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）。

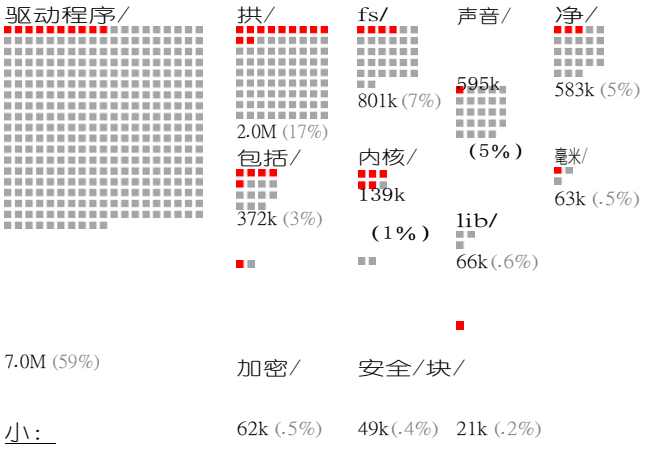
我们首先介绍支持我们的第一个研究问题RQ1的观察结果：

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/

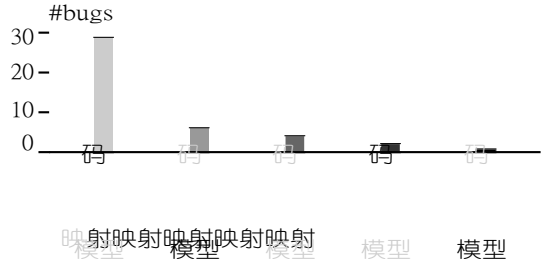


图7：哪个层是修正的错误。

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

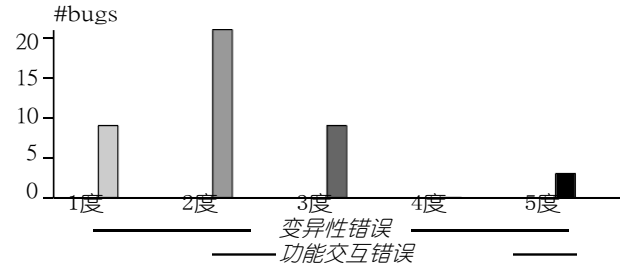


图8：错误中涉及的功能数量（功能 – 交互程度）。

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。