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.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
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.

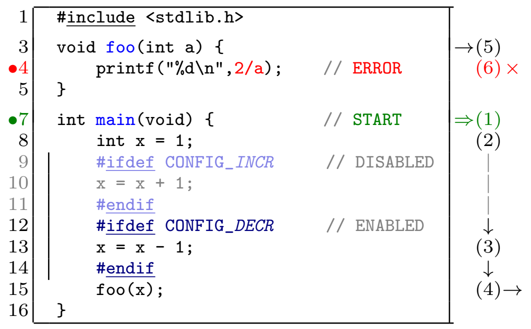
A *feature* is a unit of functionality additional to the core software [11]. The core (*base variant*) implements the basic functionality present in any variant of a program family. The different selections of features (*configurations*) define the set of program variants. Often, two features cannot 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.

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 ($\psi_{\mathrm{FM}} = \mathtt{true}$), 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 $\kappa \vDash \varphi$. As an example, consider the decrement statement in line 13, which has presence condition *DECR*, thus it is part of configurations $\kappa_0 = \neg \mathit{INCR} \land \mathit{DECR}$ and $\kappa_1 = \mathit{INCR} \land \mathit{DECR}$.

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 $\neg \mathit{INCR} \land \mathit{DECR}$.



```
 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.

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 $\psi_{\mathrm{FM}} = \mathit{DECR} \rightarrow \mathit{INCR}$ 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, we analyze and explain them. Finally, we reflect on the aggregated material to answer our research questions.

[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**

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.

```
   #include <stdlib.h>

   #ifdef CONFIG_TWL4030_CORE               // ENABLED
   #define CONFIG_IRQ_DOMAIN
   #endif

   #ifdef CONFIG_IRQ_DOMAIN                 // ENABLED
   int irq_domain_simple_ops = 1;

   void irq_domain_add(int *ops) {                        →(6)
 •     int irq = *ops;                      // ERROR       (7)×
   }
   #endif

   #ifdef CONFIG_TWL4030_CORE               // ENABLED
   void twl_probe() {                                     →(3)
       int *ops = NULL;                                    (4)
       #ifdef CONFIG_OF_IRQ                 // DISABLED     |
       ops = &irq_domain_simple_ops;                        |
       #endif                                               |
       irq_domain_add(ops);                                (5)→
   }
   #endif

 • int main(void) {                                       ⇒(1)
       #ifdef CONFIG_TWL4030_CORE           // ENABLED      ↓
       twl_probe();                                        (2)→
       #endif
   }
```

(b) Simplified version.

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

```diff
@@ -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* $\land \neg$ *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

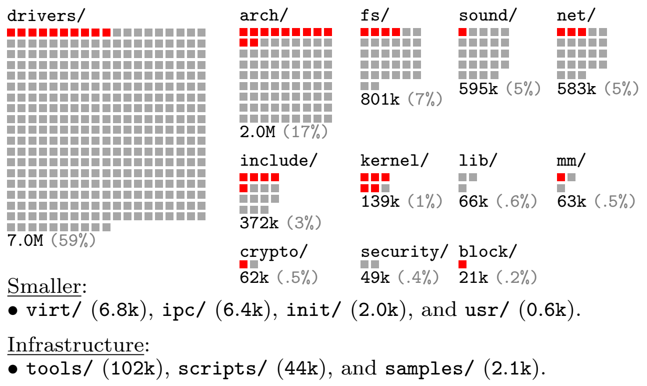


Figure 6: Location of the 42 bugs in the main Linux directories as of March 2014. Each square represents 25 thousand lines of code. The precise number of LOC and its percentage of the total is given below the squares. A red (dark) square symbolizes the occurrence of one of the bugs.

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 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.

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

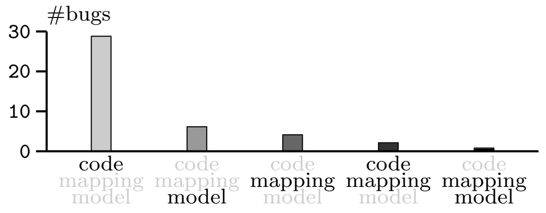


Figure 7: In which layer(s) are the bugs fixed.

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

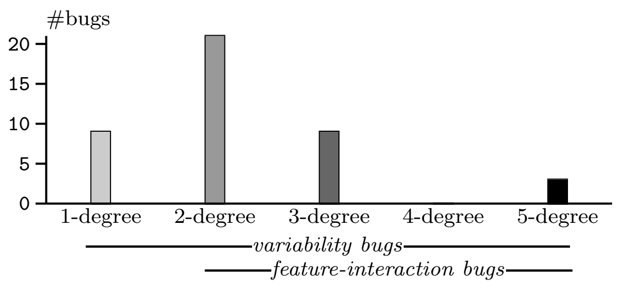


Figure 8: Numbers of features involved in a bug (*feature-interaction degree*).

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 $\forall g \in \mathbb{F}\colon (\bigwedge_{f\in\mathbb{F}\setminus\{g\}} f) \wedge \neg g$. 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)
$\forall g \in \mathbb{F}\colon (\bigwedge_{f\in\mathbb{F}\setminus\{g\}} f) \wedge \neg g$	$O(\lvert\mathbb{F}\rvert)$	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 &#38; 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.