, size, 500);
+       void *buf;
+       int ret;
+
```

图 3-3 CVE-2017-8069 对应的补丁代码部分截图

```
static int get_registers(rtl8150_t * dev, u16 indx, u16 size, void *data)
{
	return usb_control_msg(dev->udev, usb_rcvctrlpipe(dev->udev, 0),
			       RTL8150_REQ_GET_REGS, RTL8150_REQT_READ,
			       indx, 0, data, size, 500);
}

static int set_registers(rtl8150_t * dev, u16 indx, u16 size, void *data)
{
	return usb_control_msg(dev->udev, usb_sndctrlpipe(dev->udev, 0),
			       RTL8150_REQ_SET_REGS, RTL8150_REQT_WRITE,
			       indx, 0, data, size, 500);
}

static void ctrl_callback(struct urb *urb)
{
	rtl8150_t *dev;

	switch (urb->status) {
	case 0:
		break;
	case -EINPROGRESS:
		break;
	case -ENOENT:
		break;
```

图 3-4 Linux 内核 2.6.22 版本的 rtl8150.c 代码的部分截图

3.3 漏洞数据纠正

本小节旨在对上一节中获取数据进行纠正，以使漏洞态势感知与预测的结果更加准确和有说服力。

7. **第七步。**由于 NVD 中的受漏洞影响版本信息存在错误，为了纠正该错误，使所有特征因子都尽可能准确，必须编写一个“漏洞检测工具”。利用漏洞检测工具对一个 CVE 在所有 Linux 内核版本中判断，并选取最早的 Linux 内核版本，这就是漏洞最早存在的版本了，再查询 Linux 内核发布时间表便可得到漏洞最早存在的时间。将漏洞最早存在的时间与漏洞被发现的时间相减便可得到漏洞最早存在到被发现时间间隔。值得一提的是漏洞检测工具的实现需要依靠代码预处理工具、代码匹配工具、代码定位工具的实现。

   漏洞检测工具的设计实现：再次说明，一款软件往往存在若干个 CVE 条目（即多个漏洞）。每一个 CVE 条目往往包含若干个补丁代码文件（即一个漏洞涉及多个文件，这些文件均需要修改）。每一个补丁代码文件对应 Linux 内核的一个漏洞文件。每一个补丁代码文件往往具有多个 Diff 段（即一个漏洞文件中有多处地方需要修改）。本论文将补丁代码中的删除代码被视为漏洞代码，将 C 语言的函数、结构体都

视为模块。

以一项 CVE 的所有补丁代码文件（在代码实现过程中，一个补丁代码文件的文件名是该补丁代码对应的文件的路径名，其中路径分隔符替换成“~!@#”以防止操作系统不要将文件名错误地理解为路径名）和一个版本的 Linux 内核源代码树作为输入，以存在与否作为输出。

一个漏洞往往涉及多个文件，这些文件中都存在漏洞代码，因此这个漏洞对应的补丁表现为多个补丁代码文件，因此需要一个一个文件地判断漏洞代码是否存在于该版本内核的对应文件中。对于其中一个补丁代码文件，通过补丁代码的文件名确定该补丁代码相关的文件路径，据此找到该补丁相关的漏洞文件。

一个补丁代码文件中往往存在若干个 Diff 段，因此需要一一判断一个 Diff 段是否存在于漏洞文件中，由于如果只将 Diff 段中的删除代码（即漏洞代码）对漏洞文件全文匹配，那么可能在不相关的函数中查找到漏洞代码，这样就明显存在问题，因此通过代码定位工具确定漏洞代码所在模块名，然后再将漏洞代码在疑似漏洞文件的对应模块中判断是否存在，如果定位不到模块名，则在疑似漏洞文件中全文匹配。

代码预处理工具的设计实现：将代码进行预处理，以方便代码匹配等更为复杂的工作。比如将“static int crypto_ccm_auth(struct aead_request *req, struct scatterlist *plain,”转化为“static int crypto_ccm_auth (struct aead_request * req , struct scatterlist * plain ,”，也就是说将所有非字符与字符用空格进行隔开，同时去掉首尾的空格。只有这样，才能避免因为代码书写格式不一样的问题导致代码被判定为不同。

代码匹配工具的设计实现：我思考过两种方案：

- 方案一：判断一段代码是否按顺序出现在另一段代码中
- 方案二：一段代码与另一段代码完全相同

在实践过程中，由于一个 Diff 段中的漏洞代码在某些版本的内核的漏洞文件的漏洞代码处，其中间可能被穿插其他的代码，方案二不能在此条件下起作用。因此采用方案一。

判断代码段 A 是否依序出现于代码段 B。以代码段 A、代码段 B 作为输入，以是否存在作为输出。

每一次选取代码段 A 中的一行代码，进行代码预处理，然后与代码段 B 中的第一行开始逐行匹配（代码段 B 中的每一行代码也需要通过代码预处理）。如果代码段 A 中的每一行代码都依序出现于代码段 B 中，那么就输出存在。

代码定位工具的设计实现：该工具主要是针对 C 语言而设计的，由于 Linux 内核中 C 语言文件占了 77%（以 Linux 内核 4.0.1 版本为例，表 4-2），因此这个工具是有很大作用的。同时该工具还需要用到漏洞修改前代码文件，根据补丁代码来源

统计，存在漏洞修改前代码文件的补丁网站占了 99%（表 4-1）。

通过该工具，可以判断一段代码是否存在于一个目标文件中的结构体、函数体里面，如果是的话，返回结构体名或函数名，如果不是的话，则返回空，说明代码在后续漏洞检测工具中必须在可能的漏洞文件中进行全文匹配。

该工具可以首先将第三步中下载下来的代码中的“漏洞修改前代码”进行提炼，即将该文件的所有函数体、结构体都提炼为模块名称、模块信息及在文件中的首尾行号。该工具可以对 Linux 内核中可能存在漏洞的文件也进行相同的提炼。该工具还可以将补丁代码文件中的每一个 Diff 段根据其所在的模块，另外保存一份 Diff 段所在模块的模块名、首尾行号的文件。这样便可以很好的帮助漏洞检测工具的实现，并提高其效率。具体设计实现如下：

1. 首先，在一个代码文件中，根据正则表达式“\bstruct (\w+) {”提取所有的结构体名，根据正则表达式“(?!if|while)(\w+) (?:*)*(?!if|while)(\w+) \([^)]*\) {”提取所有的函数体名，这样就得到了所有的模块名。
2. 之后，重新从头遍历这个文件，找到第一个模块名，然后根据“{”“}”配对的原则，确定模块名后紧接着的“{”，并确定与该“{”匹配的“}”，然后将模块名的行号和“}”的行号同模块名一起保存在本地（为了节省代码定位工具所占用的时间，即避免重复对一个文件进行提炼）。后续所有的模块名均按此进行操作。
3. 上述步骤完成了代码模块提炼的工作。最初先完成漏洞修改前代码文件的代码模块提炼工作，之后利用代码匹配工具判断 Diff 段中的漏洞代码是否存在于一个模块，然后将这个模块的模块信息和首尾行号保存起来，说明该 Diff 段存在于这个模块中。
4. 代码定位工具的工作完成后，在漏洞检测工具中，Diff 段根据其所在的模块名在 Linux 内核的疑似漏洞文件中确定模块，然后将 Diff 段中的漏洞代码在 Linux 内核的该模块中利用代码匹配工具进行匹配，最终判断该 Diff 段是否存在。

3.4 漏洞态势感知

本小节具体阐述了漏洞态势感知模型，讲述了如何获得基本的漏洞趋势以及确定本毕业设计中非常重要的关键漏洞类型。

8. **第八步。**通过对纠正后的数据进行分析，我们首先根据每年发现的不同漏洞危害等级的漏洞数量绘制漏洞危害等级走势图，以便获得对漏洞趋势的直观的了解。
9. **第九步。**由于本毕业设计的关键问题是如何预测一款软件如 Linux 内核的稳定化过

程所需时长，因此我们需要确定 Linux 内核的关键漏洞类型并预测其高危害等级（漏洞危害等级的选取以实际需求为参考，本毕业设计中认为稳定的软件指的是一款软件的重要的高危的漏洞平均而言被修复，从而极大程度地避免了严重问题的出现）漏洞最早存在到被发现时间间隔以代表漏洞稳定化过程所需时长，那么如何获取 Linux 内核的关键漏洞类型呢？本毕业设计提出了四种确定关键漏洞类型的标准，具体如下：

- 根据各种漏洞类型的漏洞数量确定关键漏洞类型。
- 根据高危害漏洞中各种漏洞类型的漏洞数量确定关键漏洞类型。
- 首先利用公式“漏洞类型影响程度=1*低危害漏洞数量+2*中危害漏洞数量+3*高危害漏洞数量”确定各种漏洞类型的影响程度，然后选取漏洞影响程度最大的漏洞类型作为关键漏洞类型。
- 首先利用公式“漏洞类型影响程度=Σ该漏洞类型的每个漏洞的 CVSSv2 评分”确定各种漏洞类型的影响程度，然后选取漏洞影响程度最大的漏洞类型作为关键漏洞类型。

至此，我们便可获得关键漏洞类型，便能用于漏洞态势的预测了。在本毕业设计中，是将以上四个标准下的关键漏洞类型都作为需要考虑的关键漏洞类型。

3.5 漏洞态势预测

本小节阐述漏洞态势预测的模型设计实现和最终稳定化过程所需时长结果的得出方法。

10. **第十步。**将之前得到的准确的所有特征因子数据进行分组——训练组和测试组，用训练组在 BP 神经网络中进行建模，最终可以得到 BP 神经网路的各层之间的权重。将训练阶段得到的权重对测试组数据进行测试，确定模型的准确度。

BP 神经网路预测工具的设计实现：我同时采用了 Pybrain、我自己写的 BP 神经网络来建立神经网络。由于 Pybrain 会将样本分成训练组、验证组、测试组，当训练组和验证组的误差相近时，便会停止迭代，为了让迭代次数足够多，因此，我自己写了一个 BP 神经网路。

关于 Pybrain 实现 BP 神经网络比较简单，个人在此不再熬述。下面是关于我个人编写的 BP 神经网络。

根据输入的神经网络结构（如“[125, 25, 300]”），初始化各层的结点、各层之间的权重。由于在输出层可能是为了输出多个分类，如漏洞最早存在到被发现的时间间隔是在 0 个月到 300 个月之间，每一个月数都视为一个分类（因为我在反馈的过程中采用高确定度减小预测误差的方法，即以 Sigmoid 函数的导数去乘以输出差值，

因此该值总是往 0，1 这两个方向靠拢。），因此我才用独热的方式表示分类输出，如 300 个结点的独热输出。

在前馈过程中，利用 Numpy 模块实现矩阵运算加快前馈运算。在反馈过程中，利用 Sigmoid 函数的导数去乘以该层输出差值得到该层的输出误差，将前一层的转置乘以该层的输出误差作为两层之间的权重的误差，然后再将该层的输出误差通过权重反馈给前一层作为前一层的输出差值。其中在权重学习过程中，学习率这一参数利用学习退火算法使其在学习后期避免出现学习震荡现象（本人采用的方法是每 N 步，学习率减半）。

测试过程便是根据最新的权重进行一次前馈。

11. **第十一步。**确定 BP 神经网络预测工具满足要求的精度后，将高危害等级的关键漏洞类型的漏洞进行预测，获得关键漏洞的漏洞最早存在到被发现时间间隔，并以每个漏洞类型的漏洞数量为权重加权平均确定 Linux 内核的稳定化过程所需时长。

3.6 本章小结

本章首先确定了研究思路和流程，然后以分步的方式顺序地讲解了本毕业设计的研究过程，并在每一步中讲解具体的设计与实现，尤其是给出了漏洞态势感知方法和最终讲解了如何得到本毕业设计的关键问题的解答：Linux 内核稳定化过程所需时长。

4 实验结果

本章展示利用本毕业设计最终得到的结果，使得读者对本毕业设计的工作内容能有直观的了解，并为第五章的结论得出提供基础。

4.1 漏洞数据获取和筛选

本小节主要阐述了漏洞数据获取和筛选的结果，这些数据是后续结果得出的基础。

通过 CVE 信息爬虫工具，将所有 CVE 的各项信息均保存至 Linux 内核 CVE 信息表中（图 4-1，图 4-2），各项信息包括 CVE 编号、漏洞描述、分析描述、漏洞类型、CVSS 评分、参考链接等等。

				CVSS Severity V3			
CVE Number	Current	Analysis	Vulnerability Type	Base	Vector	Impact	Exploitability
CVE-2018-7492	A NULL pointer de						
CVE-2018-7480	The blkcg_init_que						
CVE-2018-7273	In the Linux kernel						
CVE-2018-6927	The futex_requeue						
CVE-2018-6412	In the fur	In the fun	Information Leak / D	7.5	CVSS:3.	3.6	3.9
CVE-2018-5750	The acpi	The acpi_	Information Leak / D	5.5	CVSS:3.	3.6	1.8
CVE-2018-5703	The tcp_	The tcp_\	Out-of-bounds Write	9.8	CVSS:3.	5.9	3.9
CVE-2018-5344	In the Lir	In the Lin	Use After Free (CW	7.8	CVSS:3.	5.9	1.8

图 4-1 Linux 内核 CVE 信息表部分截图（左半部分）

CVSS Severity V2				Reference 1	
Base	Vector	Impact	Exploitability	HyperlinkHyperlink	Resource
				http://git.kernel.org/c	
				http://git.kernel.org/c	
				http://www.securityfc	
				http://git.kernel.org/c	
5.0	(AV:N/A	2.9	10.0	https://marc.info/?l=li	Issue Tracking; Patch
2.1	(AV:L/A	2.9	3.9	https://patchwork.kei	Issue Tracking; Patch;
10.0	(AV:N/A	10.0	10.0	https://groups.google	Issue Tracking; Mailing
4.6	(AV:L/A	6.4	3.9	http://git.kernel.org/c	Patch; Third Party Advi

图 4-2 Linux 内核 CVE 信息表部分截图（右半部分）

为了获取 CVE 对应的补丁代码，需要确定存在补丁代码的网站。通过 CVE 参考链接统计工具，将参考链接的网站主站点 URL 进行计数（图 4-3），并逐一验证是否可以查阅到补丁代码。最后发现数量足够多且具有补丁代码的网站为 git.kernel.org（1158 次）、github.com（783 次）、patchwork.kernel.org（31 次）。

Hyperlink	Count
http://www.ubuntu.co	2166
http://lists.opensuse.c	1785
http://www.securityfo	1455
http://git.kernel.org	1158
http://www.openwall.(	1019
https://bugzilla.redhat	882
http://www.debian.or(	836
https://github.com	783
http://www.kernel.org	770
http://www.redhat.co(	732
http://rhn.redhat.com	700
http://www.mandriva.	510

图 4-3 CVE 参考链接的网站主站点 URL 的计数的部分截图

在确定了补丁代码网站之后，通过补丁下载工具，将每一个 CVE 的补丁信息各自保存在以 CVE 编号为名的文件夹中（图 4-4），总数共计 1131 个。

每一个 CVE 中的补丁等文件的结构如图 4-5 所示，Source.txt 保存补丁来源，以漏洞文件路径命名的文件是补丁代码文件（其中路径分隔符被替换成“~!@#”），“(AM)”开头的文件是漏洞修改后代码文件，“(BM)”开头的文件是漏洞修改前代码文件。

每一个补丁代码文件中往往由若干个 Diff 段，每一个 Diff 段如图 4-6 所示（图 4-6 所示的补丁代码文件恰巧只有一个 Diff 段），其中包括 Diff 头和补丁代码。

根据补丁来源进行统计得到表 4-1，可以看到具有 BM 文件（漏洞修改前代码文件）的 CVE 占到了 99%，具有 AM 文件（漏洞修改后代码文件）的 CVE 占到了 95%。

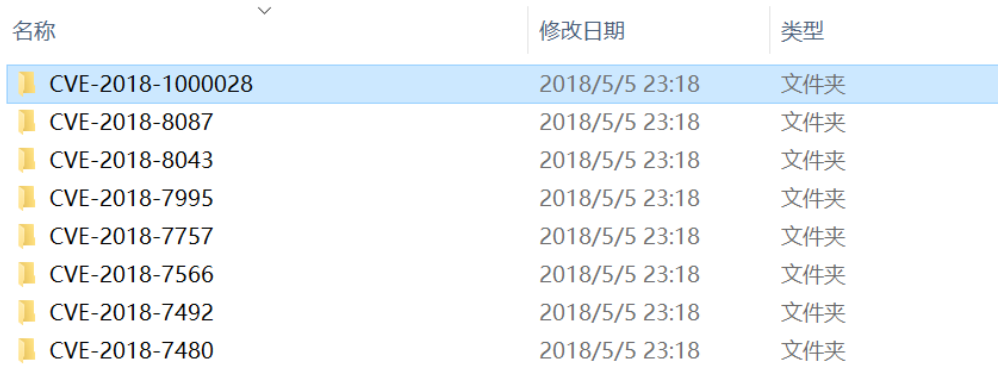


图 4-4 Linux 内核补丁目录部分截图

名称	修改日期	类型
(AM)net~!@#llc~!@#llc_conn.c	2018/4/3 17:44	C 文件
(AM)net~!@#llc~!@#llc_sap.c	2018/4/3 17:44	C 文件
(BM)net~!@#llc~!@#llc_conn.c	2018/4/3 17:43	C 文件
(BM)net~!@#llc~!@#llc_sap.c	2018/4/3 17:44	C 文件
net~!@#llc~!@#llc_conn.c	2018/4/3 17:44	C 文件
net~!@#llc~!@#llc_sap.c	2018/4/3 17:44	C 文件
Source.txt	2018/4/3 17:44	文本文档

图 4-5 CVE-2017-6345 目录结构截图

```
diff --git a/net/llc/llc_sap.c b/net/llc/llc_sap.c
index d0e1e80..5404d0d 100644
--- a/net/llc/llc_sap.c
+++ b/net/llc/llc_sap.c
@@ -290,7 +290,10 @@ static void llc_sap_rcv(struct llc_sap *sap, struct sk_buff *skb,

         ev->type   = LLC_SAP_EV_TYPE_PDU;
         ev->reason = 0;
+        skb_orphan(skb);
+        sock_hold(sk);
         skb->sk = sk;
+        skb->destructor = sock_efree;
         llc_sap_state_process(sap, skb);
 }
```

图 4-6 补丁代码文件 net~!@#llc~!@#llc_sap.c 代码截图

表 4-1 补丁来源表

网站	数量	百分比
git.kernel.org	1073	95%
github.com	43	4%
patchwork.kernel.org	15	1%

在第三章中我们确定了本文重点考虑的那些特征因子，通过特征因子爬虫工具，得到 Linux 所有 CVE 的特征因子（图 4-7，图 4-8），其中 NVD 发布的受漏洞影响的版本存在错误，因此需要有漏洞检测工具进一步纠正。

CVE Number	Version Number	Exist Time	Discover Time	Interval(Month)	Vulnerability Type
CVE-2018-7995	1.2.0	1995年03月	2018年03月	276	Race Conditions (CV
CVE-2018-7757	1.2.0	1995年03月	2018年01月	274	Resource Managem
CVE-2018-7755	1.2.0	1995年03月	2018年03月	276	Information Leak / D
CVE-2018-7740	1.2.0	1995年03月	2018年03月	276	Buffer Errors (CWE-

图 4-7 Linux 内核 CVE 特征因子部分截图（左半部分）

CVSS Severity V2

Base Score	Attack Vector	Access Complexity	Authentication	Confidentiality	Integrity	Availability
4.7;MEDIUM	Local	Medium	None	None	None	Complete
2.1;LOW	Local	Low	None	None	None	Partial
5.0;MEDIUM	Network	Low	None	Partial	None	None
4.9;MEDIUM	Local	Low	None	None	None	Complete

图 4-8 Linux 内核 CVE 特征因子部分截图（右半部分）

4.2 漏洞数据纠正

本小节是针对前一节获取数据的纠正，并展示纠正后的数据结果。

在展示纠正后的特征因子数据之前，我们有必要展示一下代码定位工具的成果，让读者能更好地理解漏洞检测工具的成果。

通过代码定位工具，可以将 BM 文件（漏洞修改前文件，图 4-9）中的模块信息提炼出来得到 Module-BM 文件（存储 BM 文件的模块的关键信息的文件，图 4-10），其中包扩了模块是结构体还是函数、返回值（函数具有）、模块名、起始行号、结束行号。

对于补丁（图 4-11）的定位如图 4-12，在 Module-Diff 文件中（存储 Diff 段所在模块的关键信息的文件）可以看到每一个 Diff 段所在的模块关键信息，每一行代表一个 Diff 段所在的模块信息，若该行为空，则说明这段代码处于全局中。通过 Moddle-Diff 文件来解决补丁代码中 Diff 头存在错误的问题。

表 4-2 显示了 C 语言文件占据了 Linux 内核所有文件的 77%，因此该工具是有意义的。

```
#include <crypto/internal/aead.h>
#include <crypto/internal/hash.h>
#include <crypto/internal/skcipher.h>
#include <crypto/scatterwalk.h>
#include <linux/err.h>
#include <linux/init.h>
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/slab.h>

#include "internal.h"

struct ccm_instance_ctx {
        struct crypto_skcipher_spawn ctr;
        struct crypto_ahash_spawn mac;
};

struct crypto_ccm_ctx {
        struct crypto_ahash *mac;
        struct crypto_skcipher *ctr;
};
```

图 4-9 漏洞修改前代码文件的代码截图

```
struct  ccm_instance_ctx        24      27
struct  crypto_ccm_ctx  29      32
struct  crypto_rfc4309_ctx      34      37
struct  crypto_rfc4309_req_ctx  39      43
struct  crypto_ccm_req_priv_ctx 45      52
struct  cbcmac_tfm_ctx  54      56
struct  cbcmac_desc_ctx 58      60
function        crypto_ccm_req_priv_ctx crypto_ccm_reqctx       62      68
function        int     set_msg_len     70      86
function        int     crypto_ccm_setkey       88      114
function        int     crypto_ccm_setauthsize  116     133
function        int     format_input    135     155
function        int     format_adata    157     174
```

图 4-10 Module-BM 文件内容截图

```
diff --git a/crypto/ccm.c b/crypto/ccm.c
index 4428488..1ce37ae 100644
--- a/crypto/ccm.c
+++ b/crypto/ccm.c
@@ -45,6 +45,7 @@ struct crypto_rfc4309_req_ctx {

 struct crypto_ccm_req_priv_ctx {
        u8 odata[16];
+       u8 idata[16];
        u8 auth_tag[16];
        u32 flags;
        struct scatterlist src[3];
@@ -183,8 +184,8 @@ static int crypto_ccm_auth(struct aead_request *req, struc
        AHASH_REQUEST_ON_STACK(ahreq, ctx->mac);
        unsigned int assoclen = req->assoclen;
        struct scatterlist sg[3];
-       u8 odata[16];
-       u8 idata[16];
+       u8 *odata = pctx->odata;
+       u8 *idata = pctx->idata;
        int ilen, err;

        /* format control data for input */
```

图 4-11 补丁文件中的 Diff 段截图

```
struct  crypto_ccm_req_priv_ctx 45      52
function        int     crypto_ccm_auth 176     233
```

图 4-12 Module-Diff 文件内容截图

表 4-2 Linux 内核文件类型分布表（Linux 内核 4.0.1 版本）

文件类型	数量	百分比
无后缀	4799	10%
.c	20981	43%
.S	1386	3%
.h	16555	34%
.dtsi	621	1%
.dts	791	2%
.txt	2698	5%
其他	1126	2%
总计	48957	100%

现在便可展示纠正后的特征因子数据（即漏洞检测工具的成果，图 4-13），由于部分 CVE 没有补丁代码，因此无法使用漏洞检测工具，由于本次实验采用的 Linux 内核为 80 个左右的 Linux 内核正式版本，有些漏洞存在于 Linux 内核的某些测试版本，因此这一部分在本次试验中检测不到 Linux 版本（对于这一点，只需要解压更多的 Linux 内核用于漏洞检测工具即可，只是漏洞检测工具所花费的时间将大幅增加，本人下载下来的 Linux 内核版本数量超过 2000）。最终得到 800 条包括漏洞最早存在版本在内所有信息均较为准确的 CVE 信息（不包含完整数据的 CVE 项需予以删除）。

CVE Number	Linux Kernel Version Number	Vulnerability Exist Time
CVE-2018-7995	3.3	2012年03月
CVE-2018-7757	2.6.19	2006年11月
CVE-2018-7755		
CVE-2018-7740		
CVE-2018-7492	2.6.30	2009年06月
CVE-2018-7480	4.3	2015年11月
CVE-2018-7273		
CVE-2018-6927	2.5.70	2003年05月
CVE-2018-6412	3.15	2014年06月
CVE-2018-5750	2.6.24	2008年01月

图 4-13 修正后的受漏洞影响的最早版本及其发布时间部分截图

4.3 漏洞态势感知

本小节主要讲述根据第三章中漏洞态势感知模型得出的结果。

在获取了 Linux 内核漏洞的基本历史数据后，对其进行分析统计，便可以得到 Linux 内核漏洞态势的感知结果。

首先，我们可以根据不同等级的漏洞数量走势绘制漏洞危害等级走势图（图 4-14），可以发现最下面的高危害漏洞的发现数量随着年份增加而波动上升，而中危害漏洞的发

现数量先增加后趋于稳定，在 37%左右，低危害漏洞的发现数量比例较低且数量非常稳定，在 10%左右。

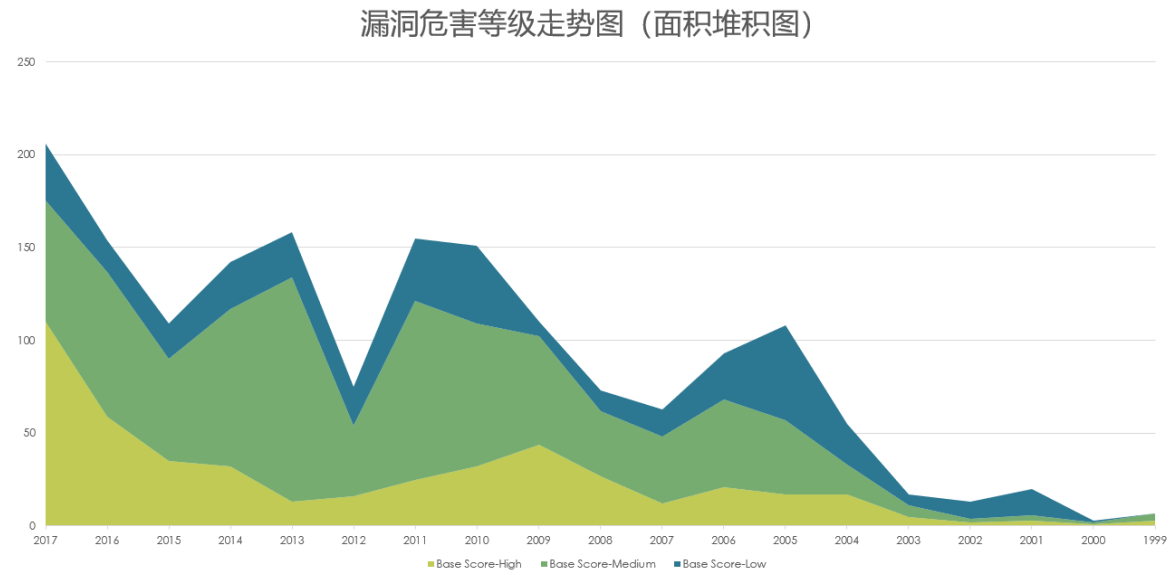


图 4-14 漏洞危害等级走势图

其次，我们可以得出不同标准下漏洞类型的分布情况，如图 4-15、图 4-16、图 4-17、图 4-18。我们可以发现并得出 CWE-264、CWE-362、CWE-189、CWE-200、CWE-20、CWE-119、CWE-399、CWE-416 为 Linux 内核历史中最为关键的漏洞类型（本毕业设计中，是将四个标准下的关键漏洞类型都作为需要考虑的关键漏洞类型。），当然随着时间的流逝，不同的版本的最关键漏洞类型会存在一些差别（最新发布的版本的 Linux 内核并无任何数据可以查询），在对本毕业设计具体使用过程中，可以结合实际经验在此基础上确定当前需要考虑的 Linux 内核关键漏洞类型，这将会起到更佳的作用。在本次实验中，以高危害程度的 CWE-264、CWE-362、CWE-189、CWE-200、CWE-20、CWE-119、CWE-399、CWE-416 作为关键漏洞，上述类型的漏洞数量分别为 203，107，164，161，170，85，204，26 个（共 1120 个），对应权重比例为 18%，10%，15%，14%，15%，8%，18%，2%。

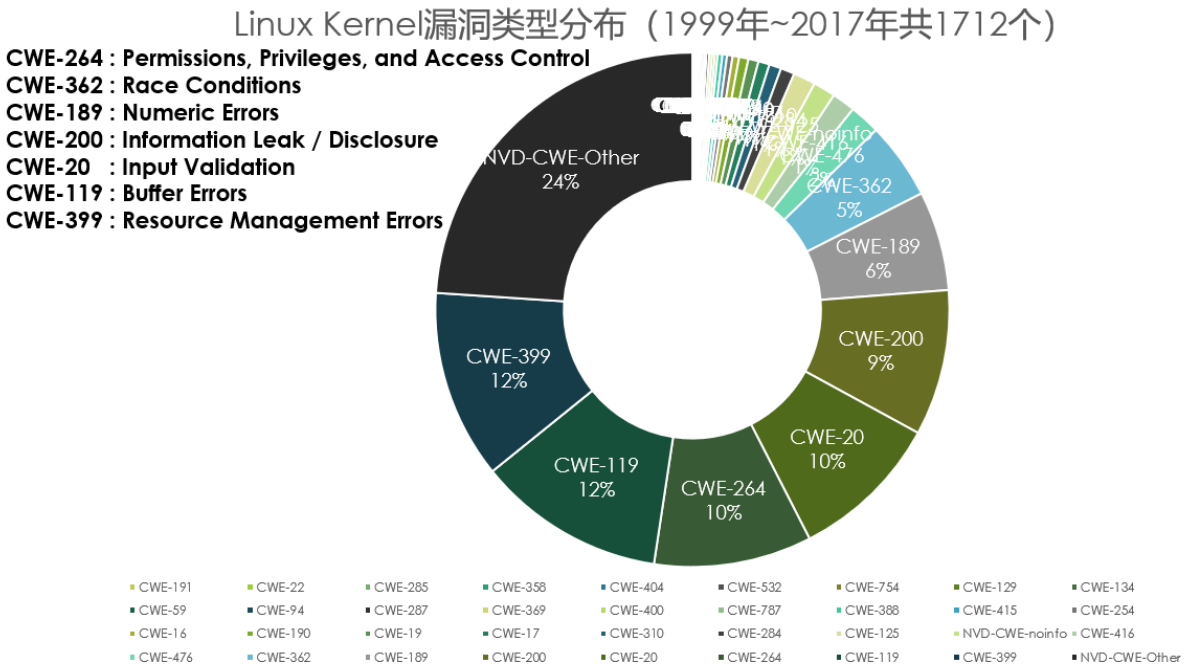


图 4-15 Linux 内核漏洞类型分布图

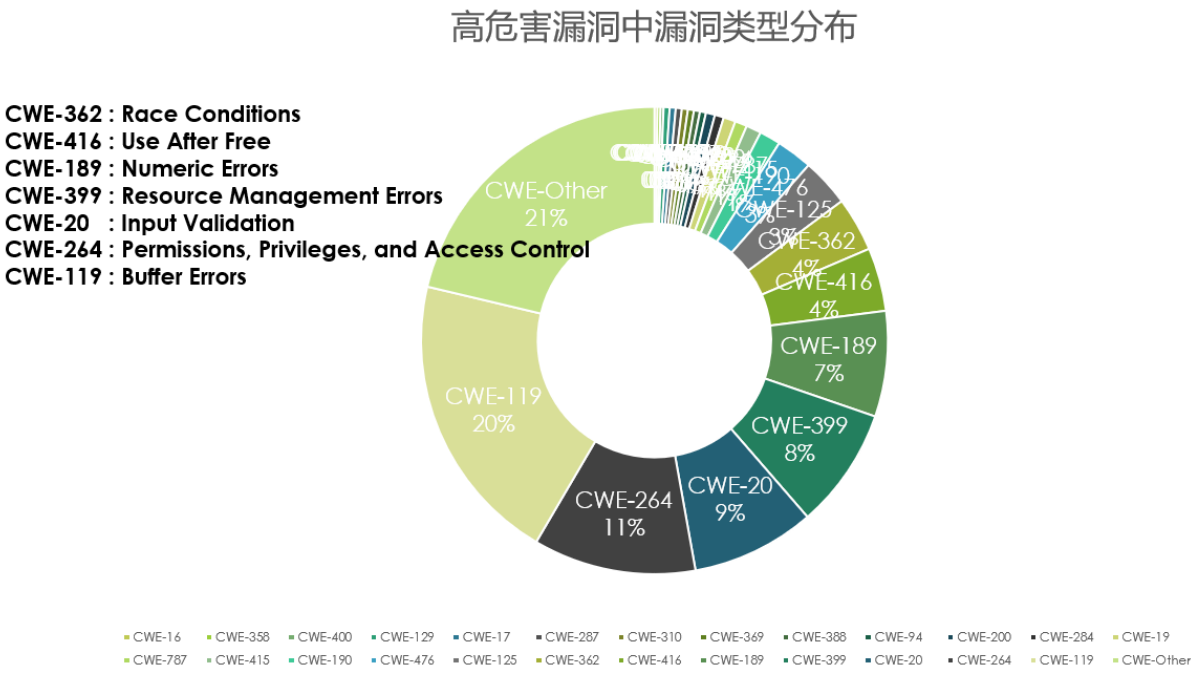


图 4-16 Linux 内核的高危害漏洞中的漏洞类型数量分布图

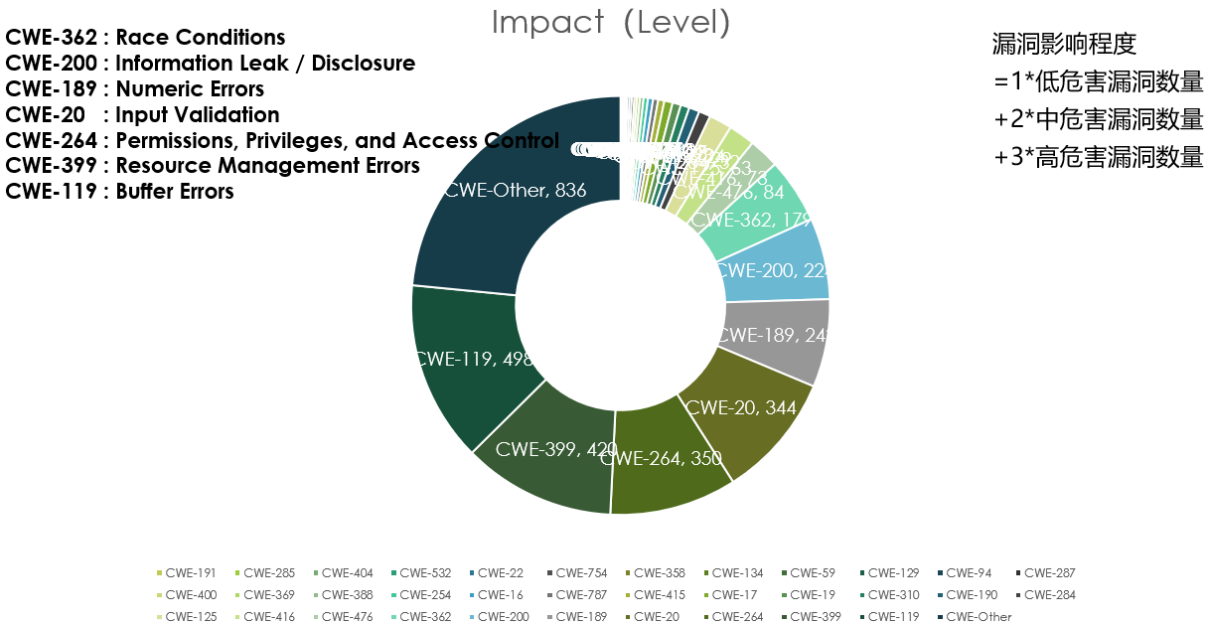


图 4-17 Linux 内核的漏洞影响程度分布图（标准一）

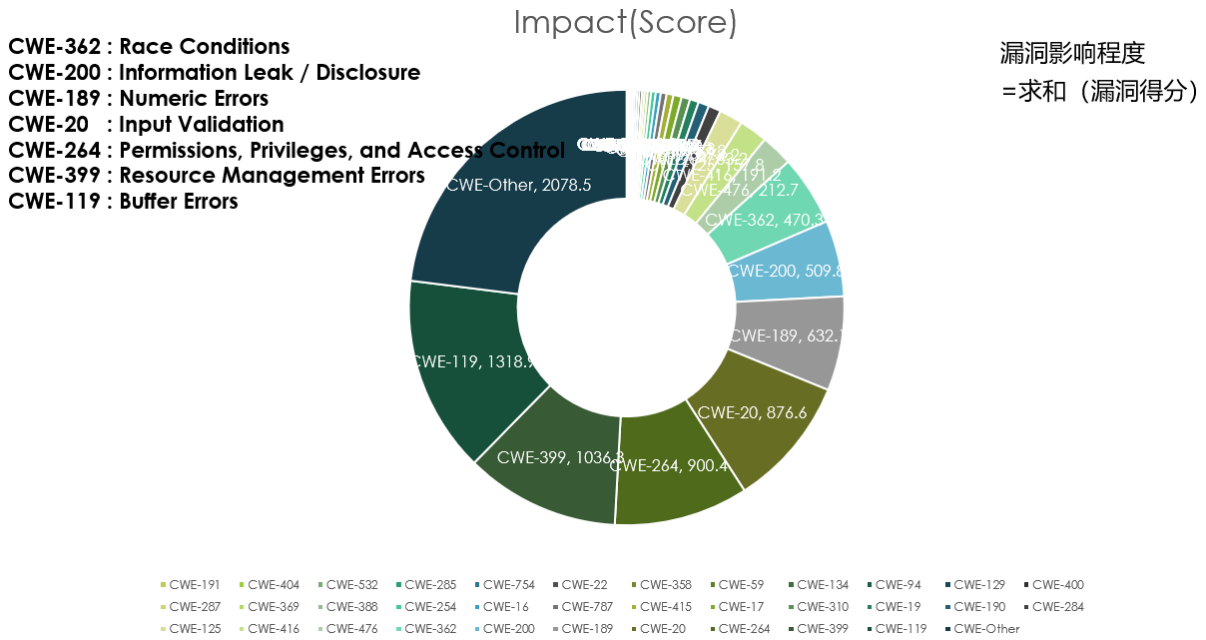


图 4-18 Linux 内核的漏洞影响程度分布图（标准二）

4.4 漏洞态势预测

我们已经获得了 800 条左右的较为准确的可训练数据，因此可以进行模型的训练。在此，本人使用了 Pybrain 工具和自己编写的 BP 神经网络工具。值得一提的是，Linux 内核的最早存在到被发现的时间间隔取值范围为 0 年-25 年（在实验中，为了表达为独热输出，月数的范围表示为 0 月-300 月）。

关于 BP 神经网络的输入，需要利用“特征因子标准化工具”对本次实验所用到的特征因子进行提取和标准化，其中 CWE 编号表示为独热模式，因为 CWE 编号为离散非连续数据。值得注意的是本次实验中 Interval 采用的月份数，使用的漏洞危害程度是 0 对应低危害，0.5 对应中危害，1 对应高危害（关于 CVSS 评分目前暂未采用）。将 CWE 编号的独热表示和漏洞危害程度作为 BP 神经网络的输入，Interval 作为神经网络的参考输出。

Pybrain 工具（BP 神经网络结构为 125-25-300，即包括输入层在内共三层的神经网络，输入层结点 125 个，隐含层结点 25 个，输出层结点 300 个）效果如图 4-19、图 4-20。测试样本测试的误差一年内的准确率为 43%，误差三年内的准确率为 64%，误差五年内的准确率 76%；回归测试的误差一年内的准确率为 32%，误差三年内的准确率为 58%，误差五年内的准确率 72%。

本人编写的工具（BP 神经网络结构为 125-25-300，即包括输入层在内共三层的神经网络，输入层结点 125 个，隐含层结点 25 个，输出层结点 300 个）效果如图 4-21、图 4-22。测试样本测试的误差一年内的准确率为 42%，误差三年内的准确率为 69%，误差五年内的准确率 77%；回归测试的误差一年内的准确率为 37%，误差三年内的准确率为 63%，误差五年内的准确率 75%。

值得注意的是，由于可能存在个例数据的异常情况，会对神经网络的效果起到一定程度的影响。此外，由于之前提到的 Linux 内核版本采用的数量不到 100，很多测试版本的 Linux 内核在本次实验中没有被考虑在内，因此可能出现漏洞最早存在版本不够精确的可能。再者，神经网络的结构层次和节点数需要通过实验来摸索出最佳的结构，125-25-300 可能不是本实验最佳的神经网络结构。综上原因都会对实验效果产生影响，这些可以在未来的研究中进行探讨。

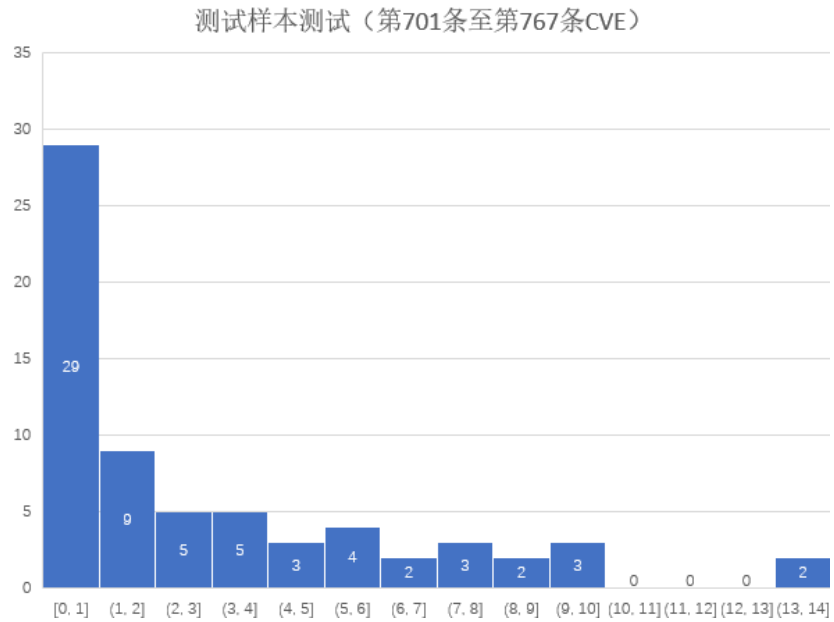


图 4-19 测试样本的测试结果误差分布图（Pybrain 库）

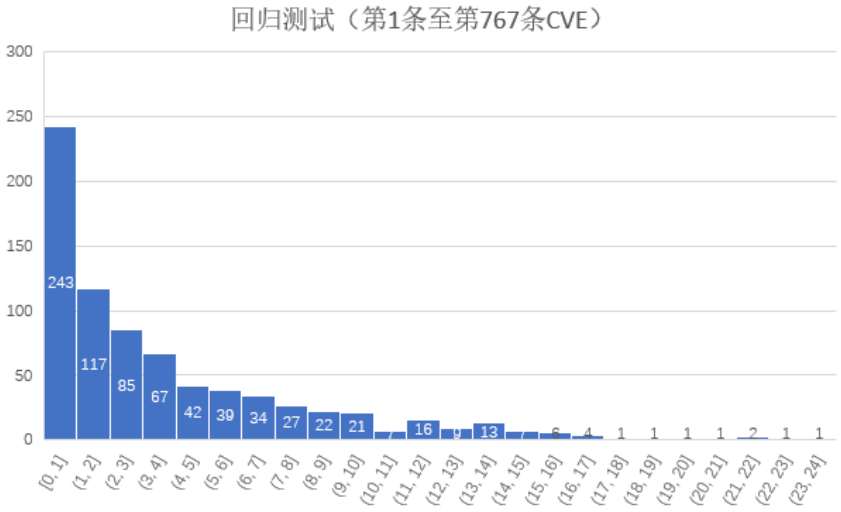


图 4-20 回归测试的测试结果误差分布图（Pybrian 库）

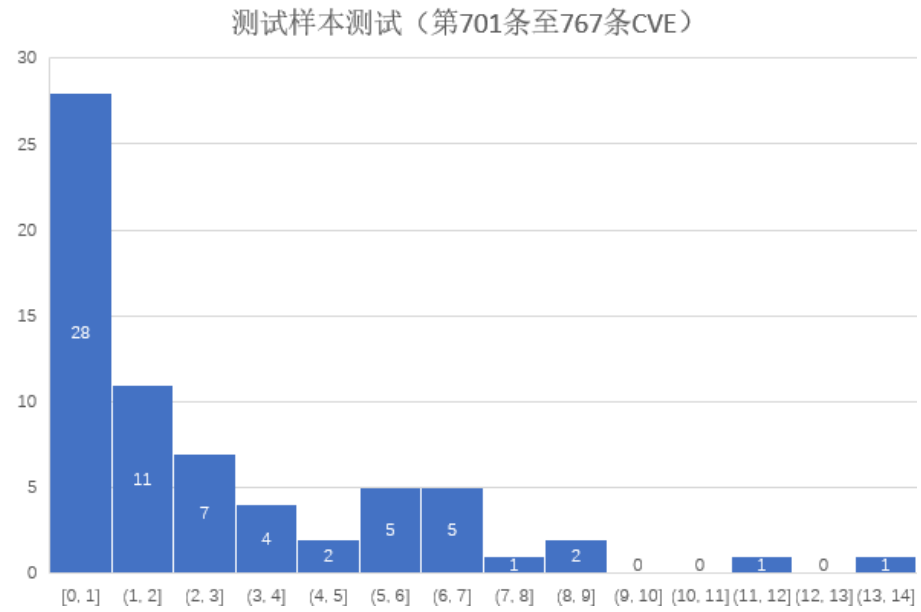


图 4-21 测试样本测试结果误差分布图（自己的而非 Pybrain 库的）

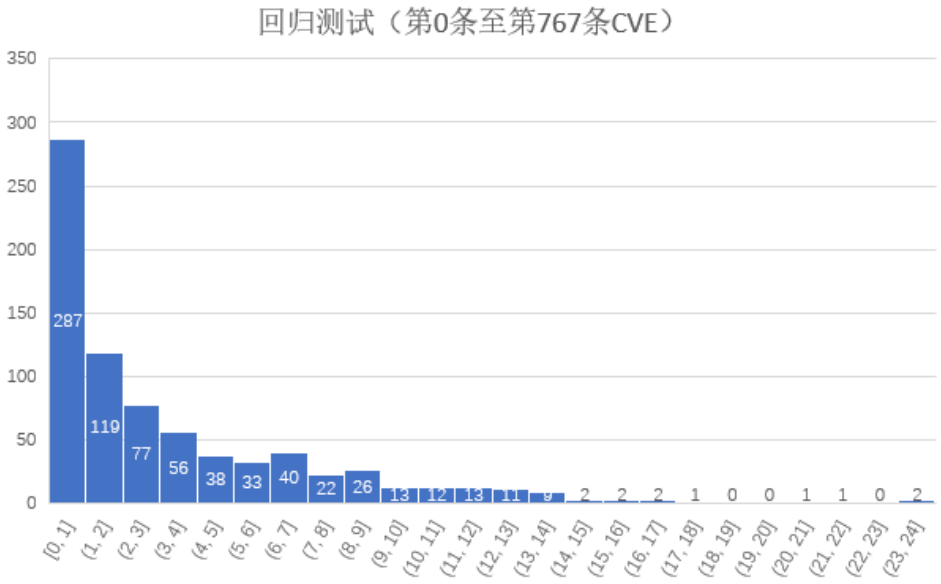


图 4-22 回归测试的测试结果误差分布图（自己的而非 Pybrain 库的）

根据神经网络训练和测试的结果，本人认为准确度达到了基本的期望，可以用于预测输出。根据 Linux 内核漏洞的感知结果，我们将预测的输入设置为 CWE-119、CWE-189、CWE-20、CWE-200、CWE-264、CWE-362、CWE-399、CWE-416 的独热表示和高危害程度（值表示为 1，表示最高的危害程度），得到预测的结果分别为 1 个月、26 个月、6 个月、266 个月、4 个月、57 个月、20 个月、18 个月。如果采用平均的方法则结果为 49.75 个月，即 4.14 年。如果采用加权平均的方法（上述类型漏洞总数量的比例分别为 18%，10%，15%，14%，15%，8%，18%，2%，以此为权重），则结果为 50.04 个月，即 4.17 年

当本人把高危害程度定义为值等于 0.7、0.8、0.9、1（其中 1 表示最高的危害程度）时，得到的预测结果如表 4-3。预测结果的平均值为 4.1875 年（对“4 个高危害程度值*8 个漏洞类型=32 个预测值”求平均）。如果先对每一个高危害程度值的对应结果（8 个漏洞类型的结果）求加权平均，那么不同高危害程度（值为 0.7、0.8、0.9、1.0）的预测结果分别为 4.83 年、4.25 年、4.16 年、4.17 年，求平均为 4.35 年。如果根据不同高危害程度以及漏洞类型进行加权平均（计数见表 4-4），则结果为 22.4 个月，即 1.87 年（相对于 4 年有明显差距，原因是时间间隔长的漏洞数量较少，时间间隔短的漏洞数量较多）。

在输入设计的过程中，可以根据实际情况来优化，那么得出的结果将更可靠。在输出的选择、分析过程中，可以根据实际情况采用更佳的权重来进行加权平均。

在本毕业设计中，在抛开 Linux 内核各版本之间的差异的情况下，Linux 内核需要 2 到 4 年左右的时间来实现稳定化过程，即 2 到 4 年为 Linux 稳定化过程的所需时长的参考。

表 4-3 Linux 内核关键漏洞类型的不同高危害程度的漏洞的最早存在到被发现时间间隔预测表

Score(Normalization)	CWE-119	CWE-189	CWE-20	CWE-200	CWE-264	CWE-362	CWE-399	CWE-416
1	1	26	6	266	4	57	20	18
0.9	1	26	6	266	23	25	20	0
0.8	1	36	6	266	23	25	20	0
0.7	1	36	36	266	1	25	46	55

表 4-4 Linux 内核关键漏洞类型的不同高危害程度数量分布表

Score(Normalization)	CWE-119	CWE-189	CWE-20	CWE-200	CWE-264	CWE-362	CWE-399	CWE-416
1	9	2	3	0	1	0	0	3
0.9	8	1	0	0	8	1	0	2
0.8	3	1	1	0	0	0	0	0
0.7	77	31	37	3	45	17	40	16

4.5 本章小结

本章按毕业设计实现过程中的先后顺序对各项成果进行展示，期望能使读者有一个直观的了解。本章得出了漏洞态势感知的结果并最终得到了 Linux 内核稳定化过程所需时长的基本参考值，可以用于结论的得出。

5 论文结论

本章展示了本毕业设计最终得到的结论和意义，说明本毕业设计的价值。

5.1 结论

本毕业设计实现了爬虫工具、漏洞检测工具、神经网络预测工具、其他辅助工具，以此来得出最终的实验结果，这些工具和结果对于相关研究的人员可以起到很好的参考作用，同时这些工具和内容通过细节部分的修改能够很好的应用于其他软件，并对这些软件的漏洞态势进行感知和对这些软件的稳定化过程时长进行预测。以下是满足第一章提出的假设约束下的漏洞态势感知和预测的结论：

在漏洞态势感知方面。首先，根据图 4-14，我们可以发现高危害漏洞的发现数量随着年份增加而波动上升，而中危害漏洞的发现数量先增加后趋于稳定，在 37%左右，低危害漏洞的发现数量比例较低且数量非常稳定，在 10%左右。其次，根据第四章的结果，我们可以发现 CWE-119、CWE-189、CWE-20、CWE-200、CWE-264、CWE-362、CWE-399、CWE-416 类型的漏洞是 Linux 内核的关键的危害影响程度最大的漏洞。

在漏洞态势预测方面，对于我采用的关键漏洞类型的不同高危害程度的漏洞最早存在到被发现时间间隔的预测如表 4-3。本论文的关键结论为：在满足第一章的假设和约束的条件下，本文认为 Linux 内核的稳定化过程所需要的时长为 2 到 4 年左右。该结论起到的参考意义如下：

- 在基于 Linux 内核定制操作系统时，开发人员可以选取 2 到 4 年前发布的 Linux 内核作为参考。
- 在 Linux 内核测试过程中，2 到 4 年可以作为 Linux 内核关键的测试阶段时长的参考。

5.2 建议

由于本人的时间和水平的约束，本论文还存在许多问题值得进一步探讨。未来的实践过程中可以对本论文中尤其是假设和约束部分进行改进。此外，在第四章实验结果中，对于数据的细节处理可以根据实际情况和反复实验来得出更佳的结论。

5.3 本章小结

本章主要阐述了在特定的前提假设下所得出的 Linux 内核漏洞态势感知结果和稳定化时间的参考值，以及对实际问题的解答和贡献，最后提出了本研究可以改进的方向。

参考文献

[1] Kuhn D R, Raunak M S, Kacker R. An Analysis of Vulnerability Trends, 2008-2016[C]. IEEE International Conference on Software Quality, Reliability and Security Companion. IEEE, 2017:587-588.

[2] Slaby J, Strejček J, Trtík M. ClabureDB: Classified Bug-Reports Database[M]. Verification, Model Checking, and Abstract Interpretation. Springer Berlin Heidelberg, 2013:268-274.

[3] Lee S C, Davis L B. Learning from experience: operating system vulnerability trends[J]. It Professional, 2003, 5(1):17-24.

[4] Grieco G, Grinblat G L, Uzal L, et al. Toward Large-Scale Vulnerability Discovery using Machine Learning[C]. ACM Conference on Data and Application Security and Privacy. ACM, 2016:85-96.

[5] Abal I, Brabrand C, Wasowski A. 42 variability bugs in the linux kernel: a qualitative analysis[C]. Acm/ieee International Conference on Automated Software Engineering. 2014:421-432.

[6] Woo M, Sang K C, Gottlieb S, et al. Scheduling black-box mutational fuzzing[C]. ACM Sigsac Conference on Computer & Communications Security. ACM, 2013:511-522.

[7] Homaei H, Shahriari H R. Seven Years of Software Vulnerabilities: The Ebb and Flow[J]. IEEE Security & Privacy, 2017, 15(1):58-65.

[8] Chang Y Y, Zavarsky P, Ruhl R, et al. Trend Analysis of the CVE for Software Vulnerability Management[C]. IEEE Third International Conference on Privacy, Security, Risk and Trust. IEEE, 2012:1290-1293.

[9] Rick Kuhn, Chris Johnson. Vulnerability Trends: Measuring Progress[J]. It Professional, 2010, 12(4):51-53.

[10] 何晶. 基于 WooYun 的视听新媒体网站漏洞统计分析[J]. 电视技术, 2014, 38(16):65-69.

[11] 佚名. 赛门铁克互联网安全威胁报告[J]. 软件和集成电路, 2003(6):64-64.

[12] Ullah N, Morisio M, Vetrò A. Selecting the Best Reliability Model to Predict Residual Defects in Open Source Software[J]. Computer, 2015, 48(6):50-58.

[13] Okamura H, Dohi T. Towards comprehensive software reliability evaluation in open source software[C]. IEEE, International Symposium on Software Reliability Engineering. IEEE Computer Society, 2015:121-129.

[14] Rahimi S, Zargham M. Vulnerability Scrying Method for Software Vulnerability Discovery Prediction Without a Vulnerability Database[J]. IEEE Transactions on Reliability, 2013, 62(2):395-407.

致　　谢

本论文的工作是在我的指导老师何永忠教授的悉心指导下完成，何永忠老师严谨的治学态度和科学的工作方法给了我极大的帮助和影响，在此由衷地感谢何永忠老师给予的关心和指导。

刘嘉乐学姐和梁迪学长在我的工作过程中给予了热情的帮助，在此向他们表达我的感激之情。

附　　录

附录 A 程序代码

详情请见见毕业设计的附件，此处贴上 22 个 python 文件中的 3 个。

BPNNByEpoch.py

说明：该 Python 代码文件内容为所编写的 BP 神经网络。

代码如下：

```
# Author: 14281055 Liheng Chen CIT BJTU
# File Name: BPNNByEpoch.py

import numpy
import os
import Repository


class BPNeuralNetwork(object):
    def sigmoid(self, x, derivation=False):
        if derivation:
            return self.sigmoid(x) * (1 - self.sigmoid(x))
        else:
            return 1 / (1 + numpy.exp(-x))

    input_data = None
    target_data = None

    layer_input_list = None
    layer_output_list = None

    layer_bias = None

    layer_number = None
    layer_node_number_list = None

    weight_list = []
    correction_matrix_list = []

    layer_output_delta_list = None
    layer_error_list = None

    activation_func = sigmoid

    epochs = 100000000
```

```
goal = 0.00001
learn_rate = 0.08    # usually in (0,0.1); annealing
correct_rate = 0.008    # annealing

loss = None

weight_file_path = None

def __init__(self, layer_node_number_list, input_data, target_data=None, weight_file_path=None):
    self.layer_node_number_list = layer_node_number_list

    self.input_data = input_data
    self.target_data = target_data

    self.weight_file_path = weight_file_path

    self.layer_number = len(self.layer_node_number_list)

    self.layer_bias = [0] * (self.layer_number - 1)

    self.layer_input_list = [numpy.array(None)] * self.layer_number
    self.layer_output_list = [numpy.array(None)] * self.layer_number

    self.layer_output_delta_list = [numpy.array(None)] * self.layer_number
    self.layer_error_list = [numpy.array(None)] * self.layer_number

    for layer_index in range(self.layer_number - 1):
        self.weight_list.append(numpy.random.randn(
            layer_node_number_list[layer_index],
            layer_node_number_list[layer_index + 1]
        ) / numpy.sqrt(self.layer_node_number_list[layer_index]))
        self.correction_matrix_list.append(numpy.zeros(self.weight_list[layer_index].shape))
    if self.weight_file_path is not None and os.path.exists(self.weight_file_path):
        self.weight_list = numpy.load(self.weight_file_path)

def train(self):
    for epoch in range(self.epochs):
        self.loss = 0
```

```
            for index in range(self.input_data.shape[0]):
                self.forward_propagation(
                    numpy.atleast_2d(self.input_data[index])
                )
                self.back_propagation(numpy.atleast_2d(self.target_data[index]))
                self.loss += numpy.square(
                    numpy.atleast_2d(self.target_data[index]) - self.layer_output_list[-1]).sum() / 2
            if self.weight_file_path is not None and epoch % 100 == 0:
                numpy.save(os.path.splitext(self.weight_file_path)[0], self.weight_list)
            self.annealing(epoch)
            print('\rEpoch:%d Loss:%f' % (epoch, self.loss), end='')
            if self.loss < self.goal:
                break

    def test(self):
        self.forward_propagation(self.input_data)
        return self.layer_output_list[-1]

    def forward_propagation(self, input_data):
        self.layer_input_list[0] = input_data
        self.layer_output_list[0] = input_data
        for layer_index in range(1, self.layer_number):
            self.layer_input_list[layer_index] = \
                self.layer_output_list[layer_index - 1].dot(self.weight_list[layer_index - 1]) + \
                self.layer_bias[layer_index - 1]
            self.layer_output_list[layer_index] = self.activation_func(self.layer_input_list[layer_index])

    def back_propagation(self, target_data):
        self.layer_output_delta_list[-1] = target_data - self.layer_output_list[-1]
        self.layer_error_list[-1] = \
            self.layer_output_delta_list[-1] * \
            self.activation_func(
                self.layer_input_list[-1],
                derivation=True
            )
        self.weight_list[-1] += self.learn_rate * self.layer_output_list[-2].T.dot(
            self.layer_error_list[-1]) + self.correct_rate * self.correction_matrix_list[-1]
        self.correction_matrix_list[-1] = self.layer_output_list[-2].T.dot(self.layer_error_list[-1])
```

```
        for layer_index in range(-2, -self.layer_number, -1):
            self.layer_output_delta_list[layer_index] = \
                self.layer_error_list[layer_index + 1].dot(self.weight_list[layer_index + 1].T)
            self.layer_error_list[layer_index] = \
                self.layer_output_delta_list[layer_index] * \
                self.activation_func(
                    self.layer_input_list[layer_index],
                    derivation=True
                )
            self.weight_list[layer_index] += \
                self.learn_rate * self.layer_output_list[layer_index - 1].T \
                    .dot(self.layer_error_list[layer_index]) \
                + self.correct_rate * self.correction_matrix_list[layer_index]
            self.correction_matrix_list[layer_index] = self.layer_output_list[layer_index - 1].T \
                .dot(self.layer_error_list[layer_index])

    def annealing(self, epoch, step=100000):
        if epoch % step == 0:
            self.learn_rate /= 2    # learn rate annealing
            self.correct_rate /= 2


def train(layer_node_number_list, train_input_file_path, train_target_file_path, weight_file_path=None,
          target_one_hot_flag=False):
    if target_one_hot_flag:
        train_target_one_hot_file_path = Repository.add_suffix_to_file_name(train_target_file_path,
                                                                            '_one_hot')
        if not os.path.exists(train_target_one_hot_file_path):
            Repository.save_one_hot(
                train_target_file_path,
                layer_node_number_list[-1],
                train_target_one_hot_file_path
            )
        train_target_file_path = train_target_one_hot_file_path
    bp_neural_network = BPNeuralNetwork(
        layer_node_number_list,
        Repository.excel_to_np_array(train_input_file_path),
        Repository.excel_to_np_array(train_target_file_path),
```

```
        weight_file_path
    )
    bp_neural_network.train()


def test(layer_node_number_list, test_input_file_path, test_predict_file_path, weight_file_path,
         target_one_hot_flag=False):
    bp_neural_network = BPNeuralNetwork(
        layer_node_number_list,
        Repository.excel_to_np_array(test_input_file_path),
        weight_file_path=weight_file_path
    )
    if target_one_hot_flag:
        test_predict_one_hot_file_path = Repository.add_suffix_to_file_name(test_predict_file_path,
                                                                            '_one_hot')
        Repository.np_array_to_excel(bp_neural_network.test(), test_predict_one_hot_file_path)
        Repository.save_one_hot_like_reversal(test_predict_one_hot_file_path, test_predict_file_path)
    else:
        Repository.np_array_to_excel(bp_neural_network.test(), test_predict_file_path)


if __name__ == '__main__':
    train(
        [125, 25, 300],
        r'C:\Users\陈力恒\Downloads\VulnerabilitySituation\Backup\Data\train_input_normal_l.xlsx',
        r'C:\Users\陈力恒\Downloads\VulnerabilitySituation\Backup\Data\train_target.xlsx',
        r'C:\Users\陈力恒\Downloads\VulnerabilitySituation\Backup\Backup3(My)\weight125_25_300.npy',
        target_one_hot_flag=True
    )
    test(
        [125, 25, 300],
        # r'C:\Users\陈力恒\Downloads\VulnerabilitySituation\Backup\Data\test_input_normal_l.xlsx',
        r'C:\Users\陈力恒\Downloads\VulnerabilitySituation\Backup\Data\predict_input_normal.xlsx',
        # r'C:\Users\陈力恒\Downloads\VulnerabilitySituation\Backup\Backup3(My)\test_predict.xlsx',
        r'C:\Users\陈力恒\Downloads\VulnerabilitySituation\Backup\Backup3(My)\predict_output.xlsx',
        r'C:\Users\陈力恒\Downloads\VulnerabilitySituation\Backup\Backup3(My)\weight125_25_300.npy',
```

```
        target_one_hot_flag=True
    )
```

GetModuleDiffIn.py

说明：该 Python 代码文件内容主要为代码定位工具。

代码如下：

```
# Author: 14281055 Liheng Chen CIT BJTU
# File Name: GetModuleDiffIn.py
import Repository
import re
import os


def match_line(module_info_list, line):
    line = Repository.separate_word_and_nonword(line)
    if module_info_list[0] == 'struct':
        return True if re.search(r'\bstruct ' + re.escape(module_info_list[1]) + ' {', line) else False
    elif module_info_list[0] == 'function':
        return True if re.search(
            r'\b' + re.escape(module_info_list[1]) + ' ' + re.escape(module_info_list[2]) + r' \(',
            line) else False
    return None


def match_lines(module_info_list, line_list, default_line_index_range=30):
    if match_line(module_info_list, line_list[0]):
        return True
    line_index_range = default_line_index_range if len(line_list) > default_line_index_range else len(
        line_list)
    lines = Repository.line_list_to_lines(line_list[:line_index_range])
    first_line_end_index = Repository.get_first_line_end_index_in_lines(line_list[0], lines)
    if module_info_list[0] == 'struct':
        match = re.search(r'\bstruct ' + re.escape(module_info_list[1]) + ' {', lines)
        if match:
            tag_end_index = match.start() + 5    # length of 'struct' is 6
            if tag_end_index <= first_line_end_index:
                return True
        return False
    elif module_info_list[0] == 'function':
        match = re.search(
            r'\b' + re.escape(module_info_list[1]) + ' (?:\* )*' + re.escape(
```

```
                module_info_list[2]) + r' \([^)]*\) {',
            lines)
        if match:
            tag_end_index = match.start() + len(module_info_list[1]) - 1
            if tag_end_index <= first_line_end_index:
                return True
        return False
    return None


def get_module_info_list(line_list):
    lines = Repository.line_list_to_lines(line_list)
    first_line_end_index = Repository.get_first_line_end_index_in_lines(line_list[0], lines)
    match = re.search(r'\bstruct (\w+) {', lines)
    if match:
        tag_end_index = match.start() + 5    # length of 'struct' is 6
        if tag_end_index <= first_line_end_index:
            return ['struct', match.group(1)]

    match = re.search(r'(?!if|while)(\w+) (?:\* )*(?!if|while)(\w+) \([^)]*\) {', lines)
    if match:
        tag_end_index = match.start() + len(match.group(1)) - 1
        if tag_end_index <= first_line_end_index:
            return ['function', match.group(1), match.group(2)]
    return None


def read_module_and_scope(module_file_path, line_index):
    line = Repository.get_file_line_list(module_file_path)[line_index]
    if line.strip() == '':
        return []
    else:
        module_and_scope = line.strip().split('\t')
        module_and_scope[-2] = int(module_and_scope[-2])
        module_and_scope[-1] = int(module_and_scope[-1])
        return module_and_scope
```

```
def read_module_and_scope_list(module_file_path):
    module_file_line_list = Repository.get_file_line_list(module_file_path)
    module_and_scope_list = []
    if module_file_line_list:
        for line in module_file_line_list:
            module_and_scope = line.strip().split('\t')
            module_and_scope[-2] = int(module_and_scope[-2])
            module_and_scope[-1] = int(module_and_scope[-1])
            module_and_scope_list.append(module_and_scope)
    return module_and_scope_list


# def get_module_info_list_list(dst_file_path, default_line_list_range=30):
#     dst_file_line_list = Repository.get_file_line_list(dst_file_path)
#     module_info_list_list = []
#     for line_index, line in enumerate(dst_file_line_list):
#         line_list_range = default_line_list_range if len(
#             dst_file_line_list) - line_index > default_line_list_range else len(
#             dst_file_line_list) - line_index
#         module = get_module_info_list(dst_file_line_list[line_index:line_index + line_list_range])
#         if module:
#             module_info_list_list.append(module)
#     return module_info_list_list


def get_module_and_scope_list(dst_file_path, default_line_list_range=30, write_flag=False):
    module_file_path = Repository.add_prefix_to_file_name(dst_file_path, '(Module)')
    is_module_file_exist = True if os.path.exists(module_file_path) else False
    dst_file_line_list = Repository.get_file_line_list(dst_file_path)
    module_and_scope_list = []
    for line_index, line in enumerate(dst_file_line_list):
        line_list_range = default_line_list_range if len(
            dst_file_line_list) - line_index > default_line_list_range else len(
            dst_file_line_list) - line_index
        module = get_module_info_list(dst_file_line_list[line_index:line_index + line_list_range])
        if module:
            scope = get_module_scope(module, dst_file_path, line_index)
            if not scope:
```

```
                scope = [None, None]
            module_and_scope = module + scope
            module_and_scope_list.append(module_and_scope)
            if write_flag and not is_module_file_exist:
                line = '\t'.join(module_and_scope[:-2]) + '\t' + str(module_and_scope[-2]) + '\t' + str(
                    module_and_scope[-1])
                Repository.append_file_with_eol(module_file_path, line)
    return module_and_scope_list


def print_module_and_scope(file_path):
    for module_and_scope in get_module_and_scope_list(file_path):
        print('\t'.join(module_and_scope[:-2]) + '\t' + str(module_and_scope[-2]) + '\t' + str(
            module_and_scope[-1]))


def write_module_and_scope(module_file_path, module_and_scope=None, flag=False):
    if flag:
        line = '\t'.join(module_and_scope[:-2]) + '\t' + str(module_and_scope[-2]) + '\t' + str(
            module_and_scope[-1])
        Repository.append_file_with_eol(module_file_path, line)
    else:
        Repository.append_file_with_eol(module_file_path, '')


def write_module_file(dst_file_path):
    module_file_path = Repository.add_prefix_to_file_name(dst_file_path, '(Module)')
    if not os.path.exists(module_file_path):
        get_module_and_scope_list(dst_file_path, write_flag=True)


def write_module_diff_file(diff_file_path):
    module_diff_file_path = Repository.add_prefix_to_file_name(diff_file_path, '(Module)')
    if not os.path.exists(module_diff_file_path):
        bm_file_path = Repository.add_prefix_to_file_name(diff_file_path, '(BM)')
        write_module_file(bm_file_path)
        for diff_segment in get_diff_segment_list(diff_file_path):
            for module_and_scope in read_module_and_scope_list(
```

```
                    Repository.add_prefix_to_file_name(bm_file_path, '(Module)')):
                if module_and_scope[-2] != 'None' and module_and_scope[-1] != 'None' \
                        and Repository.is_lines_in_lines(diff_segment[4],

Repository.get_file_line_list(bm_file_path)
                                                                [module_and_scope[-2]:
module_and_scope[-1] + 1]):
                    write_module_and_scope(module_diff_file_path, module_and_scope, True)
                    break
            else:
                write_module_and_scope(module_diff_file_path)


def is_diff_segment_in_lines(diff_segment, dst_line_list, has_module_info=True):
    if not has_module_info and diff_segment[1] == []:
        return Repository.unknown
    else:
        return Repository.true if Repository.is_lines_in_lines(diff_segment[1], dst_line_list) \
                                  and not Repository.is_lines_in_lines(diff_segment[2],
                                                                       dst_line_list)  else
Repository.false


def get_module_scope(module, dst_file_path, line_offset=0):
    dst_file_line_list = Repository.get_file_line_list(dst_file_path)
    for line_index in range(len(dst_file_line_list[line_offset:])):
        if match_lines(module, dst_file_line_list[line_offset + line_index:]):
            start_line_index = line_offset + line_index
            break
    else:
        return []
    remain_lines = Repository.line_list_to_lines(dst_file_line_list[start_line_index:])
    remain_line_end_char_index_dict = Repository.get_line_end_char_index_dict_with_lines(
        dst_file_line_list[start_line_index:],
        remain_lines)
    opening_brace_number = 0
    closing_brace_number = 0
    match = re.search(re.escape(module[-1]) + '.*?{', remain_lines)
```

```
    if match:
        # opening_brace_line_index = Repository.get_line_index(remain_line_end_char_index_dict,
        #                                                       match.end() - 1) +
start_line_index
        opening_brace_number += 1
    else:
        return []
    closing_brace_line_index = len(dst_file_line_list)
    for char_index, character in enumerate(remain_lines[match.end():]):
        if character == '{':
            opening_brace_number += 1
        elif character == '}':
            closing_brace_number += 1
            if opening_brace_number == closing_brace_number:
                closing_brace_line_index = Repository.get_line_index(
                    remain_line_end_char_index_dict,
                    char_index + match.end()
                ) + start_line_index
                break
    return [start_line_index, closing_brace_line_index]


def get_diff_segment_list(diff_file_path):
    diff_file_line_list = Repository.get_file_line_list(diff_file_path)
    diff_segment_list = []
    diff_segment = []
    diff_segment_delete_list = []
    diff_segment_add_list = []
    diff_segment_other_list = []
    diff_segment_non_add_list = []
    has_diff_segment_head = False
    for diff_file_line in diff_file_line_list:
        match = re.search(r'@@.*?@@(.*)', diff_file_line)
        if match:
            if diff_segment:
                diff_segment.append(diff_segment_delete_list)
                diff_segment.append(diff_segment_add_list)
                diff_segment.append(diff_segment_other_list)
```

```
                    diff_segment.append(diff_segment_non_add_list)
                    diff_segment_list.append(diff_segment)
                    diff_segment_delete_list = []
                    diff_segment_add_list = []
                    diff_segment_other_list = []
                    diff_segment_non_add_list = []
                    diff_segment = []
                diff_segment_head = match.group(1).strip()
                diff_segment.append(diff_segment_head)
                has_diff_segment_head = True
            else:
                if has_diff_segment_head and diff_file_line.strip() != '':
                    match = re.match(r'\+(.*)', diff_file_line)
                    if match:
                        diff_segment_add_list.append(match.group(1).strip())
                    elif re.match(r'[^-+]', diff_file_line):
                        diff_segment_other_list.append(diff_file_line.strip())
                        diff_segment_non_add_list.append(diff_file_line.strip())
                    else:
                        match = re.match(r'-(.*)', diff_file_line)
                        if match:
                            diff_segment_delete_list.append(match.group(1).strip())
                            diff_segment_non_add_list.append(match.group(1).strip())
        if diff_segment:
            diff_segment.append(diff_segment_delete_list)
            diff_segment.append(diff_segment_add_list)
            diff_segment.append(diff_segment_other_list)
            diff_segment.append(diff_segment_non_add_list)
            diff_segment_list.append(diff_segment)
        return diff_segment_list

# def write_diff_module_and_scope_tuple(diff_file_path):
#                     bm_file_path = os.path.join(os.path.dirname(diff_file_path), '(BM)' +
os.path.basename(diff_file_path))
#                     save_path = os.path.join(os.path.dirname(diff_file_path), '(Module)' +
os.path.basename(diff_file_path))
#         if os.path.exists(save_path):
#             os.remove(save_path)
```

```
#         bm_file_line_list = Repository.get_file_line_list(bm_file_path)
#         for diff in get_diff_list(diff_file_path):
#             diff_module_and_scope_tuple = get_diff_module_and_scope_tuple(diff[1:], bm_file_line_list)
#             string = '\t'.join(diff_module_and_scope_tuple[0])
#             string += '\t'
#             string += '\t'.join(
#                 [str(diff_module_and_scope_tuple[1]), str(diff_module_and_scope_tuple[2])]
#             )
#             Repository.append_file_with_eol(save_path, string)
#
#
# def write_all_diff_module_and_scope_tuple(patch_root):
#         for patch_dir_name in os.listdir(patch_root):
#             patch_dir_path = os.path.join(patch_root, patch_dir_name)
#             if os.path.isdir(patch_dir_path):
#                 for file_name in os.listdir(patch_dir_path):
#                     if not re.match(r'\(', file_name) and not re.match(r'Source\.txt', file_name):
#                         write_diff_module_and_scope_tuple(os.path.join(patch_dir_path, file_name))
```

VersionNumberAndExistTime.py

说明：该 Python 代码文件内容主要为漏洞检测工具。

代码如下：

```
# Author: 14281055 Liheng Chen CIT BJTU
# File Name: VersionNumberAndExistTime.py


import xlrd
import os
import re
import datetime
import Repository
import CharacterFactor
import GetModuleDiffIn


def is_diff_segment_in_file(diff_segment, diff_segment_index, dst_file_path, diff_file_path):
    bm_file_path = os.path.join(os.path.dirname(diff_file_path),
                                '(BM)' + os.path.basename(diff_file_path))
    if os.path.splitext(diff_file_path)[1] in ('.c', '.h') and os.path.exists(bm_file_path):
        GetModuleDiffIn.write_module_diff_file(diff_file_path)
        module_diff_file_path = os.path.join(os.path.dirname(diff_file_path),
                                             '(Module)' + os.path.basename(diff_file_path))
        diff_module_and_scope = GetModuleDiffIn.read_module_and_scope(module_diff_file_path,

diff_segment_index)
        if diff_module_and_scope:
            scope_list = GetModuleDiffIn.get_module_scope(diff_module_and_scope[:-2],
dst_file_path)
            if scope_list:
                return GetModuleDiffIn.is_diff_segment_in_lines(
                    diff_segment,
                    Repository.get_file_line_list(dst_file_path)[scope_list[0]:scope_list[1] + 1]
                )
            else:
                return False
    return GetModuleDiffIn.is_diff_segment_in_lines(diff_segment,
```

```
Repository.get_file_line_list(dst_file_path), False)


def is_one_vuln_in_linux_kernel_accurate(diff_file_name, patch_dir, kernel_dir):
    vuln_relative_path = diff_file_name.replace('~!@#', '\\')
    vuln_dir_path = os.path.join(kernel_dir, os.path.dirname(vuln_relative_path))
    if os.path.exists(vuln_dir_path):
        for file_name in os.listdir(vuln_dir_path):
            if os.path.isdir(os.path.join(vuln_dir_path, file_name)):
                continue
            elif re.match(
                    re.escape(os.path.splitext(os.path.basename(vuln_relative_path))[0])
                    + r'(_[1-5])?'
                    + re.escape(os.path.splitext(os.path.basename(vuln_relative_path))[1]),
                    file_name, re.I):
                for diff_segment_index, diff_segment in enumerate(
                        GetModuleDiffIn.get_diff_segment_list(os.path.join(patch_dir,
diff_file_name))):
                    if not is_diff_segment_in_file(diff_segment, diff_segment_index,
                                                   os.path.join(vuln_dir_path, file_name),
                                                   os.path.join(patch_dir, diff_file_name)):
                        break
                else:
                    return True
    return False


# def is_one_vuln_in_linux_kernel(patch_name, patch_dir, kernel_dir):
#     patch_relative_path = patch_name.replace('~！@#', '\\')
#     patch_relative_path_dir_name = os.path.dirname(patch_relative_path)
#     patch_relative_path_base_name = os.path.basename(patch_relative_path)
#     vuln_path_dir_name = os.path.join(kernel_dir, patch_relative_path_dir_name)
#     if not os.path.exists(vuln_path_dir_name):
#         return False
#     for file_name in os.listdir(vuln_path_dir_name):
#         if os.path.isdir(os.path.join(vuln_path_dir_name, file_name)):
#             continue
#         if os.path.splitext(patch_relative_path_base_name)[0] in file_name:
```

```
#                     with open(os.path.join(patch_dir, patch_name), 'r') as patch_file:
#                         for line in patch_file.readlines():
#                             match = re.match(r'-([^-].*)', line)
#                             if match and not Repository.is_string_in_file(
#                                     match.group(1).strip(),
#                                     os.path.join(vuln_path_dir_name, file_name)
#                             ):
#                                 break
#                         else:
#                             return True
#         return False

def is_one_vuln_in_linux_kernel_path(diff_file_name, kernel_dir):
    vuln_relative_path = diff_file_name.replace('~!@#', '\\')
    vuln_dir_path = os.path.join(kernel_dir, os.path.dirname(vuln_relative_path))

    if os.path.exists(vuln_dir_path):
        for file_name in os.listdir(vuln_dir_path):
            if os.path.isdir(os.path.join(vuln_dir_path, file_name)):
                continue
            elif re.match(
                    re.escape(os.path.splitext(os.path.basename(vuln_relative_path))[0])
                    + r'(_[1-5])?'
                    + re.escape(os.path.splitext(os.path.basename(vuln_relative_path))[1]),
                    file_name, re.I):
                return True
    return False


def is_vuln_in_linux_kernel_path(patch_dir, kernel_dir):
    for file_name in os.listdir(patch_dir):
        if file_name != 'Source.txt' and not re.match(r'\(', file_name):
            if not is_one_vuln_in_linux_kernel_path(file_name, kernel_dir):
                return False
    return True


def is_vuln_in_linux_kernel(patch_dir, kernel_dir):
```

```
    if Repository.is_dir_empty(patch_dir) or not is_vuln_in_linux_kernel_path(patch_dir, kernel_dir):
        return False
    for file_name in os.listdir(patch_dir):
        if file_name != 'Source.txt' and not re.match(r'\(', file_name):
            if not is_one_vuln_in_linux_kernel_accurate(file_name, patch_dir, kernel_dir):
                return False
    return True


def print_current_search(kernel_name, rate_str):
    print(rate_str + '\tSearch:Linux-' + get_version_number(kernel_name), end='')


def get_version_number(kernel_name):
    match = re.match(r'linux-(\d+(\.\d+)*)', kernel_name, re.I)
    if match:
        return match.group(1)
    else:
        return None


# def get_oldest_version_number_binary_search(patch_dir, kernel_root, rate_str):
#     kernel_name_list = sorted(os.listdir(kernel_root),
#                               key=Repository.cmp_to_key(Repository.kernel_name_compare))
#     low_index = 0
#     high_index = len(kernel_name_list) - 1
#     last_exist_index = None
#     while True:
#         if low_index == high_index:
#             print_current_search(kernel_name_list[low_index], rate_str)
#             if (last_exist_index and low_index == last_exist_index) \
#                     or is_vuln_in_linux_kernel(patch_dir, kernel_name_list[low_index]):
#                 return get_version_number(kernel_name_list[low_index])
#             else:
#                 return None
#         middle_index = int((low_index + high_index) / 2)
#         print_current_search(kernel_name_list[middle_index], rate_str)
#         if (last_exist_index and middle_index == last_exist_index) \
```

```
#                       or is_vuln_in_linux_kernel(patch_dir,
#                                                                                   get_kernel_dir(kernel_root,
kernel_name_list[middle_index])):
#                   last_exist_index = middle_index
#                   high_index = middle_index
#               else:
#                   low_index = middle_index + 1


def get_kernel_dir_path(kernel_root_path, kernel_name):
    kernel_dir_path = os.path.join(kernel_root_path, kernel_name)
    for file_name in os.listdir(kernel_dir_path):
        if re.match('linux', file_name, re.I) and os.path.isdir(file_name):
            return os.path.join(kernel_dir_path, file_name)
    return kernel_dir_path


def get_linux_kernel_version_number_list(path):
    r_workbook = xlrd.open_workbook(path)
    r_sheet = r_workbook.sheet_by_index(0)
    version_number_list = []
    for row_index in range(1, r_sheet.nrows):
        version_number = str(r_sheet.cell(row_index, 0).value).strip()
        version_number_list.append(version_number)
    return version_number_list


def get_linux_kernel_name_list(path):
    r_workbook = xlrd.open_workbook(path)
    r_sheet = r_workbook.sheet_by_index(0)
    kernel_name_list = []
    for row_index in range(1, r_sheet.nrows):
        version_number = str(r_sheet.cell(row_index, 0).value).strip()
        kernel_name = 'linux-' + version_number
        kernel_name_list.append(kernel_name)
    return kernel_name_list
```

```
def get_oldest_version_number(patch_dir, kernel_root, linux_kernel_release_time_file_path, rate_str):
    for kernel_name in get_linux_kernel_name_list(linux_kernel_release_time_file_path):
        match = re.match(r'linux-(\d+(\.\d+)*)', kernel_name, re.I)
        if match:
            version_number = match.group(1)
            kernel_dir = get_kernel_dir_path(kernel_root, kernel_name)
            if kernel_dir:
                print(rate_str + '\tSearch:Linux-' + version_number, end='')
                if is_vuln_in_linux_kernel(patch_dir, kernel_dir):
                    return version_number
    return None


def get_version_number_and_exist_time_tuple(patch_dir, kernel_root,
                                            linux_kernel_release_time_file_path, rate_str='\r'):
    if os.path.exists(patch_dir):
        oldest_kernel_version_number = get_oldest_version_number(patch_dir, kernel_root,

linux_kernel_release_time_file_path,
                                                                 rate_str)
        if oldest_kernel_version_number:
            release_time_tuple = CharacterFactor.get_linux_kernel_release_time_tuple(
                oldest_kernel_version_number,
                linux_kernel_release_time_file_path
            )
            return oldest_kernel_version_number, release_time_tuple
    return ()


def save_vuln_version_number_and_exist_time(save_path, excel_path, patch_root, kernel_root,
                                            linux_kernel_release_time_file_path,
start_row_index):
    r_workbook = xlrd.open_workbook(excel_path)
    r_sheet = r_workbook.sheets()[0]
    Repository.init_workbook(save_path,
                             ['CVE Number', 'Linux Kernel Version Number', 'Vulnerability Exist
Time'],
                             width=1.5)
```

```
    for row_index in range(start_row_index, r_sheet.nrows):
        cve_id = str(r_sheet.cell(row_index, 0).value).strip()
        rate_str = '\rRow Index:' + str(row_index) + '\t' + cve_id
        print(rate_str, end='')
        info_tuple = get_version_number_and_exist_time_tuple(os.path.join(patch_root, cve_id.upper()),
                                                             kernel_root,
linux_kernel_release_time_file_path,
                                                             rate_str)
        if info_tuple:
            version_number = info_tuple[0]
            if info_tuple[1]:
                release_time = datetime.datetime(*info_tuple[1])
            else:
                release_time = None
        else:
            version_number = None
            release_time = None
        Repository.write_workbook(save_path,
                                  {'CVE Number': cve_id, 'Linux Kernel Version Number':
version_number,
                                   'Vulnerability Exist Time': release_time}, row_index - 1,
                                  style_list=[None, None, Repository.set_date_style()])


if __name__ == '__main__':
    save_vuln_version_number_and_exist_time(
        r'C:\Users\79196\Downloads\NVD\CVE Oldest Version Number.xls',
        r'C:\Users\79196\Downloads\NVD\Linux Kernel Character Factor.xls',
        r'C:\Users\79196\Downloads\NVD\Linux Kernel Patch',
        r'D:\Linux Kernel\All',
        r'C:\Users\79196\Downloads\NVD\Linux Kernel Release Time.xls',
        start_row_index=2,
    )
```

附录 B 外文文献与翻译

42 Variability Bugs in the Linux Kernel: A Qualitative Analysis

Iago Abal
iago@itu.dk

Claus Brabrand
brabrand@itu.dk

Andrzej Wąsowski
wasowski@itu.dk

IT University of Copenhagen
Rued Langgaards Vej 7, 2300 Copenhagen S, Denmark

ABSTRACT

Feature-sensitive verification pursues effective analysis of the exponentially many variants of a program family. However, researchers lack examples of concrete bugs induced by variability, occurring in real large-scale systems. Such a collection of bugs is a requirement for goal-oriented research, serving to evaluate tool implementations of feature-sensitive analyses by testing them on real bugs. We present a qualitative study of 42 variability bugs collected from bug-fixing commits to the Linux kernel repository. We analyze each of the bugs, and record the results in a database. In addition, we provide self-contained simplified C99 versions of the bugs, facilitating understanding and tool evaluation. Our study provides insights into the nature and occurrence of variability bugs in a large C software system, and shows in what ways variability affects and increases the complexity of software bugs.

Categories and Subject Descriptors

D.2.0 [**Software Engineering**]: General; D.2.5 [**Software Engineering**]: Testing and Debugging

Keywords

Bugs; Feature interactions; Linux; Software Variability

1. INTRODUCTION

Many software projects have to cope with a large amount of variability. In projects adopting the Software Product Line methodology [1] variability is used to tailor development of an individual software product to a particular market niche. A related, but different, class of projects develops highly configurable systems, such as the Linux kernel, where configuration options, here referred as *features* [20], are used to tailor functional and non-functional properties to the needs of a particular user. Highly configurable systems can get very large and encompass large sets of features. Reports of industrial systems with thousands of features exist [4] and extensive open-source examples are documented in detail [5].

ASE'14, September 15-19, 2014, Vasteras, Sweden.

ACM 978-1-4503-3013-8/14/09 ...$15.00.
http://dx.doi.org/10.1145/2642937.2642990.

Features in a configurable system interact in non-trivial ways, in order to influence each others functionality. When such interactions are unintended, they induce bugs that manifest themselves in certain configurations but not in others, or that manifest differently in different configurations. A bug in an individual configuration may be found by analyzers based on standard program analysis techniques. However, since the number of configurations is exponential in the number of features, it is not feasible to analyze each configuration separately.

Family-based [33] analyses, a form of feature-sensitive analyses, tackle this problem by considering all configurable program variants as a single unit of analysis, instead of analyzing the individual variants separately. In order to avoid duplication of effort, common parts are analyzed once and the analysis forks only at differences between variants. Recently, various family-based extensions of both classic static analysis [2, 6, 9, 14, 22, 24] and model checking [3, 12, 13, 17, 30] based techniques have been developed.

Most of the research so far has focused on the inherent scalability problem. However, we still lack evidence that these extensions are adequate for specific purposes in real-world scenarios. In particular, little effort has been put into understanding what kind of bugs appear in highly configurable systems, and what are their variability characteristics. Gaining such understanding would help to ground research on family-based analyses in actual problems.

The understanding of complexity of variability bugs is not common among practitioners and in available artifacts. While bug reports abound, there is little knowledge on what of those bugs are caused by feature interactions. Very often, due to the complexities of a large project like Linux, and the lack of feature-sensitive tool support, developers are not entirely conscious of the features that affect the software they work on. As a result, bugs appear and get fixed with little or no indication of their variational program origins.

The objective of this work is to understand the complexity and nature of *variability bugs* (including *feature interaction bugs*) occurring in a large highly-configurable system, the Linux kernel. We address this objective via a qualitative in-depth analysis and documentation of 42 cases of such bugs. We make the following contributions:

- *Identification of 42 variability bugs in the Linux kernel*, including in-depth analysis and presentation for non-experts.

- *A database containing the results of our analysis*, encompassing a detailed data record about each bug.

These bugs comprise common types of errors in C software, and cover different types of feature interactions. We intend to grow the collection in the future with the help of the research community. The current version is available at http://VBDb.itu.dk.

- *Self-contained simplified C99 versions of all bugs.* These ease comprehension of the underlying causes, and can be used for testing bug-finders in a smaller scale.
- *An aggregated reflection over the collection of bugs.* Providing insight on the nature of bugs induced by feature interactions in a large project like Linux.

We adopt a qualitative manual methodology of analysis for three reasons. First, family-based automated analysis tools that scale for the Linux kernel do not exist. In fact, without this study it was unclear what tools should be built. Second, using conventional (not family-based) analysis tools on individual variants after preprocessing does not scale (if applied exhaustively) or yields low probability of finding bugs (if applied by random sampling). Third, searching for bugs with tools only finds cases that these tools cover, while we were interested in exploring the nature of variability bugs widely.

Reflecting on the collected material, we learn that variability bugs are very complex, they involve many aspects of programming language semantics, they are distributed in most parts of Linux project, involve multiple features and span code in remote locations. Detecting these bugs is difficult for both people and tools. Once feature-sensitive analyses that are able to capture these bugs are available, it will be interesting to conduct extensive quantitative experiments to confirm our qualitative intuitions.

We direct our work to designers of program analysis and bug finding tools. We believe that the collection of bugs can inspire them in several ways: (i) it will provide a set of concrete, well described challenges for analyses, (ii) it will serve as a benchmark for evaluating their tools, and (iii) it will dramatically speed up design of new techniques, since they can be tried on simplified Linux-independent bugs. Using realistic bugs from a large piece of software in evaluation can aid tuning the analysis precision, and incite designers to support certain language constructs in the analysis.

We present basic background in Sect. 2. The methodology is detailed in Sect. 3. Sections 4–5 describe the analysis: first the considered dimensions, then the aggregate observations. We finish surveying threats to validity (Sect. 6), related work (Sect. 7) and a conclusion (Sect. 8).

2. BACKGROUND

We use the term *software bug* to refer to both faults and errors as defined by IEEE Standard Glossary of Software Engineering [32]. A *fault* (*defect*) is an incorrect instruction in the software, introduced into the code as a result of a human mistake. Faults induce *errors*, that are incorrect program states, such as a pointer being null when it should not be. In this work we collected errors that manifested as runtime failures (typically a kernel panic), as well as defects spotted when building a specific kernel configuration. While the latter might look harmless (for instance an unused variable) we assume that they might be side-effects of serious misconceptions potentially leading to other bugs.

A *feature* is a unit of functionality additional to the core software [11]. The core (*base variant*) implements the basic functionality present in any variant of a program family. The different selections of features (*configurations*) define the set of program variants. Often, two features cannot be simultaneously enabled, or one feature requires enabling another. Feature dependencies are specified using a *feature model* [20] (or a decision model [18]), denoted here by ψ_{FM}; effectively a constraint over features defining legal configurations.

```
1  #include <stdlib.h>
3  void foo(int a) {                                  →(5)
4      printf("%d\n",2/a);      // ERROR               (6) ×
5  }
7  int main(void) {             // START               ⇒(1)
8      int x = 1;                                       (2)
9      #ifdef CONFIG_INCR       // DISABLED            |
10     x = x + 1;                                       |
11     #endif                                           |
12     #ifdef CONFIG_DECR       // ENABLED             ↓
13     x = x - 1;                                       (3)
14     #endif                                           ↓
15     foo(x);                                          (4)→
16 }
```

Figure 1: Example of a program family and a bug.

Preprocessor-based program families [21] associate features with macro symbols, and define their implementations as statically conditional code guarded by constraints over feature symbols. The macro symbols associated to features (*configuration options*) are often subject to naming conventions, for instance, in Linux these identifiers shall be prefixed by `CONFIG_`. We follow the Linux convention through out this paper. Figure 1 presents a tiny preprocessor-based C program family using two features, *INCR* and *DECR*. Statements at lines 10 and 13 are conditionally present. Assuming an unrestricted feature model ($\psi_{\text{FM}} = \texttt{true}$), the figure defines a family of four different variants.

A *presence condition* φ of a code fragment is a *minimal* (by the number of referred variables) Boolean formula over features, specifying the subset of configurations in which the code is included in the compilation. The concept of presence condition extends naturally to other entities; for instance, a presence condition for a bug specifies the subset of configurations in which a bug occurs. Concrete configurations, denoted by κ, can also be written as Boolean constraints—conjunctions of feature literals. A code fragment with presence condition φ is thus present in a configuration κ iff $\kappa \models \varphi$. As an example, consider the decrement statement in line 13, which has presence condition *DECR*, thus it is part of configurations $\kappa_0 = \neg \mathit{INCR} \wedge \mathit{DECR}$ and $\kappa_1 = \mathit{INCR} \wedge \mathit{DECR}$.

Features can influence the functions offered by other features—a phenomenon known as *feature interaction*, which can be either intentional or unexpected. In our example, the two features interact because both modify and use the same program variable `x`. Enabling either *INCR* or *DECR*, or both, results in different values of `x` prior to calling `foo`.

As a result of variability, bugs can occur in some configurations but not in others, and can also manifest differently in different variants. If a bug occurs in one or more configurations, and does not occur in at least one other configuration, we call it a *variability bug*. Figure 1 shows how one of the program variants in our example family, namely κ_0, will crash at line 4 when we attempt to divide by zero. Because this bug is not manifested in any other variant, it is a variability bug—with presence condition $\neg \mathit{INCR} \wedge \mathit{DECR}$.

Program family implementations are usually conceptually stratified in three layers: the *problem space* (typically a feature model), a *solution space* implementation (e.g. C code), and the *mapping* between the problem and solution spaces (the build system and CPP in Linux). We show how the division-by-zero bug could be fixed, depending on the interpretation, in our running example, in each layer separately. We show changes to code in unified diff format (`diff -U0`).

Fix in code. If function `foo` should accept any `int` value, then the bug is fixed by appropriately handling zero as input.

```
@@ -4 +4,4 @@
-  printf("%d\n",2/a);
+  if (a != 0)
+    printf("%d\n",2/a);
+  else
+    printf("NaN\n");
```

Fix in mapping. If we assume that function `foo` shall not be called with a zero argument, a possible fix is to decrement `x` only when both *DECR* and *INCR* are enabled.

```
@@ -12 +12 @@
-  #ifdef CONFIG_DECR
+  #if defined(CONFIG_DECR) && defined(CONFIG_INCR)
```

Fix in model. If the bug is caused by an illegal interaction, we can introduce a dependency in the feature model to prevent the faulty configuration κ_0. For instance, let *DECR* be only available when *INCR* is enabled. Assuming feature model $\psi_{\text{FM}} = \textit{DECR} \rightarrow \textit{INCR}$ forbids κ_0.

3. METHODOLOGY

Objective. Our objective is to qualitatively understand the complexity and nature of *variability bugs* (including *feature-interaction bugs*) occurring in a large highly-configurable system: the Linux kernel. This includes addressing the following research questions:

- RQ1: Are variability bugs limited to any particular type of bugs, "error-prone" features, or specific location?
- RQ2: In what ways does variability affect software bugs?

Subject. We study the Linux kernel, taking the Linux stable GIT[1] repository as the unit of analysis. Linux is likely the largest highly-configurable open-source system. It has about ten million lines of code and more than ten thousand features. Crucially, data about Linux bugs is available freely. We have free access to the bug tracker[2], the source code and change history[3], and to public discussions on the mailing list[4] (LKML) and other forums. There also exist books on Linux development [8, 25]—valuable resources when understanding a bug-fix. Access to domain specific knowledge is crucial for the qualitative analysis.

We focus on bugs already corrected in commits to the Linux repository. These bugs have been publicly discussed (usually on LKML) and confirmed as actual bugs by kernel developers, so the information about the nature of the bug fix is reliable, and we minimize the chance of including fictitious problems.

Methodology. Our methodology has three parts: first, we identify the variability bugs in the kernel history. Second,

[1] http://git-scm.com/
[2] https://bugzilla.kernel.org/
[3] http://git.kernel.org/cgit/linux/kernel/git/
[4] https://lkml.org/

```
CONFIG_fid
configuration
config option
if fid is [not]? set
when fid is [not]? set
if fid is [en|dis]abled
when fid is [en|dis]abled
```

(a) Message filters.

```
#if
#else
#elif
#endif
select fid
config fid
depends on fid
```

(b) Content filters.

Figure 2: Regular expressions selecting configuration-related commits in: (a) message, (b) content; *fid* abbreviates `[A-Z0-9_]+`, matching feature identifiers.

```
bug
fix
oops
warn
error
unsafe
invalid
violation
end trace
kernel panic
```

(a) Generic bug filters.

```
void *
unused
overflow
undefined
double lock
memory leak
uninitialized
dangling pointer
null [pointer]? dereference
...
```

(b) Specific bug filters.

Figure 3: Regular expressions selecting bug-fixing commits: (a) generic, (b) problem specific

we analyze and explain them. Finally, we reflect on the aggregated material to answer our research questions.

Part 1: Finding Variability Bugs. We have settled on a semi-automated search through Linux commits to find variability bugs via historic bug fixes. As of April 2014 the Linux repository has over 400,000 commits, which rules out manual investigation of each commit. We have thus *searched* through the commits for variability bugs using the following steps:

1. *Selecting variability-related commits.* We retain commits matching regular expressions of Fig. 2. Expressions in Fig. 2(a) identify commits in which the author's *message* relates the commit to specific features. Those in Fig. 2(b) identify commits introducing changes to the feature mapping or the feature model. We reject *merges* as such commits do not carry changes. This step selects in the order of tens of thousands of commits.
2. *Selecting bug-fixing commits.* We narrow to commits that fix bugs, matching regular expressions that indicate bugs within the commit message (see Fig. 3). Depending on the keywords of interest this step may select from thousands of commits, to only a few tens or less.
3. *Manual scrutiny.* We read the commit message and inspect the changes introduced by the commit to remove obvious false positives. We order commits by the number of hits in the first two searches, and down prioritize very complex commits (given the information provided in the commit message and the number of lines modified by the patch).

Part 2: Analysis. The second part of the methodology is significantly more laborious than the first part. For each variability bug identified, we manually analyze the commit message, the patch fix, and the actual code to build an understanding of the bug. When more context is required,

we find and follow the associated LKML discussion. Code inspection is supported by CTAGS[5] and the Unix GREP utility, since we lack feature-sensitive tool support.

1. *The semantics of the bug.* For each variability bug we want to understand the *cause* of the bug, the *effect* on the program semantics and the relation between the two. This often requires understanding the inner workings of the kernel, and translating this understanding to general programming language terms accessible to a broader audience. As part of this process we try to identify a relevant runtime execution *trace* and collect links to available information about the bug online.

2. *Variability related properties.* We establish what is the presence condition of a bug (precondition in terms of configuration choices) and where it was fixed (in the code, in the feature model or in the mapping).

3. *Simplified version.* Last but not least, we condense our understanding in a *simplified version of the bug.* This serves to explain the original bug, and constitutes an interesting benchmark for evaluating tools.

We analyzed Linux bugs from the previous step following this method and stored the created reports in a publicly available database. We were looking for a sufficiently diverse sample, and stopped at 42 bugs once it became possible to answer our two research questions. The detailed content of the report is explained in Sect. 4.

Part 3: Data Analysis and Verification. We reflect on the collected data set in order to find answers to our research questions. This step is supported with some quantitative data but, importantly, we do not make any quantitative conclusions about the population of the variability bugs in Linux (such conclusions would be unsound given the above research method). It purely characterizes diversity of the data set obtained. This allows to present the entire collection of bugs in an aggregated fashion (see Sect. 5).

Finally, in order to reduce bias we confront our method, findings, and hypotheses in an interview with a full-time professional Linux kernel developer.

4. DIMENSIONS OF ANALYSIS

We begin by selecting a number of properties of variability bugs to understand, analyze and document in bug reports. These are described below and exemplified by data from our database. We show an example record in Fig. 4, a null-pointer dereference bug found in a driver, which was traced back to errors both in the feature model and the mapping.

Type of Bug (`type`). In order to understand the diversity of variability bugs we establish the type of bugs according to the *Common Weakness Enumeration* (CWE)[6]—a catalog of numbered software weaknesses and vulnerabilities. We follow CWE since, it was applied to the Linux kernel before [31]. However, since CWE is mainly concerned with security, we had to extend it with a few additional types of bugs, including type errors, incorrect uses of Linux APIs, etc. The types of bugs in the obtained taxonomy are listed in Tbl. 1; our additions lack an identifier in the rightmost column. The bug types directly indicate what kind of analysis and program verification techniques can be used to address the bugs identified in the kernel. For instance the category of memory errors (Tbl. 1) maps almost directly to various program analyses: for null pointers [10, 16, 19], buffer overruns [7, 15, 35], memory leaks [10, 16], etc.

Bug Description (`descr`). Understanding a bug requires rephrasing its nature in general software engineering terms, so that the bug becomes understandable for non kernel-experts. We obtain such a description by studying the bug in depth, and following additional available resources (such as mailing list discussions, available books, commit messages, documentation and online articles). Whenever use of the Linux terminology is unavoidable, we provide links to the necessary background. Obtaining the description is often non-trivial. For example, one bug in our database (commit `eb91f1d0a53`) was fixed with the following commit message:

```
Fixes the following warning during bootup when compiling with CONFIG_SLAB:

[ 0.000000] ------------[ cut here ]------------
[ 0.000000] WARNING: at kernel/lockdep.c:2282 lockdep_trace_alloc+0x91/0xb9()
[ 0.000000] Hardware name: [ 0.000000] Modules linked in:
[ 0.000000] Pid: 0, comm: swapper Not tainted 2.6.30 #491
[ 0.000000] Call Trace:
[ 0.000000] [<ffffffff81087d84>] ? lockdep_trace_alloc+0x91/0xb9
...
```

It is summarized in our database as:

> **Warning due to a call to `kmalloc()` with flags `__GFP_WAIT` and interrupts enabled**
>
> The *SLAB* allocator is initialized by `start_kernel()` with interrupts disabled. Later in this process, `setup_cpu_cache()` performs the per-CPU *kmalloc cache* initialization, and will try to allocate memory for these caches passing the `GFP_KERNEL` flags. These flags include `__GFP_WAIT`, which allows the process to sleep while waiting for memory to be available. Since, as we said, interrupts are disabled during *SLAB* initialization, this may lead to a deadlock. Enabling *`LOCKDEP`* and other debugging options will detect and report this situation.

We add a one-line header to the description, here shown in bold, to help identification and listing of bugs.

Program Configurations (`config`). In order to confirm that a bug is indeed a variability bug we investigate under what presence condition it appears. This allows to rule out bugs that appear unconditionally and enables further investigation of variability properties of the bug, for example the number of features and nature of dependencies that enable the bug.

Our example bug (Fig. 1) is present when *`DECR`* is enabled but *`INCR`* is disabled. The Linux bug captured in Fig. 4(b) requires enabling *`TWL4030_CORE`*, and disabling *`OF_IRQ`*, in order to exhibit the erroneous behavior (see *`config`* entry in the left part).

Bug-Fix Layer (`layer`). We analyze the fixing commit to establish whether the source of the bug is in the code, in the feature model, or in the mapping. Understanding this can help direct future research on building diagnostics tools: are tools needed for analyzing models, mappings, or code? Where is it best to report an error?

The bug of Fig. 4 has been fixed both in the model and in the mapping (cf. Fig. 5). The fixing commit asserts that: first, *`TWL4030_CORE`* should not depend on *`IRQ_DOMAIN`* (fixed in the model), and, second, that the assignment of the variable `ops` to `&irq_domain_simple_ops` is part of the *`IRQ_DOMAIN`* code and not of *`OF_IRQ`* (fixed in the mapping).

Error Trace (`trace`). We manually analyze the execution trace that leads to the error state. Slicing tools cannot easily

[5] http://ctags.sourceforge.net/
[6] http://cwe.mitre.org/

type: Null pointer dereference

descr: Null pointer on !*OF_IRQ* gets dereferenced if *IRQ_DOMAIN*.

In *TWL4030* driver, attempt to register an IRQ domain with a NULL ops structure: ops is de-referenced when registering an IRQ domain, but this field is only set to a non-null value when *OF_IRQ*.

config: *TWL4030_CORE && !OF_IRQ*

bugfix:

repo: git://git.kernel.org/pub/.../linux-stable.git

hash: 6252547b8a7acced581b649af4ebf6d65f63a34b

layer: model, mapping

trace:

```
.  dyn-call drivers/mfd/twl-core.c:1190:twl_probe()
.  1235:  irq_domain_add(&domain);
.. call kernel/irq/irqdomain.c:20:irq_domain_add()
... call include/linux/irqdomain.h:74:irq_domain_to_irq()
... ERROR 77:  if (d->ops->to_irq)
```

links:

* [I2C](http://cateee.net/lkddb/web-lkddb/I2C.html)
* [TWL4030](http://www.ti.com/general/docs/...)
* [IRQ domain](http://lxr.gwbnsh.net.cn/.../IRQ-domain.txt)

(a) Bug record.

```
2   #include <stdlib.h>
4   #ifdef CONFIG_TWL4030_CORE              // ENABLED
5   #define CONFIG_IRQ_DOMAIN
6   #endif
8   #ifdef CONFIG_IRQ_DOMAIN                // ENABLED
9   int irq_domain_simple_ops = 1;
11  void irq_domain_add(int *ops) {                    →(6)
•12     int irq = *ops;                     // ERROR    (7)×
13  }
14  #endif
16  #ifdef CONFIG_TWL4030_CORE              // ENABLED
17  void twl_probe() {                                 →(3)
18      int *ops = NULL;                                (4)
19      #ifdef CONFIG_OF_IRQ                // DISABLED  |
20      ops = &irq_domain_simple_ops;                    |
21      #endif                                           |
22      irq_domain_add(ops);                            (5)→
23  }
24  #endif
•26 int main(void) {                                    ⇒(1)
27      #ifdef CONFIG_TWL4030_CORE          // ENABLED   ↓
28      twl_probe();                                    (2)→
29      #endif
30  }
```

(b) Simplified version.

Figure 4: Bug 6252547b8a7: a record example and a simplified version.

```
@@ -2,8 +2,4 @@
 #include <stdlib.h>
-#ifdef CONFIG_TWL4030_CORE
-#define CONFIG_IRQ_DOMAIN
-#endif
-
 #ifdef CONFIG_IRQ_DOMAIN
 int irq_domain_simple_ops = 1;
@@ -16,9 +12,9 @@
 #ifdef CONFIG_TWL4030_CORE
 void twl_probe() {
+  #ifdef CONFIG_IRQ_DOMAIN
   int *ops = NULL;
-  #ifdef CONFIG_OF_IRQ
   ops = &irq_domain_simple_ops;
+  irq_domain_add(ops);
   #endif
-  irq_domain_add(ops);
 }
 #endif
```

Figure 5: Fix for simplified bug 6252547b8a7. The patch is given in unified diff format (`diff -U2`).

be used for these purpose, as none of them is able to handle static preprocessor directives appropriately. Constructing a trace allows us to understand the nature and complexity of the bug. A documented failing trace allows other researchers to understand a bug much faster.

There are two types of entries in our traces: function calls and statements. Function call entries can be either static (tagged `call`), or dynamic (`dyn-call`) if the function is called via a function pointer. A statement entry highlights relevant changes in the program state. Every entry starts with a non-empty sequence of dots indicating the nesting of function calls, followed by the location of the function definition (file and line) or statement (only the line). The statement in which the error is manifested is marked with an `ERROR` label.

In Fig. 4(a) the trace starts in the driver loading function (`twl_probe`). This is called from `i2c_device_probe` at `drivers/i2c/i2c-core.c`, the generic loading function for *I2C*[7] drivers, through a function pointer (`driver->probe`). A call to `irq_domain_add` passes the globally-declared struct `domain` by reference, and the `ops` field of this struct, now aliased as `*d`, is dereferenced (`d->ops->to_irq`).

The `ops` field of `domain` is not explicitly initialized, so it has been set to null by default (as dictated by the C standard). Thus the above error trace unambiguously identifies a path from the loading of the driver to a null-pointer dereference, when *OF_IRQ* is disabled. Had *OF_IRQ* been enabled, the `ops` field would have been properly initialized prior to the call to `irq_domain_add`.

Simplified Bug. Last but not least, we synthesize a simplified version of the bug capturing its most essential properties. We write a small C99 program, independent of the kernel code, that exhibits the same essential behavior (and the same problem). The obtained simplified bugs are easily accessible for researchers willing to try program verification and analysis tools without integrating with the Linux build infrastructure, huge header files and dependent libraries, and, most importantly, without understanding the inner workings of the kernel. Furthermore, the entire set of simplified bugs constitute an easily accessible benchmark suite derived from real bugs occurring in a large-scale software system, which can be used to evaluate bug finding tools in a smaller scale.

Simplified bugs are derived systematically from the error trace. Along this trace, we preserve relevant statements and control-flow constructs, mapping information and function calls. We keep the original identifiers for features, functions and variables. However, we abstract away dynamic dispatching via function pointers, struct types, *void* pointers, casts, and any Linux-specific type, when this is not relevant for the bug. When there exist dependencies between features, we force valid configurations with `#define`. This encoding of feature dependencies has the advantage of making the simplified bug files self-contained.

[7] A serial bus protocol used in micro controller applications.

Figure 4(b) shows the simplified version of our running example bug with null pointer dereference. Lines 4–6 encode a dependency of *TWL4030_CORE* on *IRQ_DOMAIN*, in order to prevent the invalid configuration *TWL4030_CORE* $\wedge \neg$ *IRQ_DOMAIN*. We encourage the reader to study the execution trace leading to a crash by starting from `main` at line 26. This takes a mere few minutes, as opposed to many hours necessary to obtain an understanding of a Linux kernel bug normally. Note that the trace is to be interpreted under the presence condition from the bug record (decisions are specified in comments next to the `#if` conditionals).

Traceability Information. We store the URL of the repository, in which the bug fix is applied, the commit *hash*, and links to relevant context information about the bug, in order to support independent verification of our analysis.

5. DATA ANALYSIS

In order to address the research questions, we have reflected on the entire body of information gathered, arriving at detailed observations presented below. In the following, we sometimes aggregate data with numbers. The numbers are used solely for descriptive purposes—no statistical conclusions should be drawn from them (we emphasize this using a gray font).

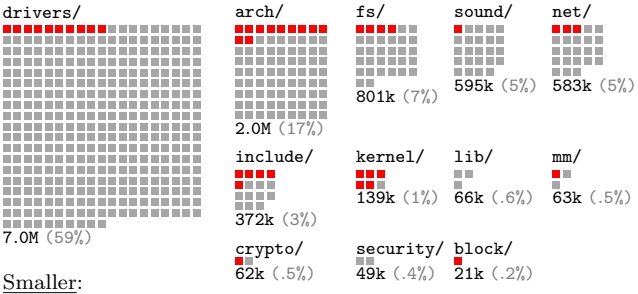
We start by presenting the observations that support our first research question, RQ1:

> OBSERVATION 1: *Variability bugs are not limited to any particular type of bugs.*

Table 1 lists the type of bugs we found, along with occurrence frequencies in the collection. For example, 15 bugs have been classified under the category of *memory errors*, four of which are null pointer dereferences. We note that variability bugs cover a wide range of qualitatively different types of bugs from

Table 1: Types of bugs among the 42 bugs. The first column gives the frequency of these bugs in our collection.

15	**memory errors:**	**CWE ID**
4	null pointer dereference	476
3	buffer overflow	120
3	read out of bounds	125
2	insufficient memory	-
1	memory leak	401
1	use after free	416
1	write on read only	-
10	**compiler warnings:**	**CWE ID**
5	uninitialized variable	457
2	incompatible types	843
1	unused function (dead code)	561
1	unused variable	563
1	void pointer dereference	-
7	**type errors:**	**CWE ID**
5	undefined symbol	-
1	undeclared identifier	-
1	wrong number of args to function	-
7	**assertion violations:**	**CWE ID**
5	fatal assertion violation	617
2	non-fatal assertion violation	617
2	**API violations:**	**CWE ID**
1	Linux *sysfs* API violation	-
1	double lock	764
1	**arithmetic errors:**	**CWE ID**
1	numeric truncation	197



Smaller:
- `virt/` (6.8k), `ipc/` (6.4k), `init/` (2.0k), and `usr/` (0.6k).

Infrastructure:
- `tools/` (102k), `scripts/` (44k), and `samples/` (2.1k).

Figure 6: Location of the 42 bugs in the main Linux directories as of March 2014. Each square represents 25 thousand lines of code. The precise number of LOC and its percentage of the total is given below the squares. A red (dark) square symbolizes the occurrence of one of the bugs.

type errors, through data-flow errors such as uninitialized variables, to locking policy violations (double locks).

We found 17 bugs, type errors and compiler warnings, caught by the compiler at build time. Despite the compiler checks, the bugs had been admitted to the repository in the first place. Since compiler errors cannot easily be ignored, we take this as evidence that the author of the commit (and the maintainer who accepted it) could not find the bug, because they compiled the code in configurations that do not exhibit it (compiler checks are not family-based).

> OBSERVATION 2: *Variability bugs appear to not be restricted to specific "error prone" features.*

Table 2 shows the complete list of features involved in the bugs: a total of 78 qualitatively different features, ranging from *debugging* options (e.g., *QUOTA_DEBUG* and *LOCKDEP*), to *device drivers* (e.g., *TWL4030_CORE* and *ANDROID*), to *network protocols* (e.g., *VLAN_8021Q* and *IPV6*), to *computer architectures* (e.g., *PARISC* and *64BIT*). Three features are involved in three of the bugs, nine features occur in two bugs, and the remaining 66 are involved in only a single bug.

Table 2: Features involved in the bugs.

64BIT	IP_SCTP	S390
ACPI_VIDEO	JFFS2_FS_WBUF_VERIFY	S390_PRNG
ACPI_WMI	KGDB	SCTP_DBG_MSG
AMIGA_Z2RAM	KPROBES	SECURITY
ANDROID	KTIME_SCALAR	SHMEM
ARCH_OMAP2420	LBDAF	SLAB
ARCH_OPAM3	LOCKDEP	SLOB
ARM_LPAE	MACH_OMAP_H4	SMP
BACKLIGHT_CLASS_DEVICE	MODULE_UNLOAD	SND_FSI_AK4642
BCM47XX	NETPOLL	SND_FSI_DA7210
BDI_SWITCH	NUMA	SSB_DRIVER_EXTIF
BF60x	OF	STUB_POULSBO
BLK_CGROUP	OF_IRQ	SYSFS
CRYPTO_BLKCIPHER	PARISC	TCP_MD5SIG
CRYPTO_TEST	PCI	TMPFS
DEVPTS_MULTIPLE_INSTANCES	PM	TRACE_IRQFLAGS
DISCONTIGMEM	PPC64	TRACING
DRM_I915	PPC_256K_PAGES	TREE_RCU
EP93XX_ETH	PREEMPT	TWL4030_CORE
EXTCON	PROC_PAGE_MONITOR	UNIX98_PTYS
FORCE_MAX_ZONEORDER=11	PROVE_LOCKING	VLAN_8021Q
HIGHMEM	QUOTA_DEBUG	VORTEX
HOTPLUG	RCU_CPU_STALL_INFO	X86
I2C	RCU_FAST_NO_HZ	X86_32
IOSCHED_CFQ	REGULATOR_MAX8660	XMON
IPV6	REISERFS_FS_SECURITY	ZONE_DMA

Observation 3: *Variability bugs are not confined to any specific location (file or kernel subsystem).*

Figure 6 shows in which subsystems the bugs are located and the relative size of each subsystem as of March 2014 —we approximate subsystems by directories. The size of each subsystem is measured in lines of code (LOC), we take the sum of LOC (for any language) as reported by CLOC[8] (version 1.53). E.g., with six squares, the `kernel/` subsystem has approximately 150 KLOC and represents about 1% of the Linux code. Superimposed onto the size visualization, the figure also shows in which directories the bugs occur. With five red (dark) squares, the directory `kernel/` thus houses five of the bugs of our collection.

We found bugs in ten of the main Linux subsystems, showing that variability bugs are not confined to any specific subsystem. These are qualitatively different subsystems of Linux ranging from *networking* (`net/`), to *device drivers* (`drivers/`, `block/`), to *filesystems* (`fs/`) or *encryption* (`crypto/`). Note that Linux subsystems are often maintained and developed by different people, which adds to diversity of our collection.

We found no bug in nine directories, representing less than the 3% of the Linux kernel code in total. Further, three of them (`tools/`, `scripts/`, and `samples/`) contain example and support code (build infrastructure, diagnostic tools, etc.) that does not run on a compiled kernel.

We are now ready to answer RQ1:

Conclusion 1: *Variability bugs are indeed not confined to any particular type of bug, error-prone feature, or location in the Linux kernel.*

We have found variability bugs falling in 20 different types of semantic errors, involving 78 qualitatively different features, and located in 10 major subsystems of the Linux kernel.

We now turn to evidence regarding research question RQ2:

Observation 4: *We have identified 30 bugs that involve non-locally defined features; i.e., features that are "remotely" defined in another subsystem than where the bug occurred.*

Understanding such bugs involves functionality and features from different subsystems, while most Linux developers are dedicated to a single subsystem. For example, bug `6252547b8a7` (Fig. 4) occurs in the `drivers/` subsystem, but one of the interacting features, *`IRQ_DOMAIN`*, is defined in `kernel/`. Bug `0dc77b6dabe`, which occurs in the loading function of the *extcon-class* module (`drivers/`), is caused by an improper use of the *sysfs* virtual filesystem API—feature *`SYSFS`* in `fs/`. We confirmed with a Linux developer that cross-cutting features constitute a frequent source of bugs.

Observation 5: *Variability can be implicit and even hidden in (alternative) configuration-dependent macro, function, or type definitions specified in (potentially different) header files.*

Hidden variability significantly complicates the identification of variability-related problems. For example, in bug `0988c4c7fb5`, function `vlan_hwaccel_do_receive` is called if a VLAN-tagged network packed is received. This function, however, has two different definitions depending on whether feature *`VLAN_8021Q`* is present or not. Variants without *`VLAN_8021Q`* support are compiled with a mockup-implementation of this function that unconditionally enters

[8] http://cloc.sourceforge.net/

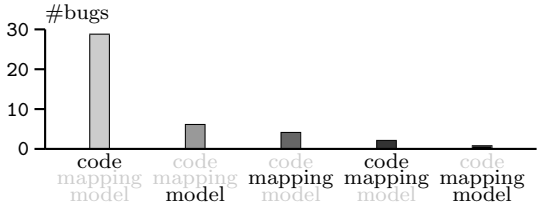


Figure 7: In which layer(s) are the bugs fixed.

an error state. Another example is bug `0f8f8094d28` which can be regarded as a trivial out of bounds access to an array, except that the length of the array (`KMALLOC_SHIFT_HIGH+1`) is architecture-dependent, and only the PowerPC architectures, for a given virtual page size, are affected. Both `vlan_hwaccel_do_receive` and `KMALLOC_SHIFT_HIGH` have alternative definitions at different locations.

Observation 6: *Variability bugs are fixed not only in the code; some are fixed in the mapping, some are fixed in the model, and some are fixed in a combination of these.*

Figure 7 shows whether the bugs in our sample were fixed in the *code*, *mapping*, or *model*. Even though we only documented bugs that manifested in code, 13 bugs in our sample were fixed in the mapping, in the model, or in two layers.

Examples of simple fixes in the mapping and in the model are commits `472a474c663` and `7c6048b7c83`, respectively. The former adds a new `#ifndef` to prevent a double call to `APIC_init_uniprocessor`—which is not idempotent, while the latter modifies *`STUB_POULSBO`*'s KCONFIG entry to prevent a build error.

Bug-fix `6252547b8a7` (Fig. 5) removes a feature dependency (*`TWL4030_CORE`* no longer depends on *`IRQ_DOMAIN`*) and changes the mapping to initialize the struct field `ops` when *`IRQ_DOMAIN`* (rather than *`OF_IRQ`*) is enabled. An example of multiple fix in mapping and code is commit `63878acfafb`, which removes the mapping of some initialization code to feature *`PM`* (power management), and adds a function stub.

This stratification into code, mapping and model may obscure the cause of bugs, because an adequate analysis of a bug requires understanding these three layers. Further, each layer involves different languages; in particular, for Linux: the code is C, the mapping is expressed using both CPP and GNU MAKE, and the feature model is specified using KCONFIG.

Presumably, this complexity may cause a developer to fix a bug in the wrong place. For instance, the dependency of *`TWL4030_CORE`* on *`IRQ_DOMAIN`* removed by our bug-fix `6252547b8a7` was added by commit `aeb5032b3f8`. Apparently `aeb5032b3f8` introduced this dependency into the feature model to prevent a build error, so to fix a bug, but this had undesirable side-effects. According to the message provided in commit `6252547b8a7`, the correct fix to the build error was to make a variable declaration conditional on the presence of feature *`IRQ_DOMAIN`*.

Observation 7: *We have identified as many as 30 feature-interaction bugs in the Linux kernel.*

We define the *feature-interaction degree of a bug*, or just *degree of a bug*, as the number of individual features occurring in its presence condition. Intuitively, the degree of a bug

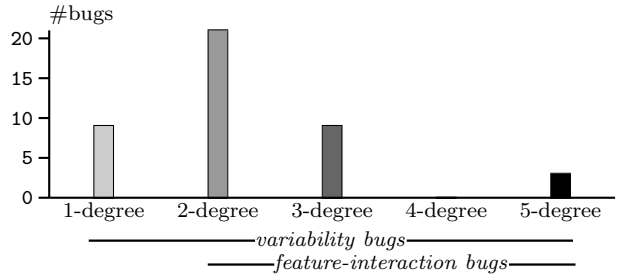


Figure 8: Numbers of features involved in a bug (*feature-interaction degree*).

indicates the number of features that have to interact so that the bug occurs. A bug present in any valid configuration is a bug independent of features, or a 0-degree bug. Bugs with a degree greater than zero are variability bugs, thus occurring in a nonempty strict subset of valid configurations. Particularly, if the degree of a bug is greater than one, the bug is caused by the interaction of two or more features. A software bug that arises as a result of feature interactions is referred to as a *feature-interaction bug*.

Feature-interaction bugs are inherently more complex because the number of variants to be considered is exponential in the degree of the bug. Bug `6252547b8a7` (cf. Fig. 4(b)) is the result of a two-feature interaction. The code slice containing the bug involves three different features, and represents four variants (corrected for the feature model), but only one of the variants presents a bug. The `ops` pointer is dereferenced in variants with *TWL4030_CORE* enabled, but it is not properly initialized unless *OF_IRQ* is enabled. A developer searching for this bug needs to either think of each variant individually, or consider the combined effect of each feature on the value of the `ops` pointer. None of these are easy to execute systematically even in a simplified scenario, and outright infeasible in practice, as confirmed by a professional Linux developer.

Feature interactions can be extremely subtle when variability affects type definitions. Commit `51fd36f3fad` fixes a bug in the Linux high-resolution timers mechanism due to a numeric truncation error, that only happens in 32-bit architectures not supporting the *KTIME_SCALAR* feature. In these particular configurations `ktime_t` is a struct with two 32-bit fields, instead of a single 64-bit field, used to store the remaining number of nanoseconds to execute the timer. The bug occurs on attempt to store some large 64-bit value in one of these 32-bit fields, causing a negative value to be stored instead. Interestingly, one of the Linux developers we interviewed also mentioned the difficulty to optimize for cache-misses due to variability in the alignment of struct fields.

> Observation 8: *We have identified 12 bugs involving three or more features.*

An example of a 3-degree bug is `ae249b5fa27`, caused by the interaction of *DISCONTIGMEM* (efficient handling of discontiguous physical memory) support in PA-RISC architectures (feature *PARISC*), and the ability to monitor memory utilization through the `proc/` virtual filesystem (feature *PROC_PAGE_MONITOR*). We also found 5-degree bugs such as commit `221ac329e93`, caused by 32-bit PowerPC architectures not disabling kernel memory write-protection when *KPROBES* is enabled—a dynamic debugging feature that requires modifying the kernel code at runtime.

Figure 8 summarizes the degree of our bugs. To the best of our knowledge, this is the first documented collection of feature-interaction bugs in the operating systems domain. So far, most feature-interaction bugs have been identified, documented, and published in telecommunication domain [11].

> Observation 9: *Presence conditions for variability bugs also involve disabled features.*

Table 3 lists and groups the structure of the presence conditions for our sample. We observe two main classes of bug presence conditions: *some-enabled*, where one or more features have to be enabled for the bug to occur; and *some-enabled-one-disabled*, where the bug is present when enabling zero or more features and disabling *exactly one* feature. We identified 20 bugs in *some-enabled* configurations, and another 20 bugs in *some-enabled-one-disabled*. (Note that one of the presence conditions has the form, $(a \vee a') \wedge \neg b$, but, since it is implied by either $a \wedge \neg b$ or $a' \wedge \neg b$, we include it in the *some-enabled-one-disabled* class.)

Testing of highly configurable systems is often approached by testing one or more maximal configurations, in which as many features as possible are enabled—in Linux this is done using the predefined configuration *allyesconfig*. This strategy allows to find many bugs with *some-enabled* presence conditions simply by testing one single maximal configuration. But, if negated features occur in practice as often as in our sample, then testing maximal configurations only, will miss a significant amount of bugs.

In our experience, the implementation of features in Linux is crosscutting many code locations, and features code is intermixed. As a result, disabling a feature can both add or delete code from another feature, and we expect negated features to be often part of bugs presence conditions. Bug `6252547b8a7` (Fig. 4) is such an example. Disabling *OF_IRQ* causes the null pointer dereference because this feature is responsible for initializing the `ops` struct field. Another example is bug `60e233a5660`, where the implementation of a function `add_uevent_var`, when feature *HOTPLUG* is disabled, fails to preserve an invariant causing a buffer overflow.

> Observation 10: *Effective testing strategies exist for the observed bug presence conditions.*

Given the observed patterns (*some-enabled* and *some-enabled-one-disabled*) in Tbl. 3, we can think of a better testing

Table 3: The structure of the presence conditions (i.e., in which configurations the 42 bugs occur).

20	***some-enabled:***	
6	a	
8	$a \wedge b$	
5	$a \wedge b \wedge c$	
0	$a \wedge b \wedge c \wedge d$	
1	$a \wedge b \wedge c \wedge d \wedge e$	
20	***some-enabled-one-disabled:***	
3	$\neg a$	
13	$a \wedge \neg b$	*one of which is:* $(a \vee a') \wedge \neg b$
3	$a \wedge b \wedge \neg c$	
0	$a \wedge b \wedge c \wedge \neg d$	
1	$a \wedge b \wedge c \wedge d \wedge \neg e$	
2	**other configurations:**	
1	$\neg a \wedge \neg b$	
1	$a \wedge \neg b \wedge \neg c \wedge \neg d \wedge \neg e$	

strategy than maximal configuration testing. We propose a *one-disabled configuration testing* strategy, where we test configurations in which exactly one feature is disabled, corresponding to the formulas $\forall g{\in}\mathbb{F}$: $(\bigwedge_{f\in\mathbb{F}\setminus\{g\}} f) \wedge \neg g$. Table 4 compares the two strategies, maximal configuration testing and *one-disabled* configuration testing. We also add an entry for exhaustive testing of all configurations, serving as a baseline (the cost is exponential there).

Maximal configuration testing has constant cost—ideally only one configuration has to be tested, and thus scales to program families with an arbitrarily large number of features ($\mathbb{F}$). It appears to be a fairly good heuristic: 48% of bugs in our sample, 20 out of 42, could be found this way. One-disabled configuration testing has a linear cost on $|\mathbb{F}|$, thus it is reasonably scalable, even for program families with thousands of features like Linux. Remarkably, 95% of our bugs, 40 out of 42, could be found by testing the $|\mathbb{F}|$ one-disabled configurations. Note that these configurations also find the bugs with a *some-enabled* presence condition (except for hypothetical cases requiring *all* features enabled).

In practice, we must consider the effect of the feature model in the testing strategy. Due to mutually exclusive dependencies between features there is often no maximal configuration, but many locally maximal configurations. Moreover, because some features depend on others to be present, we often cannot disable features individually. The practical consideration of having a feature model is that enumerating the configurations to test requires selecting *valid* configurations only, which is a NP-complete problem itself. Yet, we expect that enumerating valid one-disabled configurations would be tractable, given the scalability of modern SAT solvers (hundreds of thousands of variables and clauses) and the size of real-world program families (only thousands of features).

Let us answer RQ2 now. It is a well known fact that an exponential number of variants makes it difficult for developers to understand and validate the code, but:

> **Conclusion 2:** *In addition to introducing an exponential number of program variants, variability additionally increases the complexity of bugs in multiple ways.*

Our analysis indicates that variability affects the complexity of bugs along several dimensions. Let us summarize them:

- Bugs occur because the implementation of features is intermixed, leading to undesired interactions, for instance, through program variables;
- Interactions occur between features from different subsystems, demanding cross-subsystem knowledge from Linux developers;
- Variability may be implicit and even hidden in alternative macro, function, and type definitions specified at spare locations;
- Variability bugs are the result of errors in the code, in the mapping, in the feature model, or any combination thereof;
- Further, each of these layers involves different languages (C, CPP, GNU MAKE and KCONFIG);
- Not all these bugs will be detected by maximal configuration testing due to interactions with *disabled* features;
- The existence of compiler errors in the Linux tree shows that conventional feature-insensitive tools are not enough to find variability bugs.

Table 4: Maximal vs *one-disabled* configuration testing. The cost is the number of configurations satisfying the formula, disregarding the feature model. Benefit shown as bug coverage for our sample.

test formula(s)	cost	benefit
$\bigwedge_{f\in\mathbb{F}} f$	$O(1)$	48% (20/42)
$\forall g{\in}\mathbb{F}$: $(\bigwedge_{f\in\mathbb{F}\setminus\{g\}} f) \wedge \neg g$	$O(\lvert\mathbb{F}\rvert)$	95% (40/42)
ψ	$O(2^{\lvert\mathbb{F}\rvert})$	100% (42/42)

6. THREATS TO VALIDITY

6.1 Internal Validity

Bias due to selection process. As we extract bugs from commits, our collection is biased towards bugs that were found, reported, *and* fixed. Since users run a small subset of possible Linux configurations, and developers lack feature-sensitive tools, potentially only a subset of bugs is found.

Further, our keyword-based search relies on the competence of Linux developers to properly identify and report variability in bugs. Note, however, that in Linux, variability is ubiquitous and often "hidden". For instance, the *ath3k* bluetooth driver module file contains no explicit variability, yet after variability-preserving preprocessing and macro expansion we can count thousands of CPP conditionals involving roughly 400 features. It is then unlikely that developers are always aware of the variability nature of the bugs they fix.

In order to further minimize the risk of introducing false positives, we do not record bugs if we fail to extract a sensible error trace, or if we cannot make sense of the pointers given by the commit author. This may introduce bias towards reproducible and lower complexity bugs.

Because of inherent bias of a detailed qualitative analysis method, we are not able to make quantitative observations about bug frequencies and properties of the entire population of bugs in the Linux kernel. Note, however, that we are able to make qualitative observations such as the existential confirmation of certain kinds of bugs (cf. Sect. 5). Since we only make such observations, we do not need to mitigate this threat (interestingly though, our collection still exhibits very wide diversity as shown in Sect. 5).

False positives and overall correctness. By only considering variability bugs that have been identified and fixed by Linux developers, we mitigate the risk of introducing false positives. We only take bug-fixing commits from the Linux stable branch, the commits of which have been reviewed by other developers and, particularly, by a more experienced Linux maintainer. In addition, our data can be independently verified since it is publicly available. The risk of introducing false positives is not zero though, for instance, commit `b1cc4c55c69` adds a nullity check for a pointer that is guaranteed not to be null[9]. It is tempting to think that the above indicates a variability bug, while in fact it is just a conservative check to detect a *potential* bug.

The manual analysis of a bug to extract an error trace is also error prone, especially for a language like C and a

[9] https://lkml.org/lkml/2010/10/15/30

complex large system such as Linux. Ideally, we should support our manual analysis with feature-sensitive program slicing, if it existed. A more automated approach based on bug-finders would not be satisfactory. Bug-finders are built for certain classes of errors, so they can give good statistical coverage for their particular class of errors, but they would not be able to assess the diversity of bugs that appear.

We derive simplified bugs based on manual slicing, filtering out irrelevant statements. We also abstract away C language features such as structs and dynamic dispatching via function pointers. While the process is systematic, it is performed manually and consequently error prone.

6.2 External Validity

Small number of bugs. The size of our sample speaks against the generalizability of the observations. The process of collecting and especially analyzing these 42 bugs costed several man-months, being unfeasible the study of a larger number of bugs. We expect that our database will continue to grow, also from third-party contributions, in the near future.

Single-subject study. We decided to focus exclusively on Linux, so our findings do not readily generalize to other highly configurable software. Yet, the size and nature of Linux make it a fair worst-case representative of software with variability. The type of bugs we found, especially memory errors, are expected in any piece of configurable system software implemented in C. In addition, the significance of the Linux kernel project itself justifies investigation of its errors, even if it limits generalizability.

7. RELATED WORK

Bug databases. ClabureDB is a database of bug-reports for the Linux kernel with similar purpose to ours [31], albeit ignoring variability. Unlike ClabureDB, we provide a record with information enabling non experts to rapidly understand the bugs and benchmark their analyses. This includes a simplified C99 version of each bug were irrelevant details are abstracted away, along with explanations and references intended for researchers with limited kernel experience. The main strength of ClabureDB is its size—the database is automatically populated using existing bug finders. Our database is small. We populated it manually, as no suitable bug finders handling variability exist (which also means that none of our bugs is covered in ClabureDB adequately).

Mining variability bugs. Nadi et al. mined the Linux repository to study *variability anomalies* [28]. An *anomaly* is a *mapping* error, which can be detected by checking satisfiability of Boolean formulas over features, such as mapping code to an invalid configuration. While we conduct our study in a similar way, we focus on a broader class of semantic errors in code, including data- and control-flow bugs.

Apel and coauthors use a model-checker to find feature interactions in a simple email client [3], using a technique known as *variability encoding* (*configuration lifting* [30]). Features are encoded as Boolean variables and conditional compilation directives are transformed into conditional statements. We focus on understanding the nature of variability bugs widely. This cannot be done with a model-checker searching for a particular class of interactions. Understanding variability bugs should lead to building scalable bug finders, enabling studies like [3] to be run for Linux in the future.

Medeiros et al. have studied *syntactic* variability errors [26]. They used a variability-aware C parser [23] to automate their bug finding and exhaustively find *all* syntax errors. They found only few tens of errors in 41 families, suggesting that syntactic variability errors are rare in committed code. We focus on the wider category of more complex *semantic* errors.

Nadi et al. mine feature dependencies in preprocessor-based program families to support synthesis of variability models for existing codebases [27]. They infer dependencies from nesting of preprocessor directives and from parse-, type-, and link-errors, assuming that a configuration that fails to build is invalid. Again, we consider a much wider class of errors than can be detected automatically so far.

Methodologically related work. Tian et al. studied the problem of distinguishing bug fixing commits in the Linux repository [34]. They use semi-supervised learning to classify commits according to tokens in the commit log and code metrics extracted from the patch contents. They significantly improve recall (without lowering precision) over the prior, keyword-based, methods. In our study most of time was invested in *analyzing* commits, not in finding potential candidates, so we found a simple keyword-based method sufficient.

Yin et al. collect hundreds of errors caused by misconfigurations in open source and commercial software [36] to build a representative set of large-scale software systems errors. They consider systems in which parameters are read from configuration files, as opposed to systems configured statically. More importantly, they document errors from the *user* perspective, as opposed to (our) *programmer* perspective.

Padioleau et al. studied collateral evolution of the Linux kernel, following a method close to ours [29]. Collateral evolution occurs when existing code is adapted to changes in the kernel interfaces. They identified potential collateral evolution candidates by analyzing patch fixes, and then manually selected 72 for a more careful analysis. Similarly, they classify and perform an in-depth analysis of their data.

8. CONCLUSION

We have identified 42 variability bugs, including 30 feature-interaction bugs, in the Linux kernel repository. We analyzed their properties and condensed each of these bugs into a self-contained C99 program with the same variability properties. These simplified bugs aid understanding the real bug and constitute a publicly available benchmark for analysis tools.

We observe that variability bugs are not confined to any particular type of bug, error-prone feature, or source code location (file, or subsystem) of the Linux kernel. Moreover, variability increases the complexity of bugs in Linux in several ways, besides the well known introduction of exponentially many code variants: *a*) the implementation of features is intermixed and undesired interactions can occur easily, *b*) these interactions can happen between features from different subsystems; and *c*) bugs can occur in the code, in the mapping, in the feature model, or any combination thereof.

Acknowledgments.

We thank kernel developers, Jesper Brouer and Matias Bjørling. Julia Lawall and Norber Siegmund provided useful suggestions. This work has been supported by The Danish Council for Independent Research under a Sapere Aude project, VARIETE.

9. REFERENCES

[1] S. Apel, D. Batory, C. Kästner, and G. Saake. *Feature-Oriented Software Product Lines*. Springer-Verlag, 2013.

[2] S. Apel, C. Kästner, A. Grösslinger, and C. Lengauer. Type safety for feature-oriented product lines. *Automated Software Engineering*, 17, 2010.

[3] S. Apel, H. Speidel, P. Wendler, A. von Rhein, and D. Beyer. Detection of feature interactions using feature-aware verification. In *Proceedings of the 26th IEEE/ACM International Conference on Automated Software Engineering (ASE'11)*, Lawrence, USA, 2011. IEEE Computer Society.

[4] T. Berger, R. Rublack, D. Nair, J. M. Atlee, M. Becker, K. Czarnecki, and A. Wasowski. A survey of variability modeling in industrial practice. In S. Gnesi, P. Collet, and K. Schmid, editors, *VaMoS*. ACM, 2013.

[5] T. Berger, S. She, R. Lotufo, A. Wasowski, and K. Czarnecki. A study of variability models and languages in the systems software domain. *IEEE Trans. Software Eng.*, 39(12).

[6] E. Bodden, T. Tolêdo, M. Ribeiro, C. Brabrand, P. Borba, and M. Mezini. SPL$^{\text{LIFT}}$ - statically analyzing software product lines in minutes instead of years. In *PLDI'13*, 2013.

[7] E. Bounimova, P. Godefroid, and D. Molnar. Billions and billions of constraints: Whitebox fuzz testing in production. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, Piscataway, NJ, USA, 2013. IEEE Press.

[8] D. Bovet and M. Cesati. *Understanding the Linux Kernel*. O'Reilly Media, 2005.

[9] C. Brabrand, M. Ribeiro, T. Tolêdo, J. Winther, and P. Borba. Intraprocedural dataflow analysis for software product lines. *Transactions on Aspect-Oriented Software Development*, 10, 2013.

[10] W. R. Bush, J. D. Pincus, and D. J. Sielaff. A static analyzer for finding dynamic programming errors. *Softw. Pract. Exper.*, 30(7), June 2000.

[11] M. Calder, M. Kolberg, E. H. Magill, and S. Reiff-Marganiec. Feature interaction: A critical review and considered forecast. *Comput. Netw.*, 41(1), 2003.

[12] A. Classen, P. Heymans, P.-Y. Schobbens, and A. Legay. Symbolic model checking of software product lines. In *ICSE*, 2011.

[13] A. Classen, P. Heymans, P.-Y. Schobbens, A. Legay, and J.-F. Raskin. Model checking lots of systems: efficient verification of temporal properties in software product lines. In *ICSE'10*, Cape Town, South Africa, 2010. ACM.

[14] K. Czarnecki and K. Pietroszek. Verifying feature-based model templates against well-formedness OCL constraints. In *Proceedings of the 5th international conference on Generative programming and component engineering*, GPCE '06, New York, NY, USA, 2006. ACM.

[15] N. Dor, M. Rodeh, and M. Sagiv. CSSV: Towards a Realistic Tool for Statically Detecting All Buffer Overflows in C. *SIGPLAN Not.*, 38(5), 2003.

[16] D. Evans. Static detection of dynamic memory errors. *SIGPLAN Not.*, 31(5), 1996.

[17] A. Gruler, M. Leucker, and K. D. Scheidemann. Modeling and model checking software product lines. In *FMOODS*, 2008.

[18] G. Holl, M. Vierhauser, W. Heider, P. Grünbacher, and R. Rabiser. Product line bundles for tool support in multi product lines. In *VaMoS*, 2011.

[19] D. Hovemeyer and W. Pugh. Finding more null pointer bugs, but not too many. In *Proceedings of the 7th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, PASTE '07, New York, NY, USA, 2007. ACM.

[20] K. Kang, S. Cohen, J. Hess, W. Nowak, and S. Peterson. Feature-oriented domain analysis (FODA) feasibility study. Tech. Rep. CMU/SEI-90-TR-21, CMU-SEI, 1990.

[21] C. Kästner. *Virtual Separation of Concerns: Toward Preprocessors 2.0*. PhD thesis, Marburg, Germany, 2010.

[22] C. Kästner and S. Apel. Type-checking software product lines - a formal approach. In *Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering (ASE'08)*, L'Aquila, Italy, 2008.

[23] A. Kenner, C. Kästner, S. Haase, and T. Leich. Typechef: Toward type checking #ifdef variability in c. In *Proceedings of the 2Nd International Workshop on Feature-Oriented Software Development*, FOSD '10, New York, NY, USA, 2010. ACM.

[24] C. H. P. Kim, E. Bodden, D. Batory, and S. Khurshid. Reducing configurations to monitor in a software product line. In *1st International Conference on Runtime Verification (RV)*, volume 6418 of *LNCS*, Malta, 2010. Springer.

[25] R. Love. *Linux Kernel Development*. Developer's Library. Pearson Education, 2010.

[26] F. Medeiros, M. Ribeiro, and R. Gheyi. Investigating preprocessor-based syntax errors. In *Proceedings of the 12th International Conference on Generative Programming: Concepts & Experiences*, GPCE '13, New York, NY, USA, 2013. ACM.

[27] S. Nadi, T. Berger, C. Kästner, and K. Czarnecki. Mining configuration constraints: Static analyses and empirical results. In *36th International Conference on Software Engineering (ICSE'14)*, 2014.

[28] S. Nadi, C. Dietrich, R. Tartler, R. C. Holt, and D. Lohmann. Linux variability anomalies: what causes them and how do they get fixed? In T. Zimmermann, M. D. Penta, and S. Kim, editors, *MSR*. IEEE / ACM, 2013.

[29] Y. Padioleau, J. L. Lawall, and G. Muller. Understanding collateral evolution in linux device drivers. In *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006*, EuroSys '06, New York, NY, USA, 2006. ACM.

[30] H. Post and C. Sinz. Configuration lifting: Verification meets software configuration. In *Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering (ASE'08)*, L´Aquila, Italy, 2008. IEEE Computer Society.

[31] J. Slaby, J. Strejček, and M. Trtík. ClabureDB: Classified Bug-Reports Database. In R. Giacobazzi,

J. Berdine, and I. Mastroeni, editors, *Verification, Model Checking, and Abstract Interpretation*, volume 7737 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2013.

[32] The Institute of Electrical and Eletronics Engineers. IEEE Standard Glossary of Software Engineering Terminology. IEEE Standard, 1990.

[33] T. Thüm, S. Apel, C. Kästner, I. Schaefer, and G. Saake. A classification and survey of analysis strategies for software product lines. *ACM Computing Surveys*, 2014.

[34] Y. Tian, J. Lawall, and D. Lo. Identifying linux bug fixing patches. In *Proceedings of the 2012 International Conference on Software Engineering*, ICSE 2012, Piscataway, NJ, USA, 2012. IEEE Press.

[35] D. Wagner, J. S. Foster, E. A. Brewer, and A. Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *NDSS*. The Internet Society, 2000.

[36] Z. Yin, X. Ma, J. Zheng, Y. Zhou, L. N. Bairavasundaram, and S. Pasupathy. An empirical study on configuration errors in commercial and open source systems. In *Proc. of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, New York, NY, USA, 2011. ACM.

42个Linux内核中的可变性Bug：定性分析

Iago Abal
iago@itu.dk

Claus Brabrand
brabrand@itu.dk

Andrzej Wąsowski
wasowski@itu.dk

哥本哈根大学

Rued Langgaards Vej 7, 2300 Copenhagen S, Denmark

摘要

功能敏感的验证是对一个程序系列的指数级变量进行有效分析。 然而，研究人员缺乏真实大型系统中发生的由变异引发的具体缺陷的例子。 这样的错误集合是面向目标的研究的要求，通过对真正的错误进行测试来评估功能敏感分析的工具实现。 我们提供了一个定性研究，其中包括从bug内核提交到Linux内核存储库收集的42个可变性错误。 我们分析每个错误，并将结果记录在数据库中。 另外，我们提供了自包含的简化C99版本的错误，便于理解和工具评估。 我们的研究为大型C软件系统中可变性错误的性质和发生提供了见解，并显示了可变性如何影响并增加了软件错误的复杂性。

类别和主题描述

D.2.0 [软件工程]：一般； D.2.5 [软件工程]：测试和调试

关键词

Bug；特征交互； Linux；软件可变性

1. 介绍

许多软件项目必须应对大量的变化。 在采用软件产品线方法的项目中[1]可变性用于调整个体软件产品的开发以适应特定的市场。 一个相关但不同类的项目开发了高度可配置的系统，如Linux内核，其中配置选项（这里称为功能[20]用于根据特定用户的需求定制功能性和非功能性属性。 高度可配置的系统可以变得非常大并包含大量的功能。 具有数千个特征的工业系统的报告存在[4]和广泛的开源示例详细记录[5].

ASE'14，2014年9月15日至19日，瑞典韦斯特罗斯。

ACM 978-1-4503-3013-8 / 14/09 ... $ 15.00。
http://dx.doi.org/10.1145/2642937.2642990。

可配置系统中的功能以不重要的方式进行交互，以影响其他功能。 当这种相互作用是无意识的时候，它们诱发的错误在某些配置中表现出来，但在其他配置中则不然，或者在不同配置中表现出不同的错误。 基于标准程序分析技术的分析仪可以找到单个配置中的缺陷。 但是，由于配置的数量是特征数量的指数，所以分别分析每个配置是不可行的。

以家庭为基础[33]分析是一种功能敏感性分析，通过将所有可配置的程序变体作为一个单独的分析单元来解决这个问题，而不是单独分析各个变体。 为了避免重复工作，共同部分进行一次分析，分析仅针对变体之间的差异。 最近，各种基于家庭的扩展的经典静态分析[2, 6, 9, 14, 22, 24]和模型检查[3, 12, 13, 17, 30]基于技术已经开发出来。

到目前为止，大部分研究都集中在固有的可扩展性问题上。 但是，我们仍然没有证据表明这些扩展适用于实际场景中的特定目的。 特别是，很少有人了解高度可配置系统中出现哪种类型的错误，以及它们的变化特性是什么。 获得这样的理解将有助于在实际问题中研究基于家庭的分析。

对变异性错误的复杂性的理解在从业者和可用工件中并不常见。 虽然错误报告比比皆是，但对这些错误是由功能交互引起的还是知之甚少。 很多时候，由于像Linux这样的大型项目的复杂性以及缺乏对功能敏感的工具支持，开发人员并不完全意识到影响他们工作的软件的功能。 结果，错误出现并得到修复，很少或没有指示其变化程序的起源。

这项工作的目标是了解在一个大型高度可配置系统Linux内核中发生的变异性错误（包括特征交互错误）的复杂性和性质。 我们通过定性的深入分析和42个此类错误的记录来解决这个问题。 我们做出以下贡献：

- *识别Linux内核中的42个可变性错误，包括针对非专业人员的深入分析和演示。*
- *一个包含我们分析结果的数据库，包含关于每个错误的详细数据记录。*

这些错误包括C软件中常见的错误类型，商品，并涵盖不同类型的功能交互。

我们打算在未来用这个系列增加收藏研究界的帮助。 目前的版本是可在 http://VBDb.itu.dk.

- *自包含简化的C99版本的所有错误。 这些*便于理解根本原因，并且可以用于测试较小规模的bug查找器。
- *对错误集合进行汇总反思。*提供有关由彗星引起的错误的性质的洞察在像Linux这样的大型项目中进行功能交互。

我们采用定性的手动分析方法有三个原因。 首先，针对Linux内核扩展的基于家族的自动化分析工具不存在。事实上，没有这项研究，不清楚应该建立什么工具。 其次，预处理后对各个变体使用传统的（不是基于家庭的）分析工具不能进行扩展（如果应用的是详尽的）或者发现错误的概率较低（如果通过随机抽样来应用）。

第三，使用工具搜索错误只会发现这些工具覆盖的情况，而我们有兴趣广泛探索可变性错误的性质。 考虑到所收集的材料，我们了解到变异性错误非常复杂，涉及编程语言语义的许多方面，它们分布在Linux项目的大部分部分，涉及远程位置的多个特征和跨度代码。 检测这些错误对于人员和工具来说都很困难。 一旦能够捕捉这些错误的功能敏感分析可用，那么进行广泛的定量实验将会很有趣，以确认我们的定性直觉。

我们将我们的工作指导给程序分析和错误查找工具的设计人员。 我们认为，收集错误可以通过以下几种方式激发它们：（i）它将为分析提供一套具体的，详细描述的挑战，（ii）它将作为评估其工具的基准，以及（iii）它将极大地加速新技术的设计，因为它们可以在简化的Linux独立漏洞上进行尝试。 在评估中使用大量软件中的实际错误可以帮助调整分析精度，并鼓励设计人员在分析中支持某些语言结构。

我们介绍Sect. 2中的基本背景。 该方法详见Sect. 3。第 4–5 描述分析：首先考虑维度，然后是总体观察。 我们完成调查威胁的有效性（Sect. 6），相关工作（Sect. 7）和结论（Sect. 8).

2. 背景

我们使用术语“软件错误”来指代IEEE标准软件工程术语表定义的错误和错误[32]。 故障（缺陷）是软件中的错误指令，由于人为错误而引入代码中。 故障导致错误，即不正确的程序状态，例如指针在不应该为空时为空。 在这项工作中，我们收集了表现为运行时间故障（通常是内核恐慌）的错误，以及构建特定内核配置时发现的缺陷。 虽然后者可能看起来无害（例如未使用的变量），但我们认为它们可能是严重误解的副作用，可能导致其他错误。

功能是核心软件以外的功能单元[11]。 核心（基本变体）实现基本

```
1  #include <stdlib.h>
3  void foo (int a)                                  →(5)
4      {printf ("%d                  //ERROR         (6) ×
5      \n", 2 / a);
7  }                                                 ⇒(1)
8  int main                          //START         (2)
9      (void)                        //DISABLED      |
10     {int x =                                      |
11     1;                                            |
12     #ifdef CONFIG_INCR            //ENABLED       |
13     x = x + 1;                                    (3)
14     ##万一                                        ↓
15     #ifdef CONFIG_DECR                            (4)→
16     x = x - 1;
       #endi
```

图1：一个程序系列和一个错误的例子。

功能存在于程序系列的任何变体中。 功能（配置）的不同选择定义了一组程序变体。 通常，两个功能不能同时启用，或者一个功能需要启用另一个功能。 使用特征模型来指定特征依赖性[20]（或者一个决策模型[18]），这里用ψFM表示； 有效地限制了定义合法配置的功能。

基于预处理器的程序系列[21]将特征与宏符号相关联，并将它们的实现定义为由特征符号上的约束保护的静态条件代码。 与特征（配置选项）关联的宏符号通常受命名约定的约束，例如，在Linux中，这些标识符应以CONFIG_为前缀。 我们在本文中遵循Linux约定。 数字 1 提供了一个使用两种功能INCR和DECR的小型基于预处理器的C程序系列。 第10和13行的陈述是有条件的。 假设一个不受限制的特征模型（ψFM= true），该图定义了四种不同的变体族。

代码片段的存在条件φ是一个最小值（通过引用变量的数量）特征的布尔公式，指定代码包含在编译中的配置子集。 存在条件的概念自然延伸到其他实体；例如，错误的存在条件指定发生错误的配置的子集。 用 κ 表示的具体配置也可以写成布尔约束 – 特征文字的连接。因此存在条件 φ 的代码片段存在于配置 κ iff $\kappa \mathrm{F} \varphi$ 中。 例如，考虑第13行的递减语句，该语句具有存在条件DECR，因此它是配置 $\kappa 0 =$ INCR $\wedge$ DECR和 $\kappa 1 =$ INCR $\wedge$ DECR的一部分。

功能可以影响其他功能提供的功能，这是一种被称为特征交互的现象，可以是有意或无意的。在我们的例子中，这两个特征相互作用，因为它们都修改并使用相同的程序变量x。 在调用foo之前，启用INCR或DECR或两者都会导致x的值不同。

由于可变性，错误可能发生在某些配置中，但在其他配置中不会发生，并且也可能在不同的变体中以不同的方式表现。 如果在一个或多个配置中发生错误，并且至少在一个其他配置中没有发生错误，我们称之为可变性错误。 数字 1 展示了其中的一个pro-当我们试图除以零时，我们示例族中的克变体，即$\kappa 0$，将在第4行崩溃。 由于此错误并未在任何其他变体中出现，因此它是一个可变性错误 – 存在条件为INCR $\wedge$ DECR。

程序系列实现通常在概念上分为三层：问题空间（通常是特征模型），解决方案空间实现（例如C代码）以及问题和解决方案空间（Linux中的构建系统和cpp）之间的映射。 我们展示了如何通过我们的运行示例中的解释分别在每个层中修正零除错误。 我们用统一的差异格式（diff -U0）显示代码的更改。

*修*复代码。 如果函数foo应该接受任何int值，那么通过适当地处理零作为输入来修正该bug。

```
@@ -4 +4,4 @@
- printf ("%d \n", 2 / a) ;
+ if (a! = 0)
+   的printf ("%d \n" 个, 2 /
A) ;
+其他
+   的printf ("南\ n") ;
```

*修*复映射。 如果我们假设函数foo不应该用零参数调用，则可能的解决方法是只有在启用了DECR和INCR时才减小\x。

```
@@ -12 +12 @@
- #ifdef CONFIG_DECR
+ #if defined (CONFIG_DECR) && defined (CONFIG_INCR)
```

*修*复模型。 如果错误是由非法交互引起的，我们可以在特征模型中引入依赖关系，以防止错误配置 κ 0。 例如，让DECR仅在启用INCR时可用。 假设特征模型 ψ FM= DECR→INCR禁止 κ 0。

3. 方法

*目*的。 我们的目标是定性理解在大型高度可配置系统中发生的变异性错误（包括功能交互错误）的复杂性和性质：Linux内核。 这包括解决以下研究问题：

- RQ1：可变性错误是否限于任何特定类型的错误，“容易出错”的功能或特定位置？
- RQ2：变异性以什么方式影响软件缺陷？

*学*科。 我们研究Linux内核，采用Linux稳定的Git[1]存储库作为分析单元。 Linux可能是最大的高度可配置的开源系统。 它有大约一千万行代码和超过一万个功能。至关重要的是，有关Linux错误的数据可以自由使用。我们可以免费访问错误跟踪器[2]，源代码和更改历史记录[3]，并在邮件列表上公开讨论[4]（LKML）和其他论坛。还有关于Linux开发的书籍[8, 25]了解错误修复程序时所需的宝贵资源。 获得特定领域的知识对定性分析至关重要。

我们将重点放在已提交到Linux存储库的错误上。 这些错误已经公开讨论（通常在LKML上）并且被内核开发人员确认为实际错误，所以关于错误修复性质的信息是可靠的，并且我们将包含虚构问题的可能性降至最低。

*方*法。 我们的方法有三个部分：首先，我们确定内核历史中的变异性错误。 第二，

[1] http://git-scm.com/
[2] https://bugzilla.kernel.org/
[3] http://git.kernel.org/cgit/linux/kernel/git/
[4] https://lkml.org/

CONFIG_f ID配置配置选项
如果如果f id是[en | dis]，则当f id是[not]?时-设置f id是[not]?
当f id是[en | dis]时

(a) 消息过滤器。

#if
##其他
#elif指令
#endif选择f id
config f id
取决于f id

(b) 内容过滤器。

图2：正则表达式选择configura-（a）信息，（b）内容； f id缩写[A-ZO-9_]，匹配特征标识符。

错误
修复
oops
警告
错误
不安全
无效违规
结果追踪
内核恐慌

(a) 通用错误过滤器。

void *未使用的溢出未定义的双重锁内存泄漏未初始化的悬挂指针null [指针]?解除引用
...

(b) 特定的bug过滤器。

图3：正则表达式选择bug修复提交：（a）通用的，（b）特定的问题

我们分析并解释它们。 最后，我们反思汇总的材料以回答我们的研究问题。

*第*1部分：查找可变性错误。 我们已经通过Linux提交进行了一次半自动搜索，以通过历史错误修复发现可变性错误。截至2014年4月，Linux存储库有超过400,000个提交，这排除了每个提交的手动调查。 我们通过以下步骤搜索了变异性错误的提交：

1. *选*择变异相关的提交。 我们保留与图5的正则表达式匹配的提交。 2。 表达式在图。 2(a) 识别作者的消息将提交与特定功能相关联的提交。 那些在图。 2(b) 识别引入对特征映射或特征模型的更改的提交。 我们拒绝合并，因为此类提交不会进行更改。 这一步按数万个提交的顺序进行选择。
2. *选*择错误修复提交。 我们仅限于提交修正错误，匹配表示提交消息中的错误的正则表达式（参见图3）。3）。 取决于感兴趣的关键字，此步骤可以从数千次提交中选择，仅有几十次或更少次。
3. *手*动审查。 我们读取提交消息并检查提交引入的更改以消除明显的误报。 我们在前两次搜索中按命中次数提交提交，然后按照优先级排列非常复杂的提交（给出提交消息中提供的信息以及修补程序修改的行数）。

*第*2部分：分析。 方法的第二部分比第一部分要复杂得多。 对于识别的每个可变性错误，我们手动分析提交消息，修补程序修复程序和实际代码，以便了解该错误。 当需要更多的上下文时，

我们发现并遵循相关的LKML讨论。 代码检查由ctags支持[5]和Unix grep实用程序，因为我们缺乏对功能敏感的工具支持。

1. *错*误的语义。 对于每个可变性错误，我们想要了解错误的原因，对程序语义的影响以及两者之间的关系。 这通常需要理解内核的内部工作原理，并将这种理解转化为更广泛的受众可访问的通用编程语言术语。 作为此过程的一部分，我们尝试识别相关的运行时执行跟踪并收集有关在线错误的可用信息的链接。

2. *可*变性相关的属性。 我们确定错误的存在情况（根据配置选择的先决条件）以及它在哪里修复（代码中，特征模型中或映射中）。

3. *简*化版本。 最后但并非最不重要的一点是，我们通过简化版本的bug来浓缩我们的理解。 这可以解释最初的错误，并且是评估工具的一个有趣的基准。

我们分析了上述步骤之后的Linux bug，并将创建的报告存储在公开的数据库中。 我们正在寻找一个足够多样化的样本，一旦有可能回答我们的两个研究问题，就停止在42个问题上。 报告的详细内容在第二部分进行了解释。 4.

*第*3部分：数据分析和验证。 我们反思收集的数据集以便找到我们的研究问题的答案。 这一步由一些定量数据支持，但重要的是，我们没有对Linux中变异性错误的总体数量作出任何定量结论（鉴于上述研究方法，这些结论是不健全的）。 它纯粹表征了所获得数据集的多样性。 这允许以汇总的方式呈现整个错误集合（请参见Sect。 5).

最后，为了减少偏见，我们在接受全职专业Linux内核开发人员访谈时遇到了我们的方法，发现和假设。

4. 分析尺寸

我们首先选择一些可变性错误的属性来了解，分析和记录错误报告。 这些将在下面进行描述，并由我们的数据库中的数据来举例说明 我们在图中显示一个例子记录。 4，在驱动程序中发现一个空指针解引用错误，该错误被追溯到特征模型和映射中的错误。

*B*ug类型（类型）。 为了理解变异性错误的多样性，我们根据Common Weakness Enumeration（CWE）建立了错误类型，[6]-编号软件弱点和漏洞的目录。 我们遵循CWE，因为它在[31]。但是，由于CWE主要关注安全性，因此我们不得不扩展一些其他类型的错误，包括类型错误，Linux API的错误使用等。所获得的分类中的错误类型在Tbl中列出。 1；我们的补充在最右边的列中没有标识符。 该

[5]http://ctags.sourceforge.net/
[6]http://cwe.mitre.org/

错误类型直接指出可以使用哪种分析和程序验证技术来解决内核中发现的错误。 例如，内存错误的类别（Tbl。1）几乎直接映射到各种程序分析：对于空指针[10, 16, 19]，缓冲区溢出[7, 15, 35]，内存泄漏[10, 16]等。

*错*误描述（descr）。 理解错误需要用通用的软件工程术语来重新定义它的本质，以便对于非内核专家来说这个错误变得可以理解。 我们通过深入研究错误并追踪其他可用资源（例如邮件列表讨论，可用书籍，提交消息，文档和在线文章）来获得这样的描述。 无论何时使用Linux术语是不可避免的，我们都会提供必要背景的链接。 获得描述通常是不平凡的。 例如，我们的数据库中的一个错误（commit eb91f1d0a53）被修复了以下提交信息：

使用CONFIG_SLAB进行编译时，在启动时修复以下警告：

```
[0.000000] ------------ [切入] ------------
[0.000000]警告：在kernel / lockdep.c：2282 lockdep_trace_alloc
+ 0x91 / 0xb9 ()[0.000000]硬件名称：[0.000000]链接到的模块：
[0.000000] Pid：0，comm：交换器未被污染
2.6.30#491 [0.000000]呼叫跟踪：
[0.000000] [<ffffffff81087d84>]？ lockdep_trace_alloc + 0x91 / 0xb9
...
```

在我们的数据库中总结为：

由于使用标志GFP_WAIT调用kmalloc（）会导致警告并启用中断

SLAB分配器由start_kernel（）初始化，禁止中断。在这个过程的后面，setup_cpu_cache（）执行每CPU kmalloc高速缓存初始化，并将尝试为通过GFP_KERNEL标志的这些高速缓存分配内存。 这些标志包括GFP_WAIT，它允许进程在等待内存可用时进入休眠状态。 正如我们所说的，由于在SLAB初始化期间中断被禁用，这可能导致死锁。 启用LOCKDEP和其他调试选项将检测并报告这种情况。

我们在描述中添加一行标题，这里以粗体显示，以帮助识别和列出错误。

*程*序配置（config）。 为了确认一个错误确实是一个可变性错误，我们在它出现的状态下进行调查。 这可以排除无条件出现的错误，并且可以进一步调查错误的可变性属性，例如启用错误的依赖关系的功能数量和性质。 我们的例子bug（图。 1）在DECR被使能但INCR被禁止时存在。 图1中捕获的Linux错误。 4(b) 需要启用TWL4030_CORE，并禁用OF_IRQ，以显示错误行为（请参阅config条目左边部分）。

*B*ug-Fix Layer（图层）。 我们分析修复提交来确定错误的来源是在代码中，在特征模型中还是在映射中。了解这一点有助于指导未来对构建诊断工具的研究：是分析模型，映射还是代码所需的工具？ 哪里最好报告错误？

图。 4已被固定在模型和映射中（参见图1）。 5）。固定提交断言：首先，TWL4030_CORE不应该依赖IRQ_DOMAIN（在模型中固定），其次，将变量ops分配给&irq_domain_simple_ops是IRQ_DOMAIN的一部分代码而不是OF_IRQ（固定在映射中）。

*错*误跟踪（跟踪）。 我们手动分析导致错误状态的执行轨迹。 切片工具不容易

```
类型：                空指针解除引用
descr: 如果IRQ_DOMAIN, OF_IRQ上的空指针被取消引用。
       在TWL4030驱动程序中，尝试注册一个IRQ域
       一个NULL操作结构：注册一个IRQ域时ops被去引
       用，但是当OF_IRQ时这个字段只设置为一个非空
       值。
config: TWL4030_CORE && ! OF_IRQ
bugfix:
   repo: git://git.kernel.org/pub/.../linux-
   stable.git hash:
   6252547b8a7acced581b649af4ebf6d65f63a34b
   layer: model, mapping
跟踪：
   。  dyn-call drivers / mfd / twl-
   core.c: 1190: twl_probe ()
   。  1235: irq_domain_add (&
   domain) ;
   .. call kernel / irq /
   irqdomain.c: 20:
   irq_domain_add ()
   ...调用include / linux /
   irqdomain.h: 74:
```

(a) Bug记录。

```
2   #include <stdlib.h>
4   #ifdef CONFIG_TWL4030_CORE          //启用
5   #define CONFIG_IRQ_DOMAIN
6   ##万一
8   #ifdef CONFIG_IRQ_DOMAIN            //启用
9   int irq_domain_simple_ops = 1;
11  void irq_domain_add (int *
12      ops) {int irq = * ops;          //错误
13  }
14  ##万一
16  #ifdef CONFIG_TWL4030_CORE          //启用
17  void twl_probe
18      () {int *
19      ops = NULL;                     //DISABLED
20      #ifdef CONFIG_OF_IRQ
21      ops =&
22      irq_domain_simple_ops;
23      #endif
24      irq_domain_add
25       (ops) ;
26  }
27  ##万一                              //启用
28  int main (void) {
29      #ifdef CONFIG_TWL4030_CORE
30      twl_probe ();
```

(6) (7)× (3) (4) (5) (1) (2)

(b) 简化版本。

图4：错误6252547b8a7：记录示例和简化版本。

```
@@ -2,8 +2,4 @@
 #include <stdlib.h>
-#ifdef CONFIG_TWL4030_CORE
- #定义CONFIG_IRQ_DOMAIN
-##万一
-
 #ifdef CONFIG_IRQ_DOMAIN
 int irq_domain_simple_ops = 1;
@@ -16,9 +12,9 @@
 #ifdef CONFIG_TWL4030_CORE
 void twl_probe () {
+#ifdef CONFIG_IRQ_DOMAIN
   int *ops = NULL;
-#ifdef CONFIG_OF_IRQ
   ops =&
   irq_domain_simple_ops;
+ irq_domain_add (ops) ;
   ##万一
-  irq_domain_add
   (ops) ;
```

图5：修正了简化错误6252547b8a7。 该补丁以统一的差异格式（diff -U2）提供。

用于这些目的，因为它们都不能正确处理静态预处理器指令。 构建跟踪可以让我们了解bug的性质和复杂性。 记录失败的跟踪可以让其他研究人员更快地理解错误。

我们的跟踪中有两种类型的条目：函数调用和语句。 如果通过函数指针调用函数，则函数调用条目可以是静态（标记调用）或动态（dyn调用）。 声明条目突出显示程序状态中的相关更改。 每个条目都以非空的点序列开始，表示函数调用的嵌套，接着是函数定义（文件和行）或语句（仅限行）的位置。 出现错误的语句用ERROR标签标记。 在图1中， 4(a) 跟踪从驱动程序加载函数开始（twl_probe）。 这是从i2c_device_probe处调用的drivers / i2c / i2c-core.c，通用加载函数I2C[7]驱动程序，通过函数指针（驱动程序 - >探针）。 对irq_domain_add的调用通过引用传递全局声明的结构域，并且此结构的ops字段（现为别名* d）被解引用（d-> ops-> to_irq）。

域的ops字段没有显式初始化，所以它默认情况下设置为null（如C标准所规定）。

因此，当禁用OF_IRQ时，上述错误跟踪明确地识别从驱动程序加载到空指针解引用的路径。 如果已启用OF_IRQ，则在调用irq_domain_add之前ops字段将被正确初始化。

简化的错误。 最后但并非最不重要的是，我们综合了捕获其最重要属性的简化版本的bug。 我们编写一个独立于内核代码的小型C99程序，它具有相同的基本行为（和相同的问题）。 对于愿意尝试程序验证和分析工具而不与Linux构建基础架构，大型头文件和相关库以及最重要的是不理解内核内部工作方式的研究人员，可以轻松获得获得的简化错误。 此外，整套简化错误构成了一个易于访问的基准套件，该套件源于大型软件系统中发生的实际错误，可用于评估较小规模的错误查找工具。 简化的错误是从错误跟踪中系统地导出的。 沿着这条轨迹，我们保留了相关的语句和控制流结构，映射信息和函数调用。 我们保留特征，函数和变量的原始标识符。 然而，我们通过函数指针，结构类型，无效指针，强制类型以及任何特定于Linux的类型来抽象出动态分派，当这与bug无关时。 当功能之间存在依赖关系时，我们使用#define强制进行有效配置。 这种功能依赖关系的编码具有使得简化的错误文件自包含。

[7]微控制器应用中使用的串行总线协议。

数字 4(b) 显示了我们正在运行的带空指针解引用的示例bug的简化版本。 第4-6行将编码TWL4030_CORE对IRQ_DOMAIN的依赖关系，以防止无效配置TWL4030_CORE IRQ_DOMAIN。 我们鼓励读者从第26行的main开始研究导致崩溃的执行轨迹。这需要几分钟的时间，而不是通常需要很多时间才能正常理解Linux内核错误。 注意

跟踪将在存在条件下进行解释

从错误记录（决定在评论中指定）

#if条件旁边）。

可追溯性信息。 我们存储了存储库的URL，其中应用了错误修复，提交哈希以及有关该错误的相关上下文信息的链接，以支持对我们分析的独立验证。

5. 数据分析

为了解决研究问题，我们反映了所收集的全部信息，并提供了下面提供的详细意见。 在下文中，我们有时会使用数字汇总数据。 这些数字仅用于描述目的 - 不应从中得出统计结论（我们使用灰色字体强调这一点）。

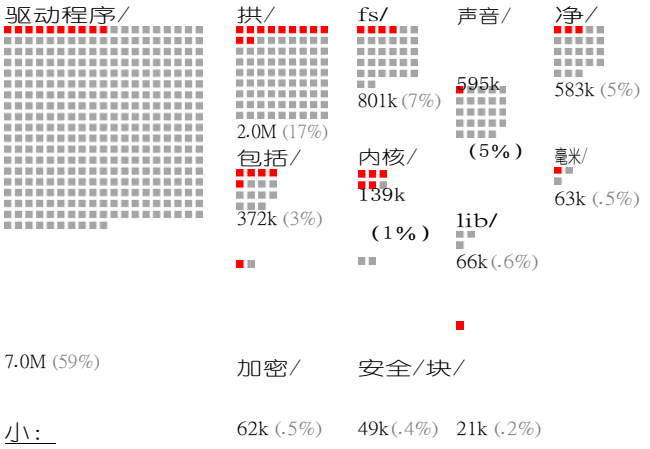
我们首先介绍支持我们的第一个研究问题RQ1的观察结果：

观察1：可变性错误不限于任何特定类型的错误。

表 1 列出了我们找到的错误类型以及集合中的出现频率。例如，15个错误已被分类在内存错误的类别下，其中四个是空指针解引用。 我们注意到，变异bug涵盖了来自各种不同类型的bug

表1：42个错误中的错误类型 第一列给出了我们收集中这些错误的频率。

15	**内存错误：**	**CWE ID**
4	空指针解引用	476
3	缓冲区溢出	120
3	读出界限	125
2	内存不足	-
1	内存泄漏	401
1	免费后使用	416
1	写在只读	-
10	**编译器警告：**	**CWE ID**
5	未初始化的变量	457
2	不兼容的类型	843
1	未使用的功能（死代码）	561
1	未使用的变量	563
1	void指针解引用	-
7	**输入错误：**	**CWE ID**
5	未定义的符号	-
1	未申报的标识符	-
1	错误的参数数量	-
7	**断言违规：**	**CWE ID**
5	致命的断言违反	617
2	非致命的断言违反	617
2	**API违规：**	**CWE ID**
1	Linux sysfs API违规	-
1	双锁	764
1	**算术错误：**	**CWE ID**
1	数字截断	197



- virt / (6.8k), ipc / (6.4k), init / (2.0k) 和usr / (0.6k)。

基础设施：

- 工具/ (102k), 脚本/ (44k) 和样本/ (2.1k)。

图6：截至2014年3月，主要Linux目录中42个错误的位置。每个正方形代表
25000行代码。 LOC的精确数量及其占总数的百分比在平方下方给出。 红色（黑色）方块象征着其中一个错误的发生。

类型错误，数据流错误（如未初始化的变量），锁定策略违规（双锁）。

我们在编译时发现了17个错误，类型错误和编译器警告。尽管进行了编译器检查，但这些错误首先已被录入到存储库中。 由于编译器错误不容易被忽略，因此我们将此作为提交者（以及接受它的维护者）无法找到该错误的证据，因为他们将代码编译为不显示它的配置（编译器检查不是以家庭为基础）。

观察2：可变性错误似乎不限于特定的“容易出错”的特征。

表 2 显示了错误中涉及的功能的完整列表：从调试选项（例如，QUOTA_DEBUG和LOCKDEP）到设备驱动程序（例如TWL4030_CORE和ANDROID）等共78种定性不同的功能，到网络协议（例如VLAN_8021Q和IPV6），计算机体系结构（例如PARISC和64BIT）。 其中三个错误涉及三个功能，九个功能发生在两个错误中，其余66个仅涉及一个错误。

表2：错误中涉及的功能。

64BIT	IP_SCTP	S390
ACPI_VIDEO	JFFS2_FS_WBUF验证S390	PRNG
ACPI_WMI	KGDB	SCTP_DBG_MSG
AMIGA_Z2RAM	Kprobes的	安全
ANDROID	KTIME标志	SHMEM
ARCH_OMAP2420	LBDAF	SLAB
ARCH_OPAM3	LOCKDEP	SLOB
ARM_LPAE	MACH_OMAP_H4	SMP背光类设备
	模块卸载	SND_FSI_AK4642
BCM47XX	NETPOLL	SND_FSI_DA7210
BDI开关	NUMA	SSB_DRIVER_EXTIF
BF60x	的	STUB_POULSBO
BLK_CGROUP	IRQ	SYSFS
CRYPTO_BLKCIPHER	PARISC	TCP_MD5SIG
CRYPTO测试	PCI	TMPFS
DEVPTS多个实例PM		跟踪IRQFLAGS
DISCONTIGMEM	PPC64	示踪
DRM_I915	PPC 256K页	TREE_RCU
EP93XX_ETH	PREEMPT	TWL4030核心
EXTCON	PROC_PAGE_MONITOR	UNIX98_PTYS
FORCE_MAX_ZONEORDER=11	证明锁定	VLAN_8021Q
HIGHMEM	QUOTA_DEBUG	涡流
热插拔	RCU_CPU_STALL_INFO	X86
I2C	RCU_FAST_NO_HZ	X86_32
IOSCHED_CFQ	调节器MAX8660	XMON
IPV6	REISERFS FS安全区域DMA	

观察3：可变性错误不限于任何特定位置（文件或内核子系统）。

数字 6 显示错误所在的子系统以及截至2014年3月的每个子系统的相对大小
– 我们通过目录来近似子系统。 每个子系统的大小是用代码行（LOC）来衡量的

由cloc报告的LOC（对于任何语言）的总和[8]（版本1.53）。 例如，有六个方块，内核/子系统约有150 KLOC，约占总数的1%
Linux代码。 叠加到尺寸可视化上，该图还显示了错误发生在哪个目录中。 有五个红色（黑色）方块，目录内核/因此容纳我们收集的五个错误。

我们在十个主要的Linux子系统中发现了错误，表明可变性错误并不局限于任何特定的子系统。 这些是从网络（网络/）到设备驱动程序（驱动程序/，块/）到文件系统（fs /）或加密（加密/）的定性不同的Linux子系统。 请注意，Linux子系统通常由不同的人维护和开发，这增加了我们收集的多样性。 我们发现9个目录中没有错误，占总体Linux内核代码的3%以下。 此外，其中三个（工具/脚本/和样本/）包含示例和支持代码（构建基础结构，诊断工具等）不能在编译的内核上运行。 我们现在准备好回答RQ1：

结*论1：可变性错误并不局限于任何特定类型的错误，容易出错的特性或Linux内核中的位置。*

我们发现可变性错误落在20种不同类型的语义错误中，涉及78个质量不同的功能，并且位于Linux内核的10个主要子系统中。

我们现在转向关于研究问题RQ2的证据：

观察4：我们发现了30个涉及非局部定义特征的错误：即在另一个子系统中“远程”定义的功能，而不是发生错误的功能。

了解这些错误涉及不同子系统的功能和特性，而大多数Linux开发人员都致力于单个子系统。 例如，错误6252547b8a7（图。 4）发生在驱动程序/子系统中，但其中一个交互功能IRQ_DOMAIN在kernel /中定义。 错误0dc77b6dabe在extcon-class模块（drivers /）的加载函数中发生，是由于在fs /中不正确地使用sysfs虚拟文件系统API特性SYSFS导致的。 我们向Linux开发人员证实，横切功能构成了错误的频繁来源。

观察5：可变性可以是隐含的，甚至可以隐藏在（可选地不同的）头文件中指定的（可选的）配置相关的宏，函数或类型定义中。

隐藏的可变性使识别变异性相关问题显着复杂化。例如，在bug 0988c4c7fb5中，如果接收到VLAN标记的网络打包，则会调用函数vlan_hwaccel_do_receive。 但是，根据功能VLAN_8021Q是否存在，该功能有两种不同的定义。没有VLAN_8021Q支持的变体通过无条件输入的该函数的模型实现进行编译

[8]http://cloc.sourceforge.net/

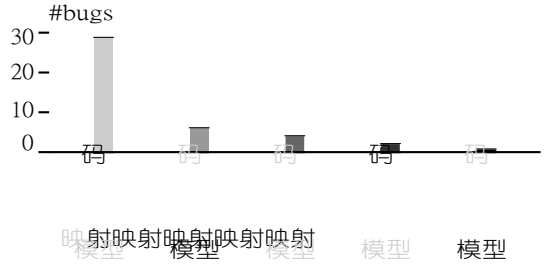


图7：哪个层是修正的错误。

错误状态。 另一个例子是bug 0f8f8094d28，除了数组的长度（KMALLOC_SHIFT_– HIGH + 1）与架构有关，并且只有PowerPC架构，对于给定的虚拟页面，它可以被视为对数组的一个微不足道的访问大小，都受到影响。 vlan_hwaccel_do_receive 和KMALLOC_SHIFT_HIGH都在不同位置有不同的定义。

观察6：变异性错误不仅在代码中得到修复，一些固定在映射中，一些固定在模型中，一些固定在这些组合中。

数字 7 显示我们示例中的错误是否在代码，映射或模型中得到解决。 尽管我们只记录了代码中出现的错误，但我们的示例中的13个错误已在映射，模型或两层中修复。

映射和模型中的简单修复示例分别是提交472a474c663和7c6048b7c83。 前者增加了一个新的#ifndef来防止对APIC_init_uniprocessor的双重调用 – 这不是幂等的，而后者修改了STUB_POULSBO的Kconfig项以防止构建错误。

Bug-fix 6252547b8a7（Fig。 5）删除了一个功能依赖项（TWL4030_CORE不再依赖于IRQ_DOMAIN），并在启用IRQ_DOMAIN（而不是OF_IRQ）时更改映射以初始化结构体字段操作。 映射和代码中的多重修复的一个例子是提交63878acfafb，它删除了某些初始化代码到特征PM（电源管理）的映射，并添加了一个函数存根。

这种对代码，映射和模型的分层可能会掩盖错误的原因，因为对错误的充分分析需要理解这三个层次。 此外，每一层涉及不同的语言； 特别是对于Linux：代码是C，使用cpp和GNU Make来表示映射，并且使用Kconfig指定特征模型。

据推测，这种复杂性可能会导致开发人员在错误的地方修复错误。 例如，我们的bug修复6252547b8a7删除了TWL4030_CORE对IRQ_DOMAIN的依赖关系，这是由commit aeb5032b3f8添加的。 Appar-
当然，aeb5032b3f8将这种依赖性引入了功能，ture模型来防止构建错误，所以修复一个bug，但是这有不良副作用。 根据提交6252547b8a7中提供的消息，构建错误的正确解决方法是在存在特征IRQ_DOMAIN的情况下作出变量声明。

观察7：我们在Linux内核中发现了多达30个功能交互错误。

我们将bug的特征交互程度或bug的程度定义为在其存在条件下发生的各个特征的数量。 直观地说，一个错误的程度

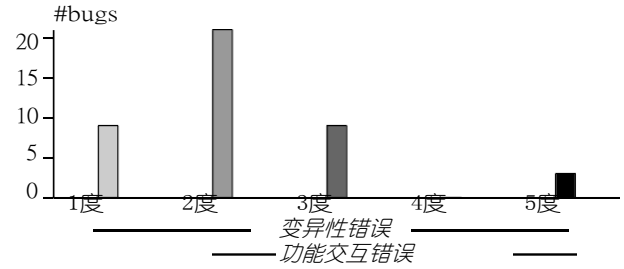


图8：错误中涉及的功能数量（功能 – 交互程度）。

表示必须交互的功能的数量，以便发生错误。 存在于任何有效配置中的错误是一个独立于功能或0度错误的错误。程度大于零的错误是可变性错误，因此出现在有效配置的非空严格子集中。 特别是，如果错误的程度大于1，则错误是由两个或更多功能的交互引起的。 由于功能交互而出现的软件错误被称为功能交互错误。

特征交互错误本质上更复杂，因为要考虑的变体数量是错误程度的指数。 Bug 6252547b8a7（参考图。4(b)）是双特征交互的结果。 包含该bug的代码片段涉及三个不同的功能，并且rep-
不满意四个变体（针对特征模型进行了更正），但只有一个变体出现了缺陷。 操作指针在启用TWL4030_CORE的变体中取消引用，但除非启用了OF_IRQ，否则它将不会正确初始化。 搜索此bug的开发人员需要分别考虑每个变体，或者考虑每个功能对操作指针值的综合影响。正如专业Linux开发人员所证实的，即使在简化的情况下，这些都不是很容易系统地执行，并且在实践中完全不可行。

当变异性影响类型定义时，特征交互可能非常微妙。Commit 51fd36f3fad修复了Linux高分辨率定时器机制中的错误，这是由于数字截断错误造成的，该错误只发生在不支持KTIME_SCALAR功能的32位体系结构中。 在这些特定的配置中，ktime_t是一个带有两个32位字段的结构，而不是一个64位字段，用于存储剩余的纳秒数以执行定时器。
尝试在这些32位字段之一中存储一些较大的64位值时会发生该错误，从而导致存储负值。 有趣的是，我们采访的一位Linux开发人员也提到了由于结构字段对齐方式的变化而导致优化缓存缺失的难度。

观察8：我们确定了12个涉及三个或更多特征的错误。

3度错误的一个例子是ae249b5fa27，由PA-RISC架构（功能PARISC）中DISCONTIGMEM（有效处理不连续物理内存）的交互支持以及通过proc /虚拟文件系统监视内存利用率的能力（功能PROC_- PAGE_MONITOR）。 我们还发现了5度错误，例如commit321ac329e93，由32位PowerPC体系结构引起，当KPROBES不可用时，不会禁用内核内存写保护启用 – 需要在运行时修改内核代码的动态调试功能。

数字 8 总结了我们的错误程度。 就我们所知，这是操作系统领域中首次记录的功能交互错误集合。 到目前为止，大多数功能交互错误已经被识别，记录并发布在电信领域[11].

观察9：可变性错误的存在条件也涉及禁用功能。

表 3 列出并分组我们样本的存在条件的结构。 我们观察了两种主要类型的bug存在条件：启用了某些功能，必须启用一个或多个功能才能使bug发生；和一些启用了一个禁用的功能，在启用零个或多个功能并禁用一个功能时存在该错误。 我们在某些启用配置中确定了20个错误，并且在某些启用了一个禁用的情况下确定了另外20个错误。 （请注意一个
的存在条件具有形式，$(aa') b$，但是，由于它是由ab或$a'b$暗示的，所以我们将它包含在一个启用一个残疾人的类中。）

通常通过测试一个或多个最大配置来测试高度可配置的系统，其中尽可能多的功能被启用 – 在Linux中，这是通过使用预定义的配置allyesconfig来完成的。 这种策略允许通过简单地测试一个单一的最大配置来找到许多具有某些启用存在条件的错误。 但是，如果在实践中出现否定特征的情况与我们的样本一样多，那么只测试最大配置，将会漏掉大量的错误。

根据我们的经验，Linux中功能的实现横切了许多代码位置，功能代码是混合的。 因此，禁用某项功能既可以添加也可以从其他功能中删除代码，并且我们预计否定功能通常会成为错误存在条件的一部分。 错误6252547b8a7（图。 4）就是这样一个例子。 禁用OF_-
因为这个特性，IRQ导致空指针解引用负责初始化操作结构字段。 另一个例子是错误60e233a5660，其中当功能HOTPLUG被禁用时，实现函数add_uevent_var无法保留导致缓冲区溢出的不变量。

观察10：观察到的臭虫存在条件存在有效的测试策略。

给定在Tbl中观察到的模式（某些启用和某些启用一个禁用）。3，我们可以想到更好的测试

表3：存在条件的结构（即，发生42个错误的配置）。

20	*一些启用：*
6	*a*
8	*a* ∧ *b*
5	*a* ∧ *b* ∧ *c*
0	*a* ∧ *b* ∧ *c* ∧ *d*
1	*a* ^ *b* ^ *c* ^ *d* ^ *e*
20	*一些启用一禁用：*
3	¬*a*
13	*a* ∧ ¬*b* 其中之一是：（*a*∨*a*¡）∧ ¬*b*3*a* ∧ *b* ∧¬*c*
0	*a* ∧ *b* ∧ *c* ∧ ¬*d*
1	*a* ^ *b* ^ *c* ^ *d* ^ *ee*
2	**其他配置：**
1	¬*a* ∧ ¬*b*
1	*a*∧∧*b*∧∧*c*∧∧*d*∧*ee*

策略而不是最大化配置测试。 我们提出了一种一次性禁用的配置测试策略，我们测试的配置中只有一个功能被禁用，对应于公式g F：（ f）g。 表 4比较两种策略，最大配置测试和一次禁用配置测试。 我们还添加了一个条目，用于对所有配置进行详尽测试，作为基线（成本在那里指数）。

最大配置测试具有不变的成本 - 理想情况下只有一个配置需要测试，因此可以扩展到具有任意大量特征（F）的程序族。 这似乎是一个相当不错的启发式：我们的样本中有48%的错误，42个中的20个可以通过这种方式找到。 一次性禁用的配置测试在F上具有线性成本，因此它具有合理的可扩展性，即使对于具有数千个功能（如Linux）的程序系列也是如此。 值得注意的是，我们的错误中有95%是42个中的40个，可以通过测试F一个禁用的配置来找到。 请注意，这些配置还会找到具有一定启用状态的错误（除了需要启用所有功能的假设情况外）。

在实践中，我们必须考虑特征模型在测试策略中的效果。 由于功能之间互相独立的依赖关系，通常没有最大配置，但许多本地最大配置。 而且，由于某些功能依赖于其他功能，我们通常不能单独禁用功能。 具有特征模型的实际考虑是枚举要测试的配置只需要选择有效的配置，这本身就是NP完全问题。 然而，鉴于现代SAT求解器（数十万变量和子句）的可扩展性以及现实世界程序系列的规模（仅数以千计的特性），我们期望枚举有效的一次禁用配置将是易于处理的。 现在让我们回答RQ2。 这是一个众所周知的事实，可变数量的变体使开发人员很难了解并验证代码，但是：

结论2：除了引入指数数量的程序变体外，变化还会以多种方式增加错误的复杂性。

我们的分析表明，可变性会影响多个维度上的错误的复杂性。 让我们总结一下：

- 错误的发生是因为功能的实现是混合的，导致不希望的交互，例如通过程序变量；
- 交互发生在不同子系统的功能之间，要求来自Linux开发人员的跨子系统知识；
- 可变性可能是隐含的，甚至隐藏在备用位置指定的备选宏，函数和类型定义中；
- 可变性错误是代码中的错误，映射中，特征模型中的错误或其任何组合的结果；
- 此外，这些层中的每一层都涉及不同的语言（C，cpp，GNU Make和Kconfig）；
- 并非所有这些错误都会通过最大配置来检测，由于与禁用功能交互而导致的配给测试；
- Linux树中编译器错误的存在表明传统的特征不敏感工具不足以发现变异性错误。

表4：最大与一个禁用配置测试。 成本是满足公式的配置数量，忽略特征模型。 我们的示例显示为bug覆盖的好处。

测试公式（s）	成本	效益
	$O(1)$	48% (20/42)
$\bigwedge f \in F$	$\neg g$ 0 (F	95% (40/42)
	)	100% (42/42)

6. 对有效性的威胁

6.1 内部有效性

由于选择过程而产生偏见。 当我们从提交中提取错误时，我们的收集偏向于找到，报告和修复的错误。 由于用户运行可能的Linux配置的一小部分，并且开发人员缺乏对功能敏感的工具，因此可能只找到一部分错误。

此外，我们基于关键字的搜索依赖于Linux开发人员正确识别和报告错误变化的能力。 但是，请注意，在Linux中，变化无处不在，通常“隐藏”。 例如，ath3k蓝牙驱动程序模块文件没有明确的可变性，但是在保持可变性预处理和宏扩展之后，我们可以计算数千个涉及大约400个特征的cpp条件。 那么开发人员总是不会意识到他们修复的错误的可变性。

为了进一步降低引入误报的风险，如果我们未能提取明显的错误跟踪，或者我们无法理解提交作者给出的指针，我们不会记录错误。 这可能会导致对可复制性和较低复杂性错误的偏见。

由于详细的定性分析方法存在内在的偏见，我们无法定量观察Linux内核中整个bug群体的错误频率和属性。 但是，请注意，我们能够进行定性观察，例如某些类型的错误的存在确认（参见Sect。 5）。 由于我们只进行这种观察，因此我们不需要减轻这种威胁（有趣的是，尽管如此，我们的收藏仍然表现出非常广泛的多样性，如第4节所示。 5).

误报和整体正确性。 通过仅考虑已被Linux开发人员识别和修复的变异性错误，我们可以减少引入误报的风险。 我们只从Linux稳定分支进行修复错误，其中的提交已经被其他开发人员审查过，特别是由一位更有经验的Linux维护人员进行了审查。 此外，我们的数据可以独立验证，因为它是公开的。 引入误报的风险不是零，但是，例如，提交b1cc4c55c69为保证不为空的指针添加无效性检查[9]。 我们很容易认为上述情况表明存在可变性错误，而事实上这只是一个错误

一个保守的检查来检测潜在的错误。
手动分析错误以提取错误跟踪
也容易出错，特别是对于像C和a这样的语言
[9] https://lkml.org/lkml/2010/10/15/30

复杂的大型系统如Linux。 理想情况下，我们应该支持使用功能敏感的程序分片进行手动分析（如果存在）。基于错误查找器的更自动化方法不能令人满意。 错误查找器是针对某些类型的错误而建立的，因此它们可以为他们的特定错误类别提供良好的统计范围，但他们无法评估出现的错误的多样性。

我们基于手动切片导出简化的错误，过滤出不相关的语句。 我们还通过函数指针抽象出C语言的特性，如结构和动态调度。 虽然这个过程是系统的，但它是手动执行的，因此容易出错。

6.2 外部有效性

*少*量的错误。 我们样本的大小反映了观测的普遍性。 收集并特别分析这42个臭虫的过程花费了几个人工月，对于大量臭虫的研究是不可行的。 我们预计在不久的将来，我们的数据库也将继续增长，也来自第三方的贡献。

*单*科研究。 我们决定专注于Linux，所以我们的发现不容易推广到其他高度可配置的软件。 然而，Linux的规模和性质使其成为具有可变性的软件的公平的最差代表性。 我们发现的错误类型，尤其是内存错误，可望在C语言中实现的任何可配置系统软件中使用。此外，Linux内核项目本身的重要性证明了其错误的调查是合理的，即使它限制了一般性。

7. 相关工作

*错*误数据库。 ClabureDB是Linux内核的bug报告数据库，其用途类似于我们的[31]，尽管忽略了变异性。 与ClabureDB不同，我们提供的信息记录使非专家能够快速了解错误并对他们的分析进行基准测试。 这包括每个bug的简化C99版本，不相关的细节都会被抽象出来，以及针对内核体验有限的研究人员的解释和参考文献。ClabureDB的主要优势在于它的大小 – 数据库使用现有的bug查找器自动填充。 我们的数据库很小。 我们手动填充它，因为没有合适的bug查找器处理可变性（这也意味着我们的bug在ClabureDB中没有充分涵盖）。

挖掘可变性错误。 纳迪等人。 挖掘Linux存储库以研究变异异常[28]。 异常是映射错误，可以通过检查布尔公式在特性上的可满足性来检测，例如将代码映射到无效配置。 虽然我们以类似的方式进行研究，但我们专注于代码中更广泛的语义错误类别，其中包括数据和控制流程错误。

Apel和合作者使用模型检查器在简单的电子邮件客户端中查找功能交互[3]，*使用一种称为变异性编码（配置提升*[30]）。 功能被编码为布尔变量，条件编译指令被转换成条件语句。 我们专注于广泛了解变异性错误的性质。这不能通过模型检查器来搜索特定类型的交互。 了解可变性错误应导致构建可扩展的错误查找器，从而实现诸如[3]在未来将用于Linux。

Medeiros等人。 已经研究了句法变异性错误[26]。 他们使用了可变性感知的C语法分析器[23]来自动查找错误并彻底查找所有语法错误。 他们发现在41个家庭中只有几十个错误，表明在承诺的代码中句法变异性错误很少见。 我们关注更广泛的更复杂的语义错误。 纳迪等人。 在基于预处理器的程序系列中支持地雷特性依赖，以支持现有代码库的可变性模型综合[27]。 他们从预处理器指令的嵌套和parse-，type-和link-errors中推断依赖关系，假设无法构建的配置无效。 我们再次考虑更广泛的一类错误比迄今可以自动检测到的要多。

*与*方法有关的工作。 田等人。 研究了在Linux存储库中区分错误修复提交的问题[34]。 他们使用半监督学习来根据提交日志中的令牌和从补丁内容中提取的代码度量来对提交进行分类。 与之前的基于关键词的方法相比，它们显着改善了召回率（不降低精度）。 在我们的研究中，大部分时间都用于分析提交，而不是找到潜在的候选人，所以我们发现了一个简单的基于关键字的方法就足够了。 尹等人。 收集由开源和商业软件中的错误配置导致的数百个错误[36]构建一套具有代表性的大规模软件系统错误。 他们考虑从配置文件中读取参数的系统，而不是静态配置的系统。 更重要的是，他们记录用户的错误透视，而不是（我们）程序员的视角。

Padioleau等人 研究了Linux内核的间接进化，遵循一种接近我们的方法[29]。 当现有代码适应内核接口的变化时，就会发生抵押演化。 他们通过分析修补程序修复程序来识别潜在的侧支进化候选者，然后手动选择72进行更仔细的分析。 同样，他们分类并对其数据进行深入分析。

8. 结论

我们在Linux内核存储库中发现了42个可变性错误，其中包括30个功能交互错误。 我们分析了它们的属性，并将这些错误中的每一个都压缩成了一个具有相同变异属性的自包含C99程序。 这些简化的错误有助于理解真正的错误并构成分析工具的公开可用基准。 我们观察到，可变性错误并不局限于任何特定类型的错误，容易出错的特性或Linux内核的源代码位置（文件或子系统）。 此外，可变性会以多种方式增加Linux中错误的复杂性，除了众所周知的指数级许多代码变体的引入之外：a）特征的实现是混合的并且不期望的交互可以容易地发生，b）这些交互可以发生在来自不同子系统；和c）代码中可能出现错误，在映射中，在特征模型中，或其任何组合。

致谢。

我们感谢内核开发人员Jesper Brouer和MatiasBjørling。Julia Lawall和Norber Siegmund提供了有用的建议。 这项工作得到了丹麦独立研究委员会在一个Sapere Aude项目VARIETE下的支持。

9. 参考

[1] S. Apel, D. Batory, C.Ka¨stner和G. Saake。 面向特征的软件产品线。 施普林格出版社，2013年。

[2] S. Apel, C.K¨stner, A.Gr¨sslinger和C. Lengauer。 面向功能的产品线的类型安全性。 自动化软件工程，2010年第17期。

[3] S. Apel, H. Speidel, P. Wendler, A. von Rhein和 D.拜尔。 使用功能感知验证检测功能交互。 在Proceedings of the 26th IEEE / ACM International Conference on Automated Software Engineering (ASE'11)，Lawrence, USA, 2011.IEEE Computer Society。

[4] T. Berger, R. Rublack, D. Nair, JM Atlee, M. Becker, K.Czarnecki和A. Wasowski。 工业实践中变异性建模的一项调查。 在S. Gnesi, P. Collet和K. Schmid编辑，VaMoS。 ACM，2013。

[5] T. Berger, S. She, R. Lotufo, A. Wasowski, and K.Czarnecki谈到。 在系统软件领域研究变异模型和语言。 IEEE Trans。 软件工程，39 (12)。

[6] E. Bodden, T. Toledo, M. Ribeiro, C. Brabrand, P. Borba和M. Mezini。 SPL电梯 - 在几分钟内而不是几年内静态分析软件产品线。 2013年PLDI'13。

[7] E. Bounimova, P. Godefroid和D. Molnar。 数十亿和数十亿的限制：生产中的Whitebox模糊测试。 在2013年国际软件工程会议论文集中，ICSE '13，Piscataway, NJ, USA, 2013.IEEE Press。

[8] D. Bovet和M. Cesati。 了解Linux内核。 O'Reilly媒体，2005年。

[9] C. Brabrand, M. Ribeiro, T. Toledo, J. Winther, and P. Borba。 软件产品线的软件内数据流分析。 交易 *面向方面的软件开发，10,2013。*

[10] WR Bush, JD Pincus和DJ Sielaff。 用于查找动态编程错误的静态分析器。 软件选装。 PRACT。 Exper。，30 (7)，2000年6月。

[11] M. Calder, M. Kolberg, EH Magill, 和 S. Reiff-Marganiec。 特征交互：重要的审查和考虑预测。 COMPUT。 网络，41 (1)，2003。

[12] A. Classen, P. Heymans, P.-Y. Schobbens和 A. Legay。 软件产品线的符号模型检查。 在ICSE，2011年。

[13] A. Classen, P. Heymans, P.-Y. Schobbens, A. Legay和J.-F. 拉斯金。 模型检查大量系统：有效验证软件产品线中的时间属性。 在ICSE'10，南非开普敦，2010年。ACM。

[14] K. Czarnecki和K. Pietroszek。 验证基于特征的模型模板反对良构OCL约束。 在第五届关于生成规划和组件工程的国际会议论文集中，GPCE'06，纽约，纽约，美国，2006年。ACM。

[15] N. Dor, M. Rodeh和M. Sagiv。 CSSV：在静态检测所有缓冲区溢出的实际工具中，SIGPLAN 不，38 (5)，2003。

[16] D.埃文斯。 静态检测动态内存错误。 *SIGPLAN Not。，31 (5)，1996。*

[17] A. Gruler, M. Leucker和KD Scheidemann。 建模和模型检查软件产品线。 FMOODS，2008。

[18] G. Holl, M. Vierhauser, W. Heider, P.Grünbacher, and R. Rabiser。 用于多产品线中工具支持的产品线捆绑包。 在VaMoS，2011年。

[19] D. Hovemeyer和W. Pugh。 寻找更多的空指针错误，但不是太多。 在Proceedings of the第7届ACM SIGPLAN-SIGSOFT软件工具和工程程序分析研讨会上，PASTE '07，纽约，纽约，美国，2007年。ACM。

[20] K. Kang, S. Cohen, J. Hess, W. Nowak, and S.彼得森。 面向特征的领域分析 (FODA) 可行性研究。 技术。 Rep。CMU / SEI-90-TR-21，CMU-SEI，1990。

[21] 凯*斯特纳 虚拟分离问题：迈向预处理器2.0。 博士论文，德国马尔堡，2010。*

[22] C.K¨stner和S. Apel。 类型检查软件产品线 - 一种正式的方法。 在第23届IEEE / ACM国际自动软件工程会议 (ASE'08)，意大利拉奎拉，意大利，2008年的会议记录。

[23] A. Kenner, C.K¨stner, S. Haase和T. Leich。 Typechef：向类型检查#ifdef变化c。 在第2届国际功能面向软件开发研讨会论文集中，FOSD '10，纽约，纽约，美国，2010年。ACM。

[24] CHP Kim, E. Bodden, D. Batory和S. Khurshid。 减少配置以在软件产品线中进行监控。 第一届国际运行验证会议 (RV)，马耳他LNCS第6418期，2010年。Springer。

[25] R.爱。 Linux内核开发。 开发人员库。 皮尔逊教育，2010年。

[26] F. Medeiros, M. Ribeiro和R. Gheyi。 研究基于预处理器的语法错误。 在第12届生成规划国际会议论文集：概念&＃38；经验，GPCE '13，纽约，纽约州，美国，2013年。ACM。

[27] S. Nadi, T. Berger, C. K.stner和K. Czarnecki。 挖掘配置约束：静态分析和实证结果。 2014年第36届国际软件工程会议 (ICSE'14)。

[28] S. Nadi, C. Dietrich, R. Tartler, RC Holt和D.罗曼。 Linux变异异常：导致它们的原因以及它们如何得到修复？ 在T.齐默尔曼，MD Penta和S. Kim，编辑，MSR。 IEEE / ACM，2013。

[29] Y. Padioleau, JL Lawall和G. Muller。 了解Linux设备驱动程序中的附带进展。 在第一届ACM SIGOPS / EuroSys欧洲计算机系统会议论文集2006，EuroSys '06，纽约，纽约，美国，2006年。ACM。

[30] H. Post和C. Sinz。 配置提升：验证符合软件配置。 在Proceedings of the 23th IEEE / ACM International Conference on Automated Software Engineering (ASE'08)，L'Aquila，意大利，2008年IEEE计算机学会。

[31] J. Slaby, J. Strejcek和M.Trt'ık。 ClabureDB：分类错误报告数据库。 在R. Giacobazzi，

J.Berdine和I.Mastroeni编辑的“验证，模型检查和抽象解释”，计算机科学讲义第7737卷。 斯普林格柏林海德堡，2013年。

[32] 电气和电子工程师协会。 IEEE软件工程术语标准术语表。 IEEE标准，1990。

[33] T. Thum，S. Apel，C.K¨stner，I. Schaefer，and G. Saake。 软件产品线分析和策略分析。 ACM Computing Surveys，2014。

[34] Y. Tian，J. Lawall和D. Lo。 识别Linux错误修复补丁。 2012年国际软件工程会议论文集，ICSE 2012，Piscataway，NJ，USA，2012. IEEE Press。

[35] D. Wagner，JS Foster，EA Brewer和A. Aiken。自动检测缓冲区溢出漏洞的第一步。 在NDSS中。互联网协会，2000年。

[36] Z. Yin，X. Ma，J. Zheng，Y. Zhou，LN Bairavasundaram和S. Pasupathy。 商业和开源系统中配置错误的实证研究。 在Proc。 第二十三届ACM操作系统原理研讨会，SOSP ’11，纽约，纽约，美国，2011年。ACM。