Variability Bugs in Highly-Configurable Systems: A Qualitative Analysis

IAGO ABAL, IT University of Copenhagen, Denmark
JEAN MELO, IT University of Copenhagen, Denmark
ȘTEFAN STĂNCIULESCU, IT University of Copenhagen, Denmark
CLAUS BRABRAND, IT University of Copenhagen, Denmark
MÁRCIO RIBEIRO, Federal University of Alagoas, Brazil
ANDRZEJ WĄSOWSKI, IT University of Copenhagen, Denmark

Variability-sensitive verification pursues effective analysis of the exponentially many variants of a program family. Several variability-aware techniques have been proposed, but researchers still lack examples of concrete bugs induced by variability, occurring in real large-scale systems. A collection of real world bugs is needed to evaluate tool implementations of variability-sensitive analyses by testing them on real bugs. We present a qualitative study of 98 diverse variability bugs (i.e., bugs that occur in some variants and not in others) collected from bug-fixing commits in the Linux, Apache, BusyBox, and Marlin repositories. We analyze each of the bugs, and record the results in a database. For each bug, we create a self-contained simplified version and a simplified patch, in order to help researchers who are not experts on these subject studies to understand them, so that they can use these bugs for evaluation of their tools. In addition, we provide single-function versions of the bugs, which are useful for evaluating intra-procedural analyses. A web-based user interface for the database allows to conveniently browse and visualize the collection of bugs. Our study provides insights into the nature and occurrence of variability bugs in four highly-configurable systems implemented in C/C++, and shows in what ways variability hinders comprehension and the uncovering of software bugs.

CCS Concepts: • **Software and its engineering → Software testing and debugging**; *Preprocessors*; • **Theory of computation →** *Program verification*; *Program analysis*;

Additional Key Words and Phrases: Bugs, Feature Interactions, Linux, Software Variability

ACM Reference Format:
Iago Abal, Jean Melo, Ștefan Stănciulescu, Claus Brabrand, Márcio Ribeiro, and Andrzej Wąsowski. 2017. Variability Bugs in Highly-Configurable Systems: A Qualitative Analysis. *ACM Trans. Softw. Eng. Methodol.* X, X, Article X (October 2017), 34 pages. https://doi.org/0000001.0000001_2

1 INTRODUCTION

Many software projects adopt variability to tailor development of individual software products to particular market niches [3]. Other software projects, such as the Linux kernel, embrace variability and use configuration options known as *features* [30] to tailor functional and non-functional properties to the needs of a particular user. Such systems are often referred to as *highly-configurable*

Authors' addresses: Iago Abal, IT University of Copenhagen, Denmark, iago@itu.dk; Jean Melo, IT University of Copenhagen, Denmark, jeanmelo@itu.dk; Ștefan Stănciulescu, IT University of Copenhagen, Denmark, scas@itu.dk; Claus Brabrand, IT University of Copenhagen, Denmark, brabrand@itu.dk; Márcio Ribeiro, Federal University of Alagoas, Brazil, marcio@ic.ufal.br; Andrzej Wąsowski, IT University of Copenhagen, Copenhagen, Denmark, wasowski@itu.dk.

1049-331X/2017/10-ARTX $15.00
https://doi.org/0000001.0000001_2

systems and can get very large and encompass large sets of features. There exist reports of industrial systems with thousands of features [7], and extensive open-source examples are documented in detail [8].

Features in a configurable system interact in non-trivial ways, in order to influence the functionality of each other. Interestingly, bugs in configurable systems do not always occur unconditionally, in all configurations. Bugs involving one or more feature that have to be either enabled or disabled in order for the bug to occur are known as *variability bugs*. Importantly, variability bugs therefore occur only in certain configurations and not in others. Some variability bugs involve multiple (two or more) features each of which have to be enabled, respectively, disabled in order for the bug to occur; such bugs are known as *feature-interaction bugs*. A bug in an individual configuration may be found by analyzers based on standard program analysis techniques. However, since the number of possible configurations is exponential in the number of features, it is not feasible to analyze each configuration separately.

Family-based analyses [56] 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 [4, 9, 12, 19, 32, 35] and model checking [5, 16, 17, 26, 36, 49] 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 variability-sensitive analyses in actual problems.

The understanding of the complexity of variability bugs is not common among practitioners and in available artifacts. While bug reports abound, there is little knowledge on how those bugs are caused by feature interactions. Very often, due to the complexities of a large project like Linux, and the lack of variability-aware 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 four highly configurable systems: Linux, Apache, BusyBox, and Marlin. We address this objective via a qualitative in-depth analysis and documentation of 98 cases of such bugs. We make the following contributions:

- *Identification of 98 variability bugs in four highly configurable systems: Linux, Apache, BusyBox, and Marlin*; including in-depth analysis and presentation for non-experts.
- *A database with 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 database is available at:

  http://VBDb.itu.dk/

- *Self-contained simplified C99*[1] *versions of all bugs, including single-function versions.* These ease comprehension of the underlying causes, and can be used for testing bug-finders in a smaller scale. The single-function versions can be used to test intraprocedural analyses.
- *Simplified patch versions of the bugs.* These patches also help to understand the bugs and present ways of fixing them in accordance with the bug-fixing commits.

[1] C99 is an informal name for ISO/IEC 9899:1999, a version of the C programming language standard.

- *An aggregated reflection over the collection of bugs.* Providing insight on the nature of bugs induced by feature interactions in four highly configurable systems.

We adopt a qualitative manual methodology of analysis for the following reasons. Most importantly, searching for bugs with tools only finds cases that these tools cover, while we are interested in exploring the nature of variability bugs widely. Tools are generally approximating and biased due to undecidability of essentially all interesting questions about programs. An automated bug hunt would be heavily biased against a few kinds of bugs for which the tools were designed, and for the cases of these bugs that they are able to handle. Also, manual sampling from historical bugs avoided false-positives that would pollute the data, had we used automatic bug finding tools. Additionally, family-based bug-finders are rare, experimental (only effective type checkers exist), and not fast enough to extensively scan the long history of Linux and similar systems. Sampling [39] is a good alternative, however uniform sampling of valid configurations for large systems (like the Linux kernel) is known to be difficult. This would require investment in new research of applying and evaluating PSAT-based solutions for the purpose, which, while a fascinating research problem, was judged to be out of scope for this work. Obviously, the manual sampling has not been uniform either but helped to direct the work towards qualitative insights. It helped to increase the diversity of the bugs covered, and in the process inspired us to generate much more information about the bugs (simplified bugs, simplified patches, etc).

Reflecting on the collected material, we learn that complexity of variability bugs comprises the following aspects: variability bugs involve many aspects of programming language semantics, they are distributed in most parts of the code bases, involve multiple features and span code in remote locations. Detecting these bugs is difficult for both people and tools. Once variability-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 all the knowledge condensed in our collection of variability 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 preliminary benchmark for evaluating their tools, and (iii) it will dramatically speed up design of new techniques, since they can be tried on simplified project-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 4–6 describe the analysis: first the considered dimensions, then the aggregate observations. We finish surveying threats to validity (Sect. 7), related work (Sect. 8) and a conclusion (Sect. 9).

2 BACKGROUND

We understand the term *software bug* broadly, as it is defined by IEEE Standard Glossary of Software Engineering [55]. This includes any run-time crash, compiler warning or error, and software weakness in the *Common Weakness Enumeration* (CWE) taxonomy. Often, these bugs manifested as a kernel panic (crash), or were spotted by the compiler (in the form of a warning) when building a specific kernel configuration. While some compiler warnings may appear harmless (for instance an unused variable), they could be side-effects of serious misconceptions that may lead to more serious problems.

A *feature* is a unit of functionality additional to the core software [15]. 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

```
1  int printf(const char * format, ...);
2
3  void foo(int a) {                          →(5)
•4     printf("%d\n",2/a);     // ERROR        (6)×
5  }
6
•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 }
```

Fig. 1. Example of a program family with a variability bug. A division-by-zero error occurs in line 4 whenever INCR is *disabled* and DECR is *enabled*. The right column traces the statements involved from (1) to (6).

specified using a *feature model* [30] (or a decision model [27]), denoted here by ψ_{FM}; effectively a constraint over features defining legal configurations.

Preprocessor-based program families [31] 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 are prefixed by CONFIG_. We follow the Linux convention throughout this paper. Figure 1 presents a tiny preprocessor-based C program family using two features, INCR and DECR. Statements at lines 10 and 13 are conditionally present. Assuming an unrestricted feature model ($\psi_{\text{FM}} = \texttt{true}$), the figure defines a family of four different variants.

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

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 explicitly through the program variable x which they both manipulate (read and write). Enabling either INCR or DECR, or both, results in different values of x prior to calling foo. In general, the presence condition of a bug will implicitly tell us *that* features interact, but it does not necessarily explicitly tell us *how* they interact.

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 $\neg\texttt{INCR} \wedge \texttt{DECR}$.

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

Fix in the Code. If function `foo` ought to accept any `int` value, then the bug could be 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 the Mapping. If we assume that function `foo` should never be called with a zero argument, a possible fix would be to decrement `x` only whenever both `DECR` and `INCR` are enabled:

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

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

3 STUDY DESIGN

Our objective is to qualitatively understand the complexity and nature of *variability bugs* (including *feature-interaction bugs*) in open-source highly-configurable software systems. This includes addressing the following research questions:

Research questions:
RQ1: Are variability bugs limited to specific type of bugs, features, or locations in the code base?
RQ2: In what ways does variability affect bugs?

This paper relies on the initial findings of our *exploratory case study* on variability bugs in Linux published previously [1]. That study followed an exploratory qualitative method, identifying what is possible to learn about diversity of variability bugs using the case study method. It produced a

Table 1. The four subject systems with size metrics (as of December 2015).

System	Domain	LOC	#Features	#Commits
Marlin	3D-printer firmware	43 k	821	2,783
BusyBox	UNIX utilities	176 k	551	13,878
Apache	Web Server	195 k	681	27,677
Linux	Operating system	14 M	16,490	521,276

method design and a list of nine[2] initial observations based on the analysis of 42 variability bugs. These observations explained the answers to the research questions RQ1 and RQ2 for the Linux kernel project, but they were hypotheses in as far as other systems are considered. Methodologically, this paper is a *confirmatory study* where we extend the previous *exploratory study* with three new systems and (in)validate the previous hypotheses. In the end, we confirm all previous observations from the original Linux-only study. This attests to the stability and generalizability of our observations (more on this later).

We extend the prior work by executing three independent confirmatory case studies, replicating the same data collection process and analysis for three new subjects that significantly differ from Linux. The case studies are executed by three new researchers on the project, and the original researcher responsible for the case study only supervises adherence to the method. This leads to extending the data sample with 55 new bug analyses, all available in our bug database. After collecting the data we check, whether the original observations formulated for Linux still hold. It turns out that we are able to confirm all the observations, thus observations in discussion of RQ1 (Sect. 5) and RQ2 (Sect. 6) are labeled as Confirmed.

3.1 Subjects

We study four open-source highly-configurable systems: Linux, Apache, BusyBox, and Marlin. Linux and BusyBox use KConfig to model their configuration space, while the other two do not have an established way of expressing their variability in a well-specified format. Crucially, all have bug data including commits, developer comments and bug trackers publicly available. They are qualitatively different highly configurable systems: one small (Marlin), two medium (Apache and BusyBox), and one large (Linux). They are also different in terms of purpose, variability, and complexity. Besides that, all have different architectures and developers, which allows us to draw slightly broader conclusions.

Linux is likely the largest highly-configurable open-source system in existence, with more than 14 million lines of code and 16 thousand features. We have free access to the bug tracker,[3] the source code and change history, and to public discussions on the mailing list [4] (LKML) and other forums. There also exist books on Linux development [11, 38]—valuable resources when understanding a bug-fix. Access to domain-specific knowledge is crucial for the qualitative analysis.

Likewise, BusyBox is an open-source highly-configurable system that provides several essential Unix tools (such as `ls`, `cp`, and `mkdir`) in a single executable file. BusyBox has more than 500 features and 176 KLOC. Compared to Linux, BusyBox is a much smaller system: about 80 times less LOC and 29 times less features.

Apache has been developed for over 20 years and is one of the most used and popular web servers. It is written in C and C++, consisting of almost 200 KLOC and 700 features.

Marlin is a firmware for 3D printers that is highly configurable, with 43 KLOC and around 800 features. The project is written in C++, is hosted on GitHub, and uses GitHub's issue tracker. Compared to the other three systems, Marlin is a much newer (started in August 2011) and smaller project (only 43 KLOC) mainly due to its focused domain.

Table 1 characterizes the subject systems by aggregating the information about domain, lines of code, number of features, and size of the commit history that we analyze. We examine the entire history (not a specific version) of each project up to December 2015, since our focus is on individual bugs in whatever version of the project they happened to reside. Throughout this paper,

[2] We merge observations 7 and 8 from [1] due to overlap.
[3] https://bugzilla.kernel.org/
[4] https://lkml.org/

we count the lines of code (in any language) with CLOC [5] version 1.53, with the default options. For Linux and BusyBox, we approximate the number of features as the number of unique *(menu)config* entries declared in KCONFIG files. (This is computed by `find . -name` *ConfigFiles* `-exec egrep '^(menu)?config '{} \; | cut -d' ' -f2 | uniq | wc -l`, where *ConfigFiles* is replaced with `'Kconfig*'` and `'Config.*'`, for Linux and BusyBox, respectively.) As Apache and Marlin do not have an explicit feature model nor use KCONFIG, we use grep to extract all `'#if or #ifdef or #elif'` directives, and parse the expressions from which we count the unique identifiers. In this process we eliminate identifiers that have a suffix of the following form: `'_H or H__ or H_'`, as these represent include guards and we do not treat them as features. The size of the commit history is measured as the number of non-merge commits in the repository, which corresponds to the output of `git rev-list HEAD --no-merges --count`. These statistics are approximate, but serve the purpose of characterizing our four subjects.

3.2 Method

For each of the four cases, we follow a three-part method developed during the Linux study: first, we identify the variability bugs in the history of our subject systems. Second, we analyze and explain them. Finally, we reflect on the aggregated material to answer our research questions (formulating hypothetical observations or confirming them respectively).

To do so, we take the Linux [6], Apache [7], BusyBox [8], and Marlin [9] repositories as the units of analysis. In all cases, we analyze the *master* branch of the repository. We focus on bugs already corrected in commits to the repositories. These bugs have been publicly discussed (usually on the project's mailing list or issue tracker) and confirmed as actual bugs by the developers, so the information about the nature of the bug fix is reliable, and we minimize the chance of including fictitious problems.

3.3 Part 1: Finding Variability Bugs

The large commit history of the projects rules out manual investigation of each commit. We have settled on a semi-automated search through the project's commits and issue tracking system (mostly for Marlin) to find variability bugs via historic bug fixes.

We have thus *searched* through the commits for variability bugs using the following steps:

(1) *Selecting variability-related commits.* We retain commits whose *message* indicates a variability-related change; or whose *patch* appears to alter the feature model (in the case of Linux and BusyBox, as Apache and Marlin do not use KCONFIG), the feature mapping, or configuration-dependent code. This is achieved by matching regular expressions of Fig. 2. (We always perform case-insensitive matching of regular expressions.) 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 (KCONFIG) feature model, the (CPP) feature mapping, or code near an `#if` conditional. In our search we exclude *merges* as such commits do not carry changes. The selection of keywords was based on our understanding of the systems, and manual analysis of code and commit messages to identify what kind of keywords are used. Linux and BusyBox follow the pattern of using CONFIG_*fid* to define feature names and refer to them in the commit message, while Apache uses HAVE_*fid* and HAS_*fid*. Marlin, however, does not employ either of the two patterns. At the time of our

[5] http://cloc.sourceforge.net/
[6] http://git.kernel.org/
[7] http://git.apache.org/httpd.git
[8] http://git.busybox.net/busybox/
[9] http://github.com/MarlinFirmware/MarlinDev

Regular expressions:
`configuration`
`config option`
`(HAS|HAVE)_`fid
`if` fid `is (not)`$^?$ `set`
`when` fid `is (not)`$^?$ `set`
`if` fid `is (en|dis)abled`
`when` fid `is (en|dis)abled`
`(CONFIG|ENABLE|FEATURE)_`fid

(a) Message filters.

Regular expressions:
`#if`
`#else`
`#elif`
`#endif`
`select` fid
`config` fid
`depends on` fid

(b) Content filters.

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

study analysis, Marlin used simple feature names without having a well-in-place method for defining them. We used previous knowledge of the system [51] and grep to identify unique feature identifiers, and used those in combination with few regular expressions as explained next to detect variability-related commits.

(2) *Selecting bug-fixing commits.* We further narrow to commits that potentially fix bugs and thus, together with the previous filter, we obtain candidates to variability bug-fixes. This is achieved by matching regular expressions of Fig. 3 against the commit message. Expressions in Fig. 3(a) are generic keywords that can appear in any bug-fixing commit's message or in any issue report. At the *Linux Kernel Summit 2013* conference, the convention to add a "`Fixes:`" footer to the commit message to identify bug-fixing commits was established.[10] For instance, the regular expression `fix` (case insensitive) will match commits adhering to this new convention, at least in the case of Linux. We used an iterative process for finding regular expressions that can match a diverse sample, based on our understanding and examining the systems. For example, we searched for commit messages that used `memory leak` as keyword and identified potential bug-related keywords. Several Linux commits use the keyword `oops` to indicate a possible kernel crash. Expressions in Fig. 3(b) try to identify bug-fixing commits for specific types of bugs, such as references to void-pointer dereferences (`void *`), undefined symbols (`undefined`), uninitialized variables (`uninitialized`), and a variety of memory errors (`overflow`, `memory leak`, etc.). Different combinations of keywords select different number of commits: generic keywords may select still thousands of commits in Linux, while specific keywords may select only a few hundreds or tens.

(3) *Manual scrutiny.* Finally, we read the commit message or the issue, and inspect the changes introduced by the commit to remove false positives. For instance, commit `7518b5890d` matches our regular expression from Fig. 2(a), as the commit message refers to `CONFIG_OF_DYNAMIC`, yet once we examined the complete commit message we understood that it does not fix a bug, but adds new functionality. Especially in the case of Marlin, where often commit messages simply refer to an issue number —e.g. *"Fix #150"*, the issue tracking system contains valuable information for triaging. We down prioritize *very complex* commits as these are more difficult to understand and to extract and examine error traces. A very complex commit either introduces more than a few changes (we choose a cut-off value of *ten*), or affects very complex subsystems (an example from Linux is the `kernel/sched` subsystem). The ideal

[10] http://lkml.iu.edu/hypermail/linux/kernel/1310.3/01046.html

Regular expressions:
bug
fix
oops
warn
error
unsafe
invalid
closes \#
violation
end trace
kernel panic

(a) Generic bug filters.

Regular expressions:
unused
void *
overflow
undefined
double lock
memory leak
uninitialized
dangling pointer
null dereference
null pointer dereference
...

(b) Specific bug filters.

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

commit has an elaborated *message* providing some form of error trace, and introduces few modifications.

3.4 Part 2: Analysis of Bug Candidates

This part of the methodology requires considerable effort in the sense that, 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. Aside from variability, the bugs involve undisciplined `#ifdef` annotations, intraprocedural dataflow, function pointers, and pointer aliasing. When more context is required, we find and follow the associated discussion on the repository's mailing list or issue tracker. Code inspection is supported by CTAGS [11] and the Unix GREP utility, since we lack feature-sensitive tool support. This step requires some knowledge of the system's internals in order to successfully understand the bug. Note that we do not focus on a specific way of finding variability bug candidates, by using only commit message or only the issue tracking system. We use both available sources, especially for smaller subjects, where the amount of the potential bug candidates is smaller. In this step we are interested in understanding the bug and not on analyzing the quality of the commit or of the bug report. The commits helped us finding bugs and understanding the bug's nature.

(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 project, 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.* We condense our understanding in a *simplified version of the bug.* This serves to explain the original bug, and constitutes an easily accessible benchmark for testing and evaluating tools. In addition, we generate *single-function versions* from the *simplified*

[11] http://ctags.sourceforge.net/

versions of the bugs, intended to help researchers test intraprocedural analyses for the same problem.

(4) *Simplified patch.* Last but not least, we also provide a *simplified patch of the bug*. Seeing how the bug has been fixed, will help researchers to comprehend the problem.

We analyzed bugs from the previous step (cf. Sect. 3.3) following this method. We stored the reports from our analyses in a publicly available database. The detailed content of the report is explained in Sect. 4.

3.5 Part 3: Data Analysis and Verification

We reflect on the set of collected data 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 our subject systems (such conclusions would be unsound given the above research method). The analysis purely characterizes diversity of the data set obtained. It allows us to present the entire collection of bugs in an aggregated fashion (e.g., see Sect. 5). We see this qualitative analysis as an important stepping stone towards a representative analysis about the bugs: any such analysis requires building tools. The qualitative analysis indicates which tools should be build.

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 Linux driver, which was traced back to errors both in the feature model and the mapping.

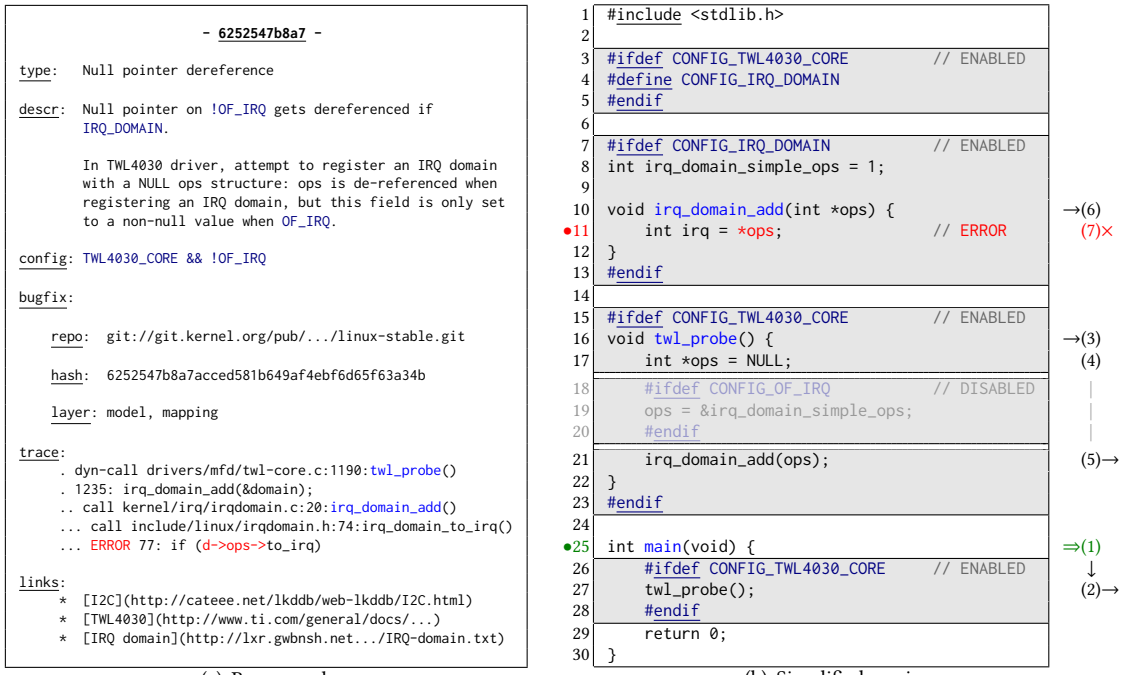
Type of Bug (`type`*).* In order to understand the diversity of variability bugs we establish the type of bugs according to the *Common Weakness Enumeration* (CWE)[12]—a catalog of numbered software weaknesses and vulnerabilities. We follow CWE since it had already been applied to the Linux kernel [50]. 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, among others. The types of bugs that we found are listed in Fig. 8; our additions lack an identifier in the CWE column.

Note that we categorize each bug mostly by its *effect* as opposed to its *cause*. This means that, for example, broken `#ifdef` statements or unsatisfiable presence conditions are not considered as a bug type, but rather a potential cause of the bug. For instance, Linux commit `66517915e09`[13] fixed an undeclared identifier error caused by a wrong presence condition. The bug types directly indicate what kind of analysis and program verification techniques can be used to address the bugs identified in the analyzed systems. For instance, the category of memory errors (Fig. 8) maps almost directly to various program analyses: for null pointers [14, 24, 28], buffer overruns [10, 22, 58], memory leaks [14, 24], etc.

Bug Description (`descr`*).* Understanding a bug requires rephrasing its nature in general software engineering terms, so that the bug becomes understandable for non-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 domain-specific terminology is unavoidable, we provide links to the necessary

[12] http://cwe.mitre.org/

[13] http://vbdb.itu.dk/#bug/linux/6651791



```
- 6252547b8a7 -

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.../IRQ-domain.txt)
```

(a) Bug record.

```
 1  #include <stdlib.h>
 2
 3  #ifdef CONFIG_TWL4030_CORE          // ENABLED
 4  #define CONFIG_IRQ_DOMAIN
 5  #endif
 6
 7  #ifdef CONFIG_IRQ_DOMAIN            // ENABLED
 8  int irq_domain_simple_ops = 1;
 9
10  void irq_domain_add(int *ops) {                  →(6)
●11      int irq = *ops;                 // ERROR     (7)×
12  }
13  #endif
14
15  #ifdef CONFIG_TWL4030_CORE          // ENABLED
16  void twl_probe() {                               →(3)
17      int *ops = NULL;                             (4)
18      #ifdef CONFIG_OF_IRQ            // DISABLED  |
19      ops = &irq_domain_simple_ops;                |
20      #endif                                       |
21      irq_domain_add(ops);                         (5)→
22  }
23  #endif
24
●25  int main(void) {                               ⇒(1)
26      #ifdef CONFIG_TWL4030_CORE      // ENABLED   ↓
27      twl_probe();                                 (2)→
28      #endif
29      return 0;
30  }
```

(b) Simplified version.

Fig. 4. An example of a bug record and a simplified version of variability bug 6252547b8a7.

background. Obtaining the description is often non-trivial. For example, one bug in our database (Linux 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, 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. To do so, we do a manual in-depth analysis

for every bug found by looking at the feature model (e.g., KConfig) and the mapping (e.g., Makefile) to determine which features must be enabled/disabled in order for the bug to occur. 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). Note that we put all changes made in the feature model (i.e., KConfig) into the header of the simplified bug version.

Error Trace (`trace`). We manually analyze the execution trace that leads to the error state. Slicing tools cannot easily be used for this 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 (which is common). 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/i2ccore.c`, the generic loading function for *I2C*[14] 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. 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 essential problem. The obtained simplified bugs are easily accessible for researchers who would like to try program verification and analysis tools without integrating with each project's build infrastructure, huge header files and dependent libraries, and, most importantly, without understanding the inner workings of these projects. Furthermore, the entire set of simplified bugs constitute an easily accessible benchmark suite derived from real bugs occurring in four highly configurable systems, which can be used to evaluate bug finding tools on a smaller scale.

The 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

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

```
@@ -1,8 +1,4 @@
 #include <stdlib.h>

-#ifdef CONFIG_TWL4030_CORE
-#define CONFIG_IRQ_DOMAIN
-#endif
-
 #ifdef CONFIG_IRQ_DOMAIN
 int irq_domain_simple_ops = 1;
@@ -15,9 +11,9 @@
 #ifdef CONFIG_TWL4030_CORE
 void twl_probe() {
+  #ifdef CONFIG_IRQ_DOMAIN
   int *ops = NULL;
-  #ifdef CONFIG_OF_IRQ
   ops = &irq_domain_simple_ops;
   #endif
   irq_domain_add(ops);
+  #endif
 }

 #endif
```

Fig. 5. Simplified patch for the simplified bug from Figure 4(b).

```
 1  #include <stdlib.h>
 2
 3  #ifdef CONFIG_TWL4030_CORE        // ENABLED
 4  #define CONFIG_IRQ_DOMAIN
 5  #endif
 6
 7  #ifdef CONFIG_IRQ_DOMAIN          // ENABLED
 8  int irq_domain_simple_ops = 1;
 9  #endif
10
•11 int main(void) {                                  ⇒(1)
12      #ifdef CONFIG_TWL4030_CORE    // ENABLED       ↓
13      int *ops = NULL;                              (2)
14      #ifdef CONFIG_OF_IRQ          // DISABLED      |
15      ops = &irq_domain_simple_ops;                  |
16      #endif                                         |
•17     int irq = *ops;               // ERROR         (3)×
18      #endif
19      return 0;
20  }
```

Fig. 6. Single-function version of the simplified bug from Figure 4(b).

the original identifiers for features, functions and variables. However, we abstract away dynamic dispatching via function pointers, structure types, void pointers, casts, and any project specific type, whenever this is not relevant for the bug. For this reason, these simplified versions only represent the original bug from the variability perspective. In particular, if a tool finds one of our simplified bugs, that does not imply that it will find the real bug too. 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.

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 $\texttt{TWL4030\_CORE} \wedge \neg\texttt{IRQ\_DOMAIN}$. We encourage the reader to study the execution trace leading to a crash by starting from `main` at line 25. 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 (enabling/disabling decisions are specified in comments next to the `#if` conditionals).

Simplified Patch. For the same reasons that motivated simplified bugs, we create a simplified patch for each simplified bug that resembles the original bug-fix. A simplified patch helps to understand a bug by explaining how and where it has been fixed. A simplified patch is representative of the real patch when the fix is implemented in the mapping, or in the model. Fixes in the code are represented as faithfully as the simplified bug manages to resemble the real bug.

Figure 5 shows the simplified patch for the simplified bug `6252547b8a7` (cf. Fig. 4(b)). The patch is given in unified diff format (`diff -U2`). To fix the bug, the commit message says that the feature `OF_IRQ`, which encompasses `ops = &irq_domain_simple_ops`, should be removed and wrapping the `IRQ` domain bits of the driver with `IRQ_DOMAIN` instead. Besides that, `TWL4030_CORE` should not depend on `IRQ_DOMAIN`.

Fig. 7. Screenshot of VBDb (bug from Fig. 4).

Single-Function Bug. We also provide single-function versions of the bugs, which are derived from the already simplified versions. Figure 6 shows a single-function bug corresponding to the simplified bug from Fig. 4(b). Single-function bugs are intended to assist the development and evaluation of intraprocedural analysis tools, but can also be useful while debugging interprocedural tools. These single-function versions further help understanding the essence of variability bugs, especially for bugs with deep function call graphs such as `eb91f1d0a53`.

To generate each intraprocedural version, we simply take the `main` method and transitively inline all function calls. Note that for some bugs related to function-calls, a single-function version does not make sense as it would abstract away the bug itself. For instance, bug `7c6048b7c83`, which is an *undefined function* bug, cannot have a single-function version as it would *not* fail to compile as it ought to (i.e., it would *not* preserve the error of the original bug).

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.

We have put all of the studied bugs along with all the information recorded for each of them online with a Web User Interface: http://VBDb.itu.dk/. The raw data is also available online.[15] Figure 7 shows a screenshot of our Web UI database for our sample bug `6252547b8a7` from Fig. 4.

[15] https://bitbucket.org/modelsteam/vbdb/src

L	bug type	CWE	M	B	A	Σ
7	**declaration errors:**		4	5	9	25
4	undefined function	–		2	2	8
2	undeclared identifier	–	4	2	7	15
1	multiple function definitions	–				1
	undefined label	–		1		1
10	**resource mgmt. errors:**			4	5	19
5	uninitialized variable	457		2	1	8
1	memory leak	401		1	2	4
1	use after free	416		1	1	3
2	duplicate operation	675			0	2
1	double lock	764				1
	file descriptor leak	403			1	1
11	**memory errors:**		1	2	4	18
4	null pointer dereference	476		2	2	8
3	buffer overflow	120	1		2	6
3	read out of bounds	125				3
1	write on read only	–				1
8	**logic errors:**		2	3	1	14
5	fatal assertion violation	617				5
2	non-fatal assertion violation	617				2
1	behavioral violation	440	2	3	1	7
4	**type errors:**		4	1	1	10
2	incompatible types	843	2	1	1	6
1	wrong number of func. args.	685	2		0	3
1	void pointer dereference	–				1
2	**dead code:**			3	2	7
1	unused variable	563		3		4
1	unused function	561			2	3
1	**arithmetic errors:**		3			4
1	numeric truncation	197				1
	integer overflow	190	3			3
	validation errors:				1	1
	OS command injection	078			1	1
43	**TOTAL**	–	14	18	23	98

Fig. 8. Types of variability bugs in our study of Linux [1] and all of VBDb. (L is for Linux, M is for Marlin, B is for BusyBox and A is for Apache.)

5 ARE VARIABILITY BUGS LIMITED TO SPECIFIC TYPE OF BUGS, FEATURES, OR LOCATIONS (RQ1)?

In the following, we sometimes aggregate data with numbers. The numbers are used solely to describe the collected sample—no statistical conclusions about the broader bug population should be drawn from them. The reader can use these numbers to get an aggregated characterization of the data in the variability bugs database. That is, the figures presented here serve exclusively to characterize population of bugs we found, not to hint at any representative bug distribution. To emphasize this limited significance of numbers we typeset them in gray.

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

> Confirmed observation 1: Variability bugs are not be limited to any particular type of bug.

L	#occurrences of a feature	M	B	A	Σ
71	**occurs in one VBDb bug:**	17	27	24	139
71	once (1x)	17	27	24	139
12	**occurs in 2+ VBDb bugs:**	2	1	1	16
8	twice (2x)		1		9
4	thrice (3x)	2			6
	four times (4x)				0
	five times (5x)			1	1
83	**TOTAL**	19	28	25	155

Fig. 9. Features involved in variability bugs in Linux and all of VBDb.

Figure 8 lists the type of variability bugs found in the exploratory study of 43 variability bugs in Linux, along with occurrence frequencies in Linux (leftmost column, labeled L for Linux) and associated CWE number whenever applicable (third column). We return to the four rightmost columns shortly. For now, observe that all bug types have been grouped into eight broad error categories, ranging from *declaration errors* to *arithmetic errors* (and one category, *validation errors*, not occurring in the Linux bugs). The groups are shown in gray background with accumulated sub-totals corresponding to each category. For instance, we can see that four of the Linux bugs involved *null-pointer dereferences* (CWE 476) in the broad category *memory errors*, harboring 11 of the Linux bugs.

The prior study *hypothesized* that variability bugs—*in general*—span a wide range of qualitatively different types of bugs [1]. In Figure 8, we see that the variability bugs in Linux span 21 different kinds of bugs, falling into seven broad categories.

We now test the hypothesis by considering the results of our confirmatory case study of three independent systems with variability. The right columns testify how many times a given bug type occurs in each of the systems: M for Marlin, B for BusyBox, and A for Apache. We *confirm* that, *in general*: *variability bugs are not limited to any particular type of bugs*. Just like for Linux, the variability bugs encountered in these systems, also fall into qualitatively different categories.

Considering *all* bugs in the four systems (the Σ column), we see that a staggering 42 of all the variability bugs are caught by the compiler at build time, if compiled in the appropriate configuration: 25 declaration errors, 10 type errors, and seven cases of dead code. Despite the compiler checks, the bugs had been admitted to the code repositories. Since build errors cannot easily be ignored, we take this as evidence that the authors of the commits, and the maintainers that accepted them, were unaware of the bugs, presumably because they did not compile the code in configurations that exhibit the bugs (compiler checks are not family-based).

It appears that conventional automatic code analyzers targeting individual program configurations are insufficient. In order to find the variability bugs in VBDb, analyzers that are able to cope with variability seem to be needed.

> Confirmed observation 2: Variability bugs are not restricted to any specific error prone feature.

Figure 9 shows the number of times a feature is involved in the bugs. We see that the Linux bugs involve a total of 83 different features, ranging from *debugging* options (e.g., `QUOTA_DEBUG` and `LOCKDEP`), through *device drivers* (`TWL4030_CORE` and `ANDROID`), and *network protocols* (`VLAN_8021Q` and `IPV6`), to *computer architectures* (`PARISC` and `64BIT`). As many as 71 of these features are involved only in a single bug; eight are involved in two bugs; and only four features occur in three of the Linux bugs. Thus, there are no obvious particularly "error-prone features" in Linux.

Let us confront the hypothesis with the three systems in our confirmatory case study (the columns: M, B, and A). For example, for BusyBox, we see only one feature, CLEAN_UP that is involved in two bugs. In fact, only one feature in Apache that stands out, namely, APR_HAS_SHARED_MEMORY, which is implicated in five variability bugs. Investigation, however, reveals that four of those occurrences are related to LDAP which, at the time, was an experimental module, thus temporarily of lower quality than others.

In total, the vast majority of features are involved only in a single bug in our collection (139 out of 155, see the Σ column). Only nine features are involved in two bugs and six features in three bugs. The consequence of variability bugs *not* being concentrated around certain error-prone features, is that variability analyzers and sampling strategies for testing and analysis should target system features broadly, not selectively.

> CONFIRMED OBSERVATION 3: Variability bugs are not confined to any specific *location* (file or subsystem).

Figure 10(d) shows a visualization of the organization and relative size of each subsystem in Linux along with the locations of the bugs in our collection. The size of each subsystem is measured in lines of code (LOC); a square (regardless of color) represents 25 KLOC. For instance, the `kernel/` subsystem with six squares, has approximately 150 KLOC constituting about 1% of the Linux code. Superimposed onto the size visualization, the figure *also* shows in which directories the bugs occur. A bug is visualized as a red (darker) square. With five red (dark) squares, the aforementioned directory `kernel/` thus houses five of our VBDb variability bugs. Note carefully that there are two units used in the diagram: LOC represented by the number of squares, and the number of bugs represented by the number of *red* squares. This is a discrete variant of a visualization using two curves of different units in a single graph, where correlation of their dynamics is relevant. It allows us to show the number of bugs with respect to the size of the subsystem in LOC.

We approximate subsystems by existing directory structure. The figure abstracts away *smaller* subsystems accounting for less than 0.1% such as `virt/` (8.1k), as well as *infrastructure*[16] subsystems such as `tools/` (133.1k) and `scripts/` (48.1k). None of these directories contained any of our bugs.

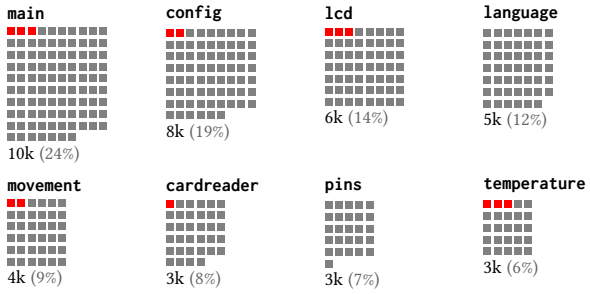
We found bugs in *ten* of the main subsystems in Linux (cf. Fig. 10(d)), suggesting that variability bugs do *not* appear to be confined to any specific subsystem. The bugs occur in qualitatively different subsystems of Linux ranging from *networking* (`net/`) to *device drivers* (`drivers/`, `block/`), 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.

For testing the hypothesis, we collected the corresponding data for the other cases (cf. Figures 10(a), 10(b), and 10(c)). For Marlin, a square visualizes 100 LOC whereas for BusyBox and Apache a square denotes 500 LOC. For Marlin which does not have an appropriate directory structure, we use a logical organization into subsystems. As before, we abstract away *smaller* subsystems.

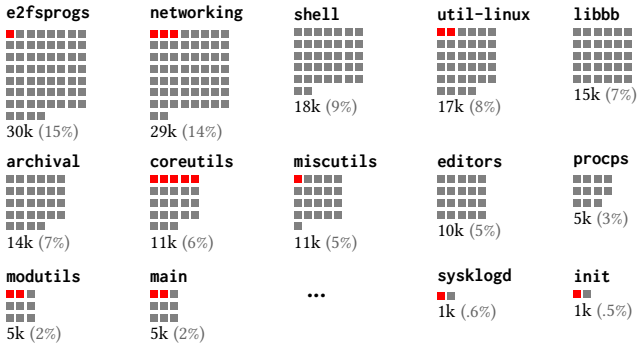
As for Linux, variability bugs in the other systems appear to *not* be confined to any particular subsystems. In fact, only two out of eight subsystems of Marlin do not house any of our bugs. For BusyBox, only three out of 14 subsystems are not represented in VBDb. For Apache, only two out of seven subsystems do not harbor bugs.

The consequence for variability bug hunters, is that there are no short-cuts with respect to subsystems; the analysis needs to target the entire code-base broadly.

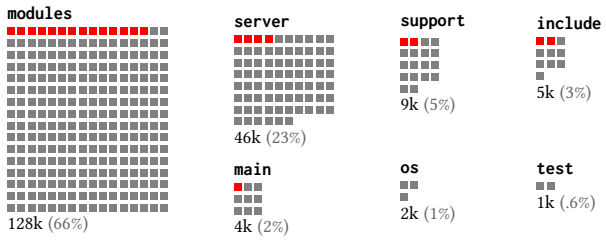
[16] E.g., examples, scripts, documentation, and build infrastructure.



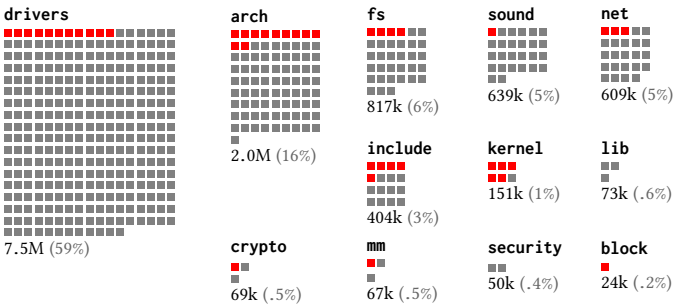
(a) Marlin: ■ (possibly red) = 100 LOC; ■ = 1 bug.



(b) BusyBox: ■ (possibly red) = 500 LOC; ■ = 1 bug.



(c) Apache: ■ (possibly red) = 500 LOC; ■ = 1 bug.



(d) Linux: ■ (possibly red) = 25,000 LOC; ■ = 1 bug.

Fig. 10. Project structure and relative size of subsystems vs location of bugs in VBDb. Note: The figure serves *exclusively* to characterise population of bugs (including their locations), *not* to hint at any representative bug distribution.

Conclusion for RQ1

We are now ready to answer RQ1. Based on analyzing four highly-configurable systems, we conclude that:

> **Conclusion 1:** Variability bugs are *not* confined to any particular *type of bug*, error-prone *feature*, or *location*.

In total, we have found 98 variability bugs falling in 25 different types of error categories, involving 155 distinct features, and spread out in over 30 different subsystems in the four systems investigated.

Variability is ubiquitous. There appears to be no specific *nature* of variability bugs that could be exploited. If analysis tools were to focus on particular flavors of variability bug during family-based analysis, they would thus fail to detect large classes of errors (the flavors not focussed on). Consequently, the analysis of variability bugs in highly-configurable systems needs to be targeted widely at *all* types of bugs, *all* kinds of features, and *all* subsystems. This conclusion is also interesting from the point of view of understanding the reasons for which bugs appear. Appearing everywhere, variability bugs hint that it is the variability itself that enables or amplifies their introduction (possibly standalone, or in concert with other aspects of system complexity). Perhaps this is not so surprising, but now we can *confirm* these folkloric hypotheses with *evidence* in terms of hard data. Further, the tremendous variation among the bugs in the VBDb collection itself provides a useful resource for further research on variability bugs and bug finders. In fact, VBDb has already been used in a variety of recent publications [2, 29, 39]. (We elaborate on this in Section 9.)

6 IN WHAT WAYS DOES VARIABILITY AFFECT BUGS (RQ2)?

We now turn to evidence regarding research question RQ2:

> Confirmed observation 4: Variability bugs may involve *non-locally defined features* (i.e., features defined in another subsystem than where the bug occurred).

In Linux, we have identified 30 bugs that involve non-locally defined features. 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.

We now use our three replication systems to *test* the hypothesis that variability bugs may involve features defined in "remote" subsystems. However, among the three systems considered, only BusyBox permits *local* feature models where KConfig files may be nested to define features that are *local* to subsystems. We thus note that not all highly-configurable systems have a concept of *local features*.

In BusyBox, we identified seven cases of non-locally defined features that testify that bugs may involve variability cross-cutting remote locations in the code. For instance, bug `5cd6461b6fb` occurs due to a wrong format parameter to `printf()` whenever the feature `LFS` (large file support) is enabled. The error occurs in `networking/` whereas the `LFS` feature is defined in the `util-linux/` directory.

For developers of highly-configurable systems, this observation means that when modifying one subsystem, they cannot simply ignore features in other subsystems. Feature definitions may be scattered across subsystems. For tools, this means that they cannot simply zoom in on one subsystem without taking the features defined in other subsystems into consideration.

```
1  #ifdef CONFIG_VLAN_8021Q          // DISABLED
2  void vlan_hwaccel_do_receive() {
3      ...
4  }
5  #else                             // ENABLED
6  void vlan_hwaccel_do_receive() {
7      BUG();                        // ERROR
8  }
9  #endif
10
11 void __netif_receive_skb()
12     vlan_hwaccel_do_receive();    // USAGE
13 }
```

Fig. 11. Excerpt from bug `0988c4c7fb5` illustrating a configuration-dependent definition of a function. In line 12, the function `vlan_hwaccel_do_receive` is invoked. The actual code run, however, will depend on the configuration. If the feature `VLAN_8021Q` is enabled, the function is defined in lines 2–4 will run; otherwise, the function is defined in lines 6–8 will run (which provokes an assertion violation in line 7).

Confirmed observation 5: The use of a function, variable, macro, or type may involve *implicit variability* caused by configuration-dependent definitions.

We investigated *configuration-dependent definitions* (functions, variables, macros, and types) that are defined differently in different configurations, or conditionally defined in only some configurations whose use in other configurations provokes an error. Configuration-dependent definitions complicate the identification of variability-related problems as the variability is *implicit*, most often hidden in a header file, or in another translation unit. Even if variability is explicit in the definition, it is *not* visible at the usage location.

In Linux, for instance, bug `242f1a34377` involves a *conditionally dependent definition*; the function `crypto_alloc_ablkcipher()` is only defined whenever `CRYPTO_BLKCIPHER` is *enabled*. The bug occurs due to a function call to `crypto_alloc_ablkcipher()` in another file, leading to an *undefined function* error (Fig. 8) when `CRYPTO_BLKCIPHER` is *disabled*.

For an example of different definitions in different configurations, consider Linux bug `0988c4c7fb5`. Figure 11 shows an exerpt of this bug. Here, the 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. (In reality, the two alternative functions are defined in different files.) Variants without `VLAN_8021Q` support are compiled with a mockup-implementation of this function that unconditionally enters an error state. The definition clearly involves variability. Its use, however, shows no apparent involvement of variability. Deceptively, the definition of the function itself (in lines 6–8), appears to involve no variability. However, since the function definition is wrapped inside a conditional `#ifdef` annotation, the error will only occur whenever the feature `VLAN_8021Q` is disabled.

Another example is bug `0f8f8094d28`, where a variability-dependent macro definition is involved. It can be regarded as a simple 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, and only for a particular virtual page size, are affected. Macro `KMALLOC_SHIFT_HIGH` has alternative definitions at different source locations.

Perhaps an even more subtle example of implicitly variable code is a conditional *if* statement with guard on the size of a type: for instance `(sizeof(type) != 0)`, which introduces dependency of code execution on a *type* being defined as non-empty under some feature condition. Type

L	layer	M	B	A	Σ
39	**single layer:**	14	17	23	93
28	code	11	7	14	60
5	mapping	3	9	9	26
6	model	–	1	–	7
4	**multiple layers:**		1		5
2	code & mapping		1		3
1	mapping & model	–		–	1
1	code & mapping & model	–		–	1
43	**TOTAL**	14	18	23	98

Fig. 12. Bug-fixing layers.

declarations are typically made in header files, and they are not immediately visible in the use place. Such cases are rather difficult to handle by simple extensions to single-program analyzers, as variability in the imperative code is mixed with the variability in the type language of the program (and even worse so via size properties of types). An example of such implicit variability can be found in bug `218ad12f42e`, involving a selected field in the structure type `rwlock_t`.

It turns out that implicit variability likely appears in Linux's source code due to internal coding conventions. The following coding guidelines on `#ifdef` usage from *How to Get Your Change Into the Linux Kernel*[17] advises:

> *"Code cluttered with ifdefs is difficult to read and maintain. Don't do it. Instead, put your ifdefs in a header, and conditionally define 'static inline' functions, or macros, which are used in the code."*

We now consider configuration-dependent definitions involved in variability bugs in our three independent systems.

In Marlin, bug `831016b`, for instance, involves the function, `lcd_setstatus`, which is defined to take *two* arguments when the feature `ULTRA_LCD` is enabled and only *one* argument whenever `ULTRA_LCD` is disabled. However, whenever `SDSUPPORT` is *enabled* and `ULTRA_LCD` is *diabled* (2-degree bug), `lcd_setstatus` is erroneously invoked with *two* arguments (instead of *one*).

In BusyBox, bug `bc0ffc0e971` involves a function called `delete_eth_table()` that has two different definitions depending on whether feature `CLEAN_UP` is enabled or not. Variants without `CLEAN_UP` are compiled with a mockup implementation of this function (which, like in Linux, appears to be common practice). BusyBox bug `5cd6461` involves the use of a variable `total` which, depending on whether the feature `LFS` is enabled or not, is defined either as a `long long` or a `long`. However, in configurations where `LFS` is disabled, when attempting to print the value of the `total`, `printf` is erroneously invoked with the format `%ld` (`long`) which ought to have been `%lld` (`long long`).

For developers, configuration-dependent definitions means that programs may deceptively involve variability even though they appear not to. For analyzers, this means that variability tools should make sure to associate definitions with presence conditions (i.e., keep associations between definitions and configurations).

> CONFIRMED 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 even fixed in a *combination* of these layers.

[17] https://www.kernel.org/doc/Documentation/SubmittingPatches

A bug can be fixed in the *code*, *mapping*, and *model* (cf. Section 2). Since bug fixes often involve multiple locations, variability bugs can occur in multiple layers. Figure 12 shows whether the bugs in our sample were fixed in the *code*, *mapping*, *model*, or combinations thereof. For our replication studies, please note that Marlin and Apache have no notion of *feature model* (at least, not in the classical sense). Instead, these projects capture feature dependencies operationally, as we do in our simplified bugs (see lines 3–5 of Fig. 4(b)). We therefore include a dash in the figure for layers involving the *model*.

In Linux, commits `472a474c663` and `7c6048b7c83`, fix variability bugs in the *mapping* and *model*, 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.

Linux 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. We also found one Linux bug, `e68bb91baa0`, that was fixed in all the three layers.

In Figure 12, we see that the variability bugs in Marlin, BusyBox, and Apache are also not only fixed in the *code*, but in various layers. Although, like for Linux, the variability bugs appear to be fixed predominantly in the *code* and *mapping* layers. In BusyBox, commit `5cd6461b6fb` fixes an incompatible type bug, caused by a wrong format parameter in a `printf()` method, in multiple layers, by changing the *code* and *mapping* layers.

In total (the Σ column), note that, even though we only documented bugs that manifested themselves in code, 38 bugs in our sample were, in fact, *not* exclusively fixed in the code: 26 bugs were fixed exclusively in the *mapping*, seven exclusively in the *model*, and five in multiple layers.

The 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, in Linux, the dependency of `TWL4030_CORE` on `IRQ_DOMAIN` removed by 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`.

The realization that bugs in highly-configurable software might need to be fixed outside the main code, is congruent with the work of Passos and co-authors [48], who observe that evolution of features in the Linux kernel involves all the three layers. This should inform research on bug finding and bug fixing. For instance, it is not sufficient to look at the feature model in isolation in order to find complex bugs, yet most of the research on analysis of feature models does exactly that [6]. Similarly, for bug fixing techniques [25], it is not sufficient to synthesize patches for C programs—changes to the preprocessor directives and build scripts (that specify the mapping), as well as to the feature model should be considered, too.

> Confirmed observation 7: Many variability bugs involve multiple features and are hence *feature-interaction bugs*.

We define the *variability degree* of a bug (or just the *degree* of a bug), as the number of individual features occurring in its presence condition. Intuitively, the degree of a bug indicates the number

L	degree	M	B	A	Σ
8	**single-feature bugs:**	**7**	**9**	**17**	**41**
8	1-degree	7	9	17	41
35	**feature-interaction bugs:**	**7**	**9**	**6**	**57**
22	2-degree	3	6	4	35
9	3-degree	4	3	1	17
1	4-degree				1
3	5-degree			1	4
43	**TOTAL**	**14**	**18**	**23**	**98**

Fig. 13. Variability degrees.

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 known as *variability bugs*, involving one or more features, thus occur in a non-empty strict subset of valid configurations. In particular, if the degree of a bug is strictly 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*.

Figure 13 summarizes the variability degrees of the bugs studied; there are 57 of those in our bug collection and 22 of those involve three features or more.

Our exploratory study of Linux identified 35 so-called feature-interaction bugs. For instance, Linux bug `6252547b8a7` (cf. Fig. 4(b)) is a feature interaction bug. 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 [41, 42], and outright infeasible in practice, as confirmed to us by a professional Linux developer, whom we interviewed.

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 an attempt to store some large 64-bit value in one of these 32-bit fields, causing a negative value to be stored instead. Interestingly, the Linux developer we interviewed also mentioned the difficulty to optimize for cache-misses due to variability in the alignment of struct fields.

Linux bug `ae249b5fa27`, constitutes a 3-degree bug caused by the interaction of `DISCONTIGMEM` (efficient handling of discontinuous 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`). Linux bug `218ad12f42e` is a 4-degree bug that has a memory leak which occurs when an array of locks is allocated if `SMP` or any of two particular debugging options are enabled; but is not freed if feature `NUMA` is present. We also found 5-degree bugs such as commit `221ac329e93`, again in Linux, due to 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.

L	precondition	M	B	A	Σ
21	**some enabled:**	**9**	**7**	**14**	**49**
5	a	6	3	7	21
10	$a \wedge b$	3	3	5	21
5	$a \wedge b \wedge c$		1		6
1	$a \wedge b \wedge c \wedge d \wedge e$				1
20	**some-enabled-one-disabled:**	**4**	**11**	**10**	**45**
3	$\neg a$	1	6	10	20
13	$a \wedge \neg b$	3	4		20
3	$a \wedge b \wedge \neg c$		1		4
1	$a \wedge b \wedge c \wedge d \wedge \neg e$				1
2	**other configurations:**	**1**		**1**	**4**
1	$\neg a \wedge \neg b$				1
	$a \wedge \neg b \wedge \neg c$	1			1
1	$a \wedge \neg b \wedge \neg c \wedge \neg d \wedge \neg e$			1	2
43	**TOTAL**	**14**	**18**	**23**	**98**

Fig. 14. Presence conditions under which the bugs occur.

Looking at the data for our replication studies, we see another 22 feature interaction bugs; seven in Marlin, nine in BusyBox, and six in Apache. The Linux study revealed 13 bugs with a degree of at least three; the replication study uncovered another nine such high-degree bugs.

BusyBox bug `95755181b82` is a logic error involving three features interacting with each other: `BB_MMU`, `HTTPD_GZIP`, and `HTTPD_BASIC_AUTH`. With `HTTPD_GZIP` enabled, if a request contained "`AcceptEncoding: gzip`", then the HTTP error response would be incorrectly marked as being gzip encoded ("`Content-Encoding: gzip`") even though it is not. Marlin bug `b8e79dc` is a 3-degree bug; it occurs only whenever `ULTRA_LCD` is enabled and `ENCODER_RATE_MULTIPLIER` as well as `TEMP_SENSOR_0` are disabled. In Apache, the bug `c76df14` is also a 3-degree bug that occurs whenever `CROSS_COMPILE` is enabled and either `WIN32` or `OS2` are enabled.

It is interesting to note that more than half of the bugs in our VBDb collection are, in fact, *feature-interaction bugs* (cf. the Σ column in Figure 13). While most feature-interaction bugs have been identified, documented, and published in telecommunication domain [15], this study provides a documented collection of feature-interaction bugs in the context of a wider collection of highly-configurable systems.

Feature-interaction bugs are inherently more complex to find and reason about [41] because the number of variants, that a developer needs to consider, is *exponential* in the degree of the bug (number of features involved). This impacts both variability program developers and analyzers that consequently have to cope with this combinatorial blow up.

> Confirmed observation 8: Presence conditions for variability bugs may also involve *disabled* features.

In our exploratory study of Linux, we observed that the precense conditions, under which the bugs occur, often involved disabled features. Figure 14 lists and groups the structure of the presence conditions. Two main classes of bug presence conditions emerged: *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 21 bugs in *some-enabled* configurations, and another 20 bugs in *some-enabled-one-disabled*. Only two configurations fell outside these two categories. Please note that a few of the presence conditions have 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

configuration test strategy	sample size	benefit
all enabled (*maximal*)	$O(1)$ in practice	50% (49/98)
one disabled	maximum $\lvert\mathbb{F}\rvert$	96% (94/98)
exhaustive (all configs.)	maximum $2^{\lvert\mathbb{F}\rvert}$	100% (98/98)

Fig. 15. Effectiveness (cost/benefit) of various testing strategies if applied to our collection of bugs.

the *some-enabled-one-disabled* class. Similarly, for presence conditions of the form $(a \vee a') \wedge b$, we classified as *some-enabled*. (For this reason, Fig. 13 and Fig. 14 may appear inconsistent.)

Considering our replication studies, we see the same pattern. A total of 25 bugs in the replication studies fall into the *some-enabled-one-disabled* category, involving disabled features: four in Marlin, eleven in BusyBox, and ten in Apache. Similarly to Linux, only two bugs fall outside the two categories (one in Marlin and one in Apache). In total (the Σ column), the contents of VBDb amounts to 49 bugs in *some-enabled* configurations, and another 45 bugs in *some-enabled-one-disabled*. Only four configurations fall outside the two main categories identified.

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.

> Confirmed observation 9: A *one-disabled* testing strategy, with a sample size bounded by the number of features, would find 96% of bugs in our collection.

We propose a *one-disabled* configuration testing strategy, which considers a maximal configuration and then disables each of the individual features, one by one.

Figure 15 compares the two strategies, *all-enabled* (maximal) configuration testing and *one-disabled* configuration testing. The *sample size* is the number of configurations generated by the given formula (an upper bound). For the *all-enabled* strategy this number is approximate: in practice, since feature models are underconstrained [43], a small number of configurations will suffice for real systems (thus constant in practice). In the worst case *all-enabled* degrades to *one-enabled* (the dual of *one-disabled*), but the authors have yet to see a pathological system like that. For *one-disabled*, the size of the sample is always at most $|\mathbb{F}|$, the maximum occurs if all features can be disabled independently.

The benefit is measured as bug coverage for our sample: for each strategy we check what percentage of bugs in our database would be detected by them. We also add an entry for *exhaustive* testing of all configurations, serving as a baseline. For exhaustive testing, the sample size is exponential in $|\mathbb{F}|$. This is in practice reduced by feature constraints, but not below the exponential growth due to sparsity of the constraints, at least not in highly configurable systems (some software product lines, in contrast, have very small configuration spaces).

All-enabled (maximal) appears to be a fairly good heuristic intercepting exactly half of the bugs in our sample at a constant cost (in terms of the number of configurations considered). 49 out of 98 the bugs could be found this way. *One-disabled* configuration testing has a linear cost in $\mathbb{F}$ and thus can scale reasonably well. Remarkably, 96% of the bugs in VBDb (94 out of 98) 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 the hypothetical configuration requiring *all* features to be enabled).

In practice, we must consider the effect of the feature model in the testing strategy. Because some features depend on others to be present, we often cannot disable features individually. A [Max]SAT solver is required in order to enumerate the configurations to test, while selecting *valid* configurations only. 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), the size of real-world program families (more often only hundreds of features) and sparsity of their constraint systems [43].

The proposed *one-disabled* sampling strategy is related to other well-established strategies discussed in literature, including the most popular *t-wise* (also known as combinatorial interaction testing [13, 18, 20]), as well as other heuristic strategies such as *all-enabled*, *all-disabled*, *code-coverage* [52–54] and *random sampling* strategies. Medeiros et al. [39] executed a comparative quantitative study of effectiveness of various sampling strategies for testing and analysis of configurable systems, including all the above, one-disabled[18] and its dual version, *one-enabled*, added for symmetry. Like suggested above, they use a solver to enumerate (almost perfectly) *one-disabled* and *one-enabled* configurations that satisfy feature constraints.

For large sampling problems, and in the present of feature constraints, Medeiros et al. report that *one-disabled* finds more bugs than pair-wise testing, and it scales better [39]. In fact, *one-disabled* is the only non-trivial method that is able to scale to all of the Linux kernel among those that they studied. None of the *t*-wise methods do. Besides *one-disabled*, only the simple sampling strategies scale, but with worse fault detection rate (*one-enabled*, *all-enabled*, *all-disabled*, and random sampling). It appears though that classic combinatorial interaction testing techniques are a better choice for small configuration spaces. We refer the reader to the original work of Medeiros et al. for a much more comprehensive discussion, including the delimitation of conclusion threats.

Conclusion for RQ2

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 increases the complexity of bugs along several dimensions:

- 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 the developers;
- Variability may be implicit and even hidden in alternative or conditionally defined function, macro, variable, and type definitions specified at remote 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 (e.g., C, Cpp, GNU Make and Kconfig for Linux);
- Not all these bugs will be detected by maximal configuration testing due to interactions with *disabled* features;
- The existence of compiler errors in committed code trees shows that conventional feature-insensitive tools are not enough to find variability bugs.

[18]The *one-disabled* strategy was known to them thanks to personal communication with the authors of the present paper who proposed the strategy in an earlier version [1].

7 THREATS TO VALIDITY

We now consider first *internal*, then *external* validity.

7.1 Internal Validity

Bias due to selection process. As we extract bugs from commits, our collection is biased towards bugs that were not only found and reported, but also fixed. Since users run a small subset of possible configurations, and developers lack feature-sensitive tools, potentially only a subset of bug categories and properties is found this way.

Further, our keyword-based search relies on the competence of developers to properly identify and report variability in bugs. Note, however, that in our subject systems, variability is ubiquitous and often "hidden". For instance, in Linux 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. So certain kinds of bugs involving variability might have been missed, as they were not clearly identified as such by developers. Additionally, note that we focused on semantic variability errors that have been confirmed by the developers, minimizing the risk of studying fictitious problems. Syntactic variability errors consist of a small percentage of bugs. In fact, researchers have found that syntactic variability errors are indeed not common [40]. For this reason, we focused on this range that seems most relevant. Anything beyond that it is out of the scope of the paper.

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 like representativeness in Marlin, BusyBox, Apache, and Linux. 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. The analysis of the bugs is not run by domain experts, which introduces the risk of mistaken identification of bugs. This also applies to determining the presence condition of each bug (under which configurations the bug does and does not occur). By only considering variability bugs that have been identified and fixed by the developers, we mitigate the risk of introducing false positives. We only take bug-fixing commits from the subjects repositories, the commits of which have been reviewed by other developers and, particularly, by a more experienced maintainer. All of our data have been validated by at least two researchers.

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, Linux commit `b1cc4c55c69` adds a nullity check for a pointer that is guaranteed not to be null.[19] 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 complex systems such as Marlin, BusyBox, Apache, and 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

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

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.

7.2 External Validity

Preprocessors. Our study is dedicated and tailored to a particular technique for dealing with variability: preprocessors. Since developers often use preprocessors [23, 33, 37], which is a well-known technique, mainly in industry, to implement features in the code level. Generalization to other variability techniques is not intended.

Number of bugs. The size of our sample speaks against the generalizability of the observations. However, as we explained before, we firstly analyzed a diverse set of 42 variability bugs in an exploratory manner (cf. our previous work [1]). Then, we took three others highly-configurable systems (Marlin, BusyBox, and Apache) and analyzed another 55 bugs to reinforce our observations, following a confirmatory case study research method. We also added a 43rd Linux bug, which came from an external contributor. The process of collecting and especially analyzing these 98 bugs cost several man-months, which makes a study of a much larger number of bugs infeasible. We hope that our database will continue to grow, also from third-party contributions, in the future.

Simplicity bias. Since we considered bugs that were already found, reported, confirmed, and fixed, our collection might be biased towards simpler rather than more complex bugs. Presumably, however, this bias mainly applies to bugs that do not manifest themselves with clear symptoms. Bugs causing real problems obviously stand a higher chance of being caught by the developers. We wanted to study a *wider* range of bugs occuring in real systems; for this reason, we adopted a manual strategy rather than studying a *narrower* set of errors for which bug finders happen to exist (and scale to Linux). Note regarding external validity that even if we had compiled a bug collection based on errors reported by automated bug detection tools, we would still have had a similar bias towards simplicity. After all, simpler bugs are easier to find, not only for humans, but also for tools. In addition, tools would introduce the additional risks of studying fictitious problems disguised as false positive errors reported by the tools.

Subject studies. We used four *open-source* highly-configurable systems in our study: Marlin, BusyBox, Apache, and Linux. These are qualitatively different systems in terms of size, purpose, variability and complexity. Besides that, all have different architectures and developers, which allows us to draw slightly broader conclusions. However, we acknowledge that our claims might not generalize to all other highly-configurable systems, especially commercial ones, which require further investigation.

Usability of the data for other studies. All future studies using this data should very carefully consider the threats described above, following from the collection (for instance drawing statistical conclusions solely based on this data is not sound). Example good applications of this data are qualitative: one can use it to extract new hypotheses, learn about properties of problems, or pre-test hypotheses. Any actual statistical hypotheses should be cross-checked on random samples of bugs (this one is not random). Our main intention for use of this data, is to scaffold tool development. Simplified bugs can be used to build the tools faster and to experiment earlier. The evaluation of the tools on our simplified bugs can show feasibility of solving problems. Scalability should be tested on the original (not simplified bugs). Precision and recall should be measured on representative samples (this one is not).

8 RELATED WORK

This paper extends previous work [1]. Beyond 42 bugs in Linux, this paper confirms our previous hypotheses by considering 55 variability bugs from three other highly configurable systems: Marlin, BusyBox, and Apache. In terms of the database, we added simplified patches and single-function versions of all bugs.

We have divided this section into work on *bug databases*, *mining variability bugs*, and *methodologically related work*.

8.1 Bug Databases

ClabureDB is a database of bug-reports for the Linux kernel with similar purpose to ours [50], 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).

Palix *et al.* reproduced an old analysis (from 2001) on Linux to reevaluate and investigate the evolution of bugs in Linux over the last decade [47]. The results are available in a public archive.[20] This study has identified a series of bugs and rule violations such as "do not use floating point in the Linux kernel". However, variability was not in their focus. We in turn focus on qualitatively understanding the complexity and nature of variability bugs. In addition, we consider four qualitatively different open-source software systems.

Do *et al.* provided an infrastructure to help the execution of controlled experiments related to software testing techniques [21]. The idea is to support reproducible experimentation and minimize certain challenges when performing a new study, such as the high costs when gathering proper artifacts for the controlled experiment. To do so, the infrastructure provides elements to execute test cases (e.g., oracles, test classes, stubs, etc) and inputs to reveal faults. Similarly, VBDb can also contribute to future studies and experiments, but it is a more specific data infrastructure, since we focus only on bugs related to *variability*. In this context, future research can benefit, for example, from the simplified bugs (which can reduce effort when compared to understanding the actual bugs) and from the inputs, including configurations, that reveal them. In addition, the database might be used to conduct an empirical study to better understand how developers introduce variability bugs in highly-configurable systems. The work that introduces the infrastructure [21] also includes a list of research already using and benefiting from it. VBDb has also already been used in a variety of recent publications [2, 29, 39].

8.2 Mining Variability Bugs

Nadi *et al.* mined the Linux repository to study *variability anomalies* [45]. 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 [5], using a technique known as *variability encoding* (*configuration lifting* [49]). 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

[20] http://faultlinux.lip6.fr/

with a model-checker searching for a particular class of interactions. Understanding variability bugs should lead to building scalable bug finders, enabling studies like [5] to be run for Linux in the future.

Medeiros *et al.* have studied *syntactic* variability errors [40]. They used a variability-aware C parser [34] 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 [44]. 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.

8.3 Methodologically Related Work

Tian *et al.* studied the problem of distinguishing bug fixing commits in the Linux repository [57]. 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 contrast, we use the keyword-based method for pragmatic reasons. First, our main emphasis was on *analyzing* commits, whereas finding them was secondary and not difficult for our study. That is, in our study most of the time was invested in *analyzing* commits, and not in using a precise method with a high recall of finding potential bugs. Second, this is a straightforward method to apply in any project that stores historical information on changes. Thus, we found the keyword-based method sufficient for our purpose.

Yin *et al.* collect hundreds of errors caused by misconfigurations in open source and commercial software [59] 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 [46]. 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.

9 CONCLUSION

Previously, we have conducted an exploratory case study of variability bugs in Linux which led to nine testable hypotheses [1]. We subsequently performed a confirmatory case study involving three independent replications: Apache, BusyBox, and Marlin. The study confirmed all hypotheses.

In total, we studied 98 variability bugs in four highly-configurable systems. For each of the bugs, we analyzed relevant variability properties and condensed our understanding of 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. Also, we created simplified patches, and single-function versions of the bugs for evaluation of prototype and intraprocedural analyses.

We conclude that variability bugs are not confined to any particular *type of bugs*, error-prone *features*, or specific *locations* (see Section 5). Hence, analysis tools aiming to find variability bugs in highly-configurable systems need to be targeted widely at all types of bugs, all kinds of features, and all subsystems.

We also characterize in what ways variability affects bugs (see Section 6). In addition to introducing an exponential number of program variants, variability increases the complexity of bugs along several dimensions:

- Variability bugs may involve undesired *feature-interactions* (e.g., via program variables);
- Feature-interactions may span multiple subsystems (demanding cross-subsystem knowledge);
- Variability may be implicitly hidden in *configuration-dependent definitions*;
- Variability bugs may occur in multiple layers (*code*, *mapping*, and/or *model*)
- These layers involve different languages (e.g., C, Cpp, Gnu Make and Kconfig for Linux);
- Variability bugs may involve *disabled* features (thus not all variability bugs will be detected by maximal configuration testing); and
- The existence of compiler errors in committed code trees shows that conventional feature-insensitive tools are not enough to find variability bugs.

A natural direction to continue this work would be to design quantitative studies to confirm our qualitative observations. Such studies can be designed in two directions: either by building suitable tools and applying them massively to the available historical source code, or by designing controlled experiments when programmers are observed during programming, with attention to bug finding and bug fixing tasks. Observing bug introduction however is very difficult in a quantitative manner, and would have to be done qualitatively.

Some of these observations may lead to better sampling strategies for configurable systems, or optimizations for family-based analysis, which is our main envisioned direction for the future. This work has already influenced a quantitative study on the effectiveness of sampling strategies for configurable systems [39]. Additionally, Iosif-Lazar et al. [29] used our dataset to evaluate their variability-related transformations, which translate program families into single programs by replacing compile-time variability with run-time variability. Al-Hajjaji et al. [2] also used our database to derive a set of mutation operators for software with preprocessor-based variability. We thus hope that our variability bugs database will continue being useful to the variability research community, especially to designers of program analysis and bug finding tools. At the same time, we also hope that the community can contribute to the usefullness of this data by providing new bug reports and new simplified bugs. The VBDb project allows contributions as pull requests against its bitbucket repository and as discussion comments in the online website.

ACKNOWLEDGMENTS

We thank Linux kernel developers, Jesper Brouer and Matias Bjørling. Nicolas Dintzner pointed us to an interesting commit that became the 43rd Linux bug in our database. This work has been supported by The Danish Council for Independent Research under a Sapere Aude project, VARIETE. Jean Melo is funded by the Brazilian program *Science without Borders*, grant no. 249020/2013-0.

REFERENCES

[1] Iago Abal, Claus Brabrand, and Andrzej Wasowski. 2014. 42 Variability Bugs in the Linux Kernel: A Qualitative Analysis. In *Proceedings of the 29th IEEE/ACM International Conference on Automated Software Engineering (ASE '14)*.

[2] Mustafa Al-Hajjaji, Fabian Benduhn, Thomas Thüm, Thomas Leich, and Gunter Saake. 2016. Mutation Operators for Preprocessor-Based Variability. In *Proceedings of the Tenth International Workshop on Variability Modelling of Software-intensive Systems (VaMoS '16)*. ACM, New York, NY, USA, 81–88. https://doi.org/10.1145/2866614.2866626

[3] Sven Apel, Don Batory, Christian Kästner, and Gunter Saake. 2013. *Feature-Oriented Software Product Lines*. Springer-Verlag.

[4] Sven Apel, Christian Kästner, Armin Grösslinger, and Christian Lengauer. 2010. Type safety for feature-oriented product lines. *Automated Software Engineering* 17 (2010). Issue 3.

[5] Sven Apel, Hendrik Speidel, Philipp Wendler, Alexander von Rhein, and Dirk Beyer. 2011. Detection of Feature Interactions using Feature-Aware Verification. In *Proceedings of the 26th IEEE/ACM International Conference on Automated Software Engineering (ASE'11)*. IEEE Computer Society, Lawrence, USA.

[6] David Benavides, Sergio Segura, and Antonio Ruiz Cortés. 2010. Automated analysis of feature models 20 years later: A literature review. *Inf. Syst.* 35, 6 (2010), 615–636. https://doi.org/10.1016/j.is.2010.01.001

[7] Thorsten Berger, Ralf Rublack, Divya Nair, Joanne M. Atlee, Martin Becker, Krzysztof Czarnecki, and Andrzej Wasowski. 2013. A survey of variability modeling in industrial practice. In *VaMoS*, Stefania Gnesi, Philippe Collet, and Klaus Schmid (Eds.). ACM.

[8] Thorsten Berger, Steven She, Rafael Lotufo, Andrzej Wasowski, and Krzysztof Czarnecki. [n. d.]. A Study of Variability Models and Languages in the Systems Software Domain. *IEEE Trans. Software Eng.* 39, 12 ([n. d.]).

[9] Eric Bodden, Társis Tolêdo, Márcio Ribeiro, Claus Brabrand, Paulo Borba, and Mira Mezini. 2013. SPL$^{\text{LIFT}}$ - Statically Analyzing Software Product Lines in Minutes Instead of Years. In *PLDI'13*.

[10] Ella Bounimova, Patrice Godefroid, and David Molnar. 2013. Billions and Billions of Constraints: Whitebox Fuzz Testing in Production. In *Proceedings of the 2013 International Conference on Software Engineering (ICSE '13)*. IEEE Press, Piscataway, NJ, USA.

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

[12] Claus Brabrand, Márcio Ribeiro, Társis Tolêdo, Johnni Winther, and Paulo Borba. 2013. Intraprocedural Dataflow Analysis for Software Product Lines. *Transactions on Aspect-Oriented Software Development* 10 (2013).

[13] K. Burr and W. Young. 1998. Combinatorial test tech- niques: Table-based automation, test generation and code coverage. In *In Proc. of the Intl. Conf. on Software Testing Analysis & Review*.

[14] William R. Bush, Jonathan D. Pincus, and David J. Sielaff. 2000. A Static Analyzer for Finding Dynamic Programming Errors. *Softw. Pract. Exper.* 30, 7 (June 2000).

[15] Muffy Calder, Mario Kolberg, Evan H. Magill, and Stephan Reiff-Marganiec. 2003. Feature Interaction: A Critical Review and Considered Forecast. *Comput. Netw.* 41, 1 (2003).

[16] Andreas Classen, Patrick Heymans, Pierre-Yves Schobbens, and Axel Legay. 2011. Symbolic model checking of software product lines. In *ICSE*.

[17] Andreas Classen, Patrick Heymans, Pierre-Yves Schobbens, Axel Legay, and Jean-François Raskin. 2010. Model checking lots of systems: efficient verification of temporal properties in software product lines. In *ICSE'10*. ACM, Cape Town, South Africa.

[18] D. M. Cohen, S. R. Dalal, M. L. Fredman, and G. C. Patton. 1997. The AETG system: an approach to testing based on combinatorial design. *IEEE Transactions on Software Engineering* 23, 7 (1997), 437–444.

[19] Krzysztof Czarnecki and Krzysztof Pietroszek. 2006. 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)*. ACM, New York, NY, USA.

[20] S. R. Dalal, A. J. N. Karunanithi, J. M. L. Leaton, G. C. P. Patton, and B. M. Horowitz. 1999. Model-based testing in practice. In *Proc. of the Intl. Conf. on Software Engineering (ICSE '99)*. 285–294.

[21] Hyunsook Do, Sebastian Elbaum, and Gregg Rothermel. 2005. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engineering: An International Journal* 10 (2005), 405–435.

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

[23] Michael D. Ernst, Greg J. Badros, and David Notkin. 2002. An Empirical Analysis of C Preprocessor Use. *IEEE TRANSACTIONS ON SOFTWARE ENGINEERING* 28, 12 (2002), 2002.

[24] David Evans. 1996. Static Detection of Dynamic Memory Errors. *SIGPLAN Not.* 31, 5 (1996).

[25] Claire Le Goues, Stephanie Forrest, and Westley Weimer. 2013. Current challenges in automatic software repair. *Software Quality Journal* 21, 3 (2013), 421–443. https://doi.org/10.1007/s11219-013-9208-0

[26] Alexander Gruler, Martin Leucker, and Kathrin D. Scheidemann. 2008. Modeling and Model Checking Software Product Lines. In *FMOODS*.

[27] Gerald Holl, Michael Vierhauser, Wolfgang Heider, Paul Grünbacher, and Rick Rabiser. 2011. Product line bundles for tool support in multi product lines. In *VaMoS*.

[28] David Hovemeyer and William Pugh. 2007. 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)*. ACM, New York, NY, USA.

[29] Alexandru Florin Iosif-Lazar, Jean Melo, Aleksandar S. Dimovski, Claus Brabrand, and Andrzej Wasowski. 2017. Effective Analysis of C Programs by Rewriting Variability. *CoRR* abs/1701.08114 (2017). http://arxiv.org/abs/1701.08114

[30] Kyo Kang, Sholom Cohen, James Hess, William Nowak, and Spencer Peterson. 1990. *Feature-Oriented Domain Analysis (FODA) Feasibility Study*. Tech. Rep. CMU/SEI-90-TR-21. CMU-SEI.

[31] Christian Kästner. 2010. *Virtual Separation of Concerns: Toward Preprocessors 2.0.* Ph.D. Dissertation. Marburg, Germany.
[32] Christian Kästner and Sven Apel. 2008. 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.
[33] Christian Kästner, Sven Apel, and Martin Kuhlemann. 2008. Granularity in Software Product Lines. In *Proceedings of the 30th International Conference on Software Engineering (ICSE '08)*. ACM, New York, NY, USA, 311–320. https://doi.org/10.1145/1368088.1368131
[34] Andy Kenner, Christian Kästner, Steffen Haase, and Thomas Leich. 2010. TypeChef: Toward Type Checking #Ifdef Variability in C. In *Proceedings of the 2Nd International Workshop on Feature-Oriented Software Development (FOSD '10)*. ACM, New York, NY, USA.
[35] Chang Hwan Peter Kim, Eric Bodden, Don Batory, and Sarfraz Khurshid. 2010. Reducing Configurations to Monitor in a Software Product Line. In *1st International Conference on Runtime Verification (RV) (LNCS)*, Vol. 6418. Springer, Malta.
[36] Kim Lauenroth, Klaus Pohl, and Simon Toehning. 2009. Model Checking of Domain Artifacts in Product Line Engineering. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering (ASE '09)*. IEEE Computer Society, Washington, DC, USA, 269–280. https://doi.org/10.1109/ASE.2009.16
[37] Jörg Liebig, Sven Apel, Christian Lengauer, Christian Kästner, and Michael Schulze. 2010. An Analysis of the Variability in Forty Preprocessor-based Software Product Lines. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1 (ICSE '10)*. ACM, New York, NY, USA, 105–114. https://doi.org/10.1145/1806799.1806819
[38] R. Love. 2010. *Linux Kernel Development.* Pearson Education.
[39] Flávio Medeiros, Christian Kästner, Márcio Ribeiro, Rohit Gheyi, and Sven Apel. 2016. A Comparison of 10 Sampling Algorithms for Configurable Systems. In *Proceedings of the 38th International Conference on Software Engineering (ICSE '16)*. ACM, New York, NY, USA, 643–654. https://doi.org/10.1145/2884781.2884793
[40] Flávio Medeiros, Márcio Ribeiro, and Rohit Gheyi. 2013. Investigating Preprocessor-based Syntax Errors. In *Proceedings of the 12th International Conference on Generative Programming: Concepts & Experiences (GPCE '13)*. ACM, New York, NY, USA.
[41] Jean Melo, Claus Brabrand, and Andrzej Wąsowski. 2016. How Does the Degree of Variability Affect Bug Finding?. In *Proceedings of the 38th International Conference on Software Engineering (ICSE '16)*. ACM, New York, NY, USA, 679–690. https://doi.org/10.1145/2884781.2884831
[42] Jean Melo, Fabricio Batista Narcizo, Dan Witzner Hansen, Claus Brabrand, and Andrzej Wasowski. 2017. Variability Through the Eyes of the Programmer. In *Proceedings of the 25th International Conference on Program Comprehension (ICPC '17)*. IEEE Press, Piscataway, NJ, USA, 34–44. https://doi.org/10.1109/ICPC.2017.34
[43] Marcílio Mendonça, Andrzej Wasowski, and Krzysztof Czarnecki. 2009. SAT-based analysis of feature models is easy. In *Software Product Lines, 13th International Conference, SPLC 2009, San Francisco, California, USA, August 24-28, 2009, Proceedings (ACM International Conference Proceeding Series)*, Dirk Muthig and John D. McGregor (Eds.), Vol. 446. ACM, 231–240. https://doi.org/10.1145/1753235.1753267
[44] Sarah Nadi, Thorsten Berger, Christian Kästner, and Krzysztof Czarnecki. 2014. Mining Configuration Constraints: Static Analyses and Empirical Results. In *36th International Conference on Software Engineering (ICSE'14)*.
[45] Sarah Nadi, Christian Dietrich, Reinhard Tartler, Richard C. Holt, and Daniel Lohmann. 2013. Linux variability anomalies: what causes them and how do they get fixed?. In *MSR*, Thomas Zimmermann, Massimiliano Di Penta, and Sunghun Kim (Eds.). IEEE / ACM.
[46] Yoann Padioleau, Julia L. Lawall, and Gilles Muller. 2006. Understanding Collateral Evolution in Linux Device Drivers. In *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006 (EuroSys '06)*. ACM, New York, NY, USA.
[47] Nicolas Palix, Gaël Thomas, Suman Saha, Christophe Calvès, Julia Lawall, and Gilles Muller. 2011. Faults in Linux: Ten Years Later. *SIGARCH Comput. Archit. News* 39, 1 (March 2011), 305–318. https://doi.org/10.1145/1961295.1950401
[48] Leonardo Passos, Leopoldo Teixeira, Dintzner Nicolas, Sven Apel, Andrzej Wasowski, Krzysztof Czarnecki, Paulo Borba, and Jianmei Guo. 2015. Coevolution of Variability Models and Related Software Artifacts: A Fresh Look at Evolution Patterns in the Linux Kernel. *Empirical Software Engineering, Springer* (To appear 2015). https://doi.org/10.1007/s10664-015-9364-x
[49] H. Post and C. Sinz. 2008. Configuration Lifting: Verification meets Software Configuration. In *Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering (ASE'08)*. IEEE Computer Society, L´Aquila, Italy.
[50] Jiri Slaby, Jan Strejček, and Marek Trtík. 2013. ClabureDB: Classified Bug-Reports Database. In *Verification, Model Checking, and Abstract Interpretation*, Roberto Giacobazzi, Josh Berdine, and Isabella Mastroeni (Eds.). Lecture Notes in Computer Science, Vol. 7737. Springer Berlin Heidelberg.
[51] Ştefan Stănciulescu, Sandro Schulze, and Andrzej Wąsowski. 2015. Forked and Integrated Variants in an Open-Source Firmware Project. In *31st International Conference on Software Maintenance and Evolution (ICSME'15)*.

[52] Reinhard Tartler. 2011. Finding and burying Configuration Defects in Linux with the undertaker. (2011). http://www4.informatik.uni-erlangen.de/Publications/2011/plumbers-presentation-tartler.pdf

[53] Reinhard Tartler, Christian Dietrich, Julio Sincero, Wolfgang Schröder-Preikschat, and Daniel Lohmann. 2014. Static Analysis of Variability in System Software: The 90, 000 #ifdefs Issue. In *2014 USENIX Annual Technical Conference, USENIX ATC '14, Philadelphia, PA, USA, June 19-20, 2014.*, Garth Gibson and Nickolai Zeldovich (Eds.). USENIX Association, 421–432. https://www.usenix.org/conference/atc14/technical-sessions/presentation/tartler

[54] Reinhard Tartler, Daniel Lohmann, Christian Dietrich, Christoph Egger, and Julio Sincero. 2011. Configuration coverage in the analysis of large-scale system software. *Operating Systems Review* 45, 3 (2011), 10–14. https://doi.org/10.1145/2094091.2094095

[55] The Institute of Electrical and Eletronics Engineers. 1990. IEEE Standard Glossary of Software Engineering Terminology. IEEE Standard. (1990).

[56] Thomas Thüm, Sven Apel, Christian Kästner, Ina Schaefer, and Gunter Saake. 2014. A Classification and Survey of Analysis Strategies for Software Product Lines. *ACM Comput. Surv.* 47, 1, Article 6 (June 2014), 45 pages. https://doi.org/10.1145/2580950

[57] Yuan Tian, Julia Lawall, and David Lo. 2012. Identifying Linux Bug Fixing Patches. In *Proceedings of the 2012 International Conference on Software Engineering (ICSE 2012)*. IEEE Press, Piscataway, NJ, USA.

[58] David Wagner, Jeffrey S. Foster, Eric A. Brewer, and Alexander Aiken. 2000. A First Step Towards Automated Detection of Buffer Overrun Vulnerabilities. In *NDSS*. The Internet Society.

[59] Zuoning Yin, Xiao Ma, Jing Zheng, Yuanyuan Zhou, Lakshmi N. Bairavasundaram, and Shankar Pasupathy. 2011. 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)*. ACM, New York, NY, USA.