Linux Variability Anomalies: What Causes Them and How Do They Get Fixed?

Sarah Nadi*, Christian Dietrich†, Reinhard Tartler†, Richard C. Holt*, Daniel Lohmann†
* David R. Cheriton School of Computer Science
University of Waterloo, Canada
{snadi, holt}@uwaterloo.ca
† Friedrich-Alexander University Erlangen-Nuremberg, Germany
{dietrich, tartler, lohmann}@cs.fau.de

***Abstract*—The Linux kernel is one of the largest configurable open source software systems implementing static variability. In Linux, variability is scattered over three different artifacts: source code files, KCONFIG files, and Makefiles. Previous work detected inconsistencies between these artifacts that led to anomalies in the intended variability of Linux. We call these variability anomalies. However, there has been no work done to analyze how these variability anomalies are introduced in the first place, and how they get fixed. In this work, we provide an analysis of the causes and fixes of variability anomalies in Linux. We first perform an exploratory case study that uses an existing set of patches which solve variability anomalies to identify patterns for their causes. The observations we make from this dataset allow us to develop four research questions which we then answer in a confirmatory case study on the scope of the whole Linux kernel. We show that variability anomalies exist for several releases in the kernel before they get fixed, and that contrary to our initial suspicion, typos in feature names do not commonly cause these anomalies. Our results show that variability anomalies are often introduced through incomplete patches that change KCONFIG definitions without properly propagating these changes to the rest of the system. Anomalies are then commonly fixed through changes to the code rather than to KCONFIG files.**

***Index Terms*—Software Variability, Variability Anomalies, Linux, Mining Software Repositories, GIT**

I. INTRODUCTION

Software variability involves designing software to be configurable at build time according to the user's selection. One of the largest open source examples of static software variability is the Linux kernel which has over 10,000 configurable features. This makes it an interesting subject for variability research. Previous work (e.g., [1], [2], [3]) analyzed the variability constraints in Linux's feature model (KCONFIG files), source code files and build system (KBUILD). Our previous work [4], [5] showed that these constraints are not always consistent, and that *variability anomalies* arise from them, often leading to dead code or software bugs. Using tools such as UNDERTAKER [4], we detected anomalies in the form of *dead* and *undead* CPP (C preprocessor) guarded code blocks. Dead blocks are those intended to conditionally compile according to some feature(s), but never get compiled in any variant of the software. Similarly, undead blocks are intended to conditionally compile, but end up as part of every variant.

Such previous work confirmed that variability anomalies exist, and that developers eventually fix many of them which indicates that it would be beneficial to avoid them from the beginning. However, in order to avoid these anomalies, we need to analyze how they are introduced in the first place and how they later get fixed. This paper addresses this gap while using Linux as a case study.

To determine how variability anomalies get introduced, we first perform an *exploratory case study* [6] based on an existing set of 106 patches that fix variability anomalies. This set of patches has been submitted to Linux developers by some of the authors of this paper as part of their previous work [4]. The patches received considerable feedback, with over 50% of them being accepted. As a result of studying the responses of developers to these patches, we are able to recognize common problems causing these anomalies. This allowed us to develop the following four research questions:

RQ1: Are misspellings a common cause of variability anomalies?
RQ2: Are incomplete KCONFIG patches a common cause of variability anomalies?
RQ3: How are variability anomalies fixed?
RQ4: How long do variability anomalies remain unfixed?

To answer these questions, we perform a *confirmatory case study* [6]. By mining the patches in Linux's GIT repository and mapping them to affected variability anomalies, we test if the observations we make in our exploratory dataset can be applied to the whole kernel. Our results show that the occurrence of misspellings is not very common. We confirm that incomplete patches changing KCONFIG files without properly propagating the change commonly cause of variability anomalies. We show that these anomalies do not get fixed by KCONFIG changes, but rather by later changing the source code. We find that these anomalies stay in Linux for an average of 6 releases before gettin fixed. This suggests that determining potential causes of anomalies as soon as possible can save developers time in the future. The contributions of this paper are as follows.

1) Qualitative analysis of previous patches submitted to the Linux mailing list.
2) Heuristics to identify potential causes and fixes of variability anomalies in Linux.
3) Quantitative analysis of causes and fixes of variability anomalies in Linux.

4) Empirical validation that incomplete patches often cause variability anomalies and that CPP patches commonly fix these anomalies.

The rest of this paper is organized as follows. Section II provides background information about concepts and tools used in this work. Section III presents our exploratory case study, and Section IV describes the procedure we follow in our confirmatory case study. Section V presents the results of our confirmatory case study, and Section VII raises possible threats to the validity of this work. Section VIII discusses previous research related to our work, and Section IX summarizes our results, and sketches avenues of future exploration.

II. Background

A. Variability in the Linux Kernel

Variability in the Linux kernel is distributed across three separate types of artifacts: KCONFIG files, KBUILD files (Makefiles), and source code files. KCONFIG files declare all configuration features supported by the Linux kernel, as well as their interdependencies. KBUILD is a sophisticated build system implemented in MAKE. Linux source code consists of C header and implementation files, as well as other scripts.

KCONFIG features are the basis for variability implementation in Linux. When configuring an instance of the kernel, users (such as package maintainers or advanced end users) choose the features they want to enable. The features are then referenced within KBUILD or the source code. In KBUILD, configuration features control the compilation of whole source files. In the source code files, configuration features are used in CPP (C preprocessor) guarded code blocks to control conditional compilation. Thus, although there are three different places contributing to Linux's variability, they interact together and depend on each other, and should therefore be kept consistent. More details about variability implementation in Linux can be found in previous work [4], [5].

B. UNDERTAKER*: A Linux Anomaly Detector*

UNDERTAKER [4] is a tool developed at the Friedrich-Alexander University of Erlangen-Nuremberg to perform cross-checking of variability constraints in the three kinds of variability artifacts in Linux. Figure 1 illustrates how it works. UNDERTAKER extracts logical constraints, and transcribes them in the appropriate boolean formulas. Non-boolean constraints are outside the scope of this paper. It then uses a satisfiability (SAT) solver to detect conflicts that reveal variability anomalies in the form of *dead* and *undead* CPP guarded code blocks. A dead code block is guarded by a CPP condition that contains a contradiction (can never be satisfied) and so the code block is never compiled. Conversely, the condition of an undead code block is a tautology (is always satisfied) and the block is always included in the compiled binary.

There are several types of anomalies discovered by UNDERTAKER which are explained in details by Tartler et al. [4]. We discuss two of these categories here: *logical anomalies* and *referential anomalies*. If the logical constraints in the boolean formula are contradictory (e.g., specifying that some

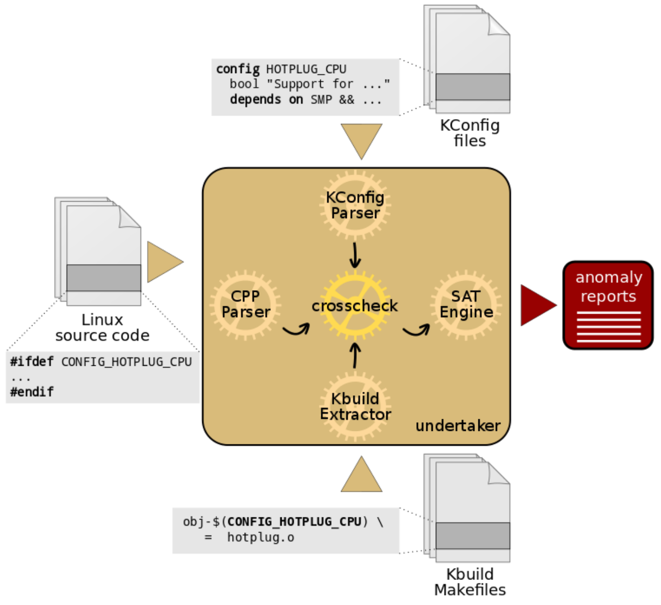


Fig. 1. Operation mode of UNDERTAKER tool. After extracting the constraints from the different variability artifacts, UNDERTAKER combines them and instruments a SAT solver to detect variability anomalies.

feature is defined and not defined at the same time), a *logical anomaly* is detected. However, some features that appear in these boolean formulas are undefined (i.e., do not have a definition in KCONFIG), and can therefore never be selected by the user. These undefined features are grounded to `false` in the boolean formula. If the formula is not satisfiable afterwards, a *referential anomaly* is detected. We refer to these undefined features as *missing*.

In this paper, we focus on referential anomalies as they represent a large percentage of variability anomalies. For example, in Linux release 3.6, UNDERTAKER detects 1,077 anomalies. Out of these, 420 (39%) are referential anomalies.

As a simple example of a referential anomaly, consider the following code:

```
#ifdef CONFIG_USB_SUPPORT
  // Block B1
#endif
```

Let us look at the first code block B1 between the #IFDEF and the #ELSE CPP statements. This block is guarded by the feature `CONFIG_USB_SUPPORT`, so it will not be compiled unless this feature is selected. Assume there is no definition for feature `CONFIG_USB_SUPPORT` in the KCONFIG files, which means that this feature is missing causing the code block to be dead. UNDERTAKER detects this anomaly automatically by using a SAT solver with the boolean formula for block B1:

$$(\mathrm{B1} \leftrightarrow \mathrm{CONFIG\_USB\_SUPPORT}) \\ \wedge \ (\neg\ \mathrm{CONFIG\_USB\_SUPPORT})$$

Based on the result of the SAT checker, UNDERTAKER generates an anomaly report (see Figure 1) which contains the boolean formula as well as the missing features. For brevity, we use the terms *anomaly* or *variability anomaly* throughout the rest of the paper to refer to *referential anomalies*.

C. The Linux Development Process

Changes to the Linux kernel happen through a strict review process. A *change* may be adding a new functionality as well as modifying, correcting, or removing an existing one. Changes are first proposed in the form of a GIT *commit* on focused mailing lists read by the relevant Linux subsystem experts. These changes are then reviewed by expert developers, and approved changes are then committed into the focused subsystem repositories under their control.

Each GIT commit contains information such as the author and date of the commit, a summary of the problem, a detailed description of the commit, as well as the *patch* applied. The patch contains the textual change to the modified files in the so called *unified diff* format[1]. The modified files can be documentation files, KCONFIG files, source code files, or Makefiles. Since each *commit* implements some *change* and has an associated *patch* transcribing this change, we use the three terms interchangeably throughout the paper for simplicity.

In this paper, we focus on KCONFIG and CPP patches, and define them as follows. KCONFIG patches are those which modify a feature definition in KCONFIG. This is despite the fact that the patch may also be modifying other types of files. CPP patches are those which modify the CPP condition in some source file. The commit patch is displayed in unified diff format where lines removed are prefixed with '–', while lines added are prefixed with '+'. For example, the following snippet shows an example of a patch renaming a feature in KCONFIG where the line containing the old feature name is removed, and the line containing the new feature name is added.

```
-config SPI_BFIN
+config SPI_BFIN5XX
  tristate "SPI controller driver ..."
```

As part of our work, we analyze Linux's history to determine if a patch causes or fixes a variability anomaly. Linux's described organizational structure ensures that all code and corresponding descriptions of changes in the master repository have been reviewed by at least two experts. The case studies in this paper can therefore reliably identify focused, well-documented changes in a large-scale, expertly controlled open-source project.

III. Exploratory Case Study

A. Description of Dataset

Previous work by some of the authors of this paper [4] has used UNDERTAKER version 1.1 to detect variability anomalies in the Linux kernel. As part of that work, they randomly chose 337 referential variability anomalies from those they detected, and manually created patches to the Linux kernel to fix these anomalies. This resulted in 106 patches submitted to the Linux mailing lists to solve these 337 anomalies (some patches fixed more than one anomaly). These patches were submitted for anomalies detected in Linux v2.6.35 or earlier.

[1]A standard format for interchanging code changes that is understood by the UNIX patch(1) tool.

TABLE I
Categorization of 106 submitted patches according to the fix they were suggesting.

Type	Count	(%)
Remove Dead Code	95	(90%)
Rename Feature Used in Code	6	(6%)
Remove Dead Code & Dead KCONFIG Features	2	(2%)
Remove Redundant Checks	2	(2%)
Remove Dead Code & Edit KCONFIG Dependencies	1	(1%)
Σ Total	106	(100%)

In their work, Tartler et al. [4] did not perform an in depth analysis of the developers' responses they received or the causes behind these anomalies. In this work, we make use of the unanalyzed data they collected to determine what causes variability anomalies, and how developers fix them. The dataset consists of the actual patches, as well as the email conversations that followed in response to these patches.

To explore the dataset, we first determine the effect of each submitted patch. Table I provides a categorization of the fix proposed by these patches. All patches in this data set fix referential anomalies. The majority of these patches (90%) remove dead code caused by the missing features. Patches that are removing dead code are important since they are removing bad code smells [7], and tend to make the code easier to read and more maintainable. About 6% of the patches rename the missing feature used in the CPP condition in the code to one that is defined in KCONFIG. Two patches (2%) remove dead code as well as dead features from KCONFIG. The dead features depend on an undefined feature, and could therefore never be selected. 2% remove #IFDEF checks either because it is a redundant check (i.e., same condition is checked twice) or because the guarding feature is not defined, but the code inside the block is needed. One patch (1%) removes a dead block that depends on a missing feature while also removing this missing feature from dependency clauses in KCONFIG.

Table II shows the *status* of the submitted patches. We studied the response of developers to each submitted patch email to determine the status of the patch. Most of the 106 patches (51%) were accepted by developers as is. There were 27 patches acknowledged as dead/undead code, but the patch was not applied for one of three reasons: 1) a fix is already being prepared for this problem, either in terms of a scheduled merge or a patch under progress (12 patches), 2) developers want a different fix than the one originally provided in the patch (11 patches), and 3) developers would like to keep the code block as it is because it is used in out of tree development or for reference purposes (4 patches). There were also 21 patches (20%) that received no reply. Only a few patches (4%) were rejected. We classify a patch as rejected if developers do not acknowledge that a problem exists (i.e., that the code is actually dead or undead). This happens in rare cases where the configuration feature(s) used are not defined in KCONFIG, but are rather set by hand in the code.

TABLE II
CATEGORIZATION OF THE STATUS OF SUBMITTED PATCHES ACCORDING TO DEVELOPERS' RESPONSES.

Status	Count	(%)
Accepted	54	(51%)
Acknowledged	27	(25%)
Under progress (12)		
Different fix suggested (11)		
Keep code (4)		
No Reply	21	(20%)
Rejected	4	(4%)
Σ Total	106	(100%)

B. Observations about Exploratory Dataset

From studying the set of 106 patches and the responses of the developers to each patch, we make two main observations.

Feature Names. There are 6 patches which are changing the name of the feature being used in the CPP code. Out of these patches, four were accepted. Additionally, from the set of patches for which developers suggested a different fix or for which they already had a fix for, there are 9 patches that originally removed dead code because of missing features, but developers suggested leaving the code in, and guarding it with a different feature that is defined in KCONFIG. This indicates that this CPP code block is not useless, but has been guarded by an undefined feature causing it to be dead. In four of these cases, we could tell from the developer's comments and the name of the suggested feature that the undefined features being replaced were caused by a misspelling or typo such as, using `CONFIG_CPU_S3C24XX` instead of `CONFIG_CPU_S3C244X` (i.e., typo is putting an X instead of the 4). For simplicity, we use the terms *misspelling* and *typo* interchangeably throughout this paper.

Incomplete Patches. In the comments of one of the patches for which developers suggested using a different feature in the CPP condition, it seems that the missing feature got renamed in KCONFIG, but developers forgot to rename it in the code. The developer's responses to three other accepted patches removing dead code also suggest that the missing features were retired in previous patches, but the code was not updated to reflect that. These observations led us to suspect that *incomplete* patches may be a common cause for variability anomalies. We use the term incomplete to indicate that the change was not completely propagated throughout the system.

To confirm that, we perform an in-depth analysis of 15 patches for which a different patch was suggested, or for which the original patch was renaming the feature being used. We manually study the history of each anomaly to understand how it got introduced. We find that 8 patches fix anomalies that have existed since the related code block has been introduced (i.e., code was dead since inception). On the other hand, 7 patches fix anomalies caused by previous incomplete patches.

We provide an example for such a case. One of these anomalies is a code block which is dead because feature `CONFIG_MTD_NAND_AT91_BUSWIDTH_16` is not defined in KCONFIG. After some investigation, we find that there is a previous patch that renames this feature to `CONFIG_MTD_NAND_`**`ATMEL`**`_BUSWIDTH_16` in KCONFIG, but does not rename it in the CPP condition resulting in the code block being dead because it uses an undefined feature. Although we cannot conclude that incomplete patches are a major source of variability anomalies from this small data sample, the data does provide indication for that, and that this should be further examined in the whole Linux kernel.

C. Research Questions

Based on the patterns we observe in the dataset described above, we develop four research questions about how Linux variability anomalies are introduced and fixed.

Our first research question is based on our first observation that leads us to conjecture that misspellings are a cause of missing features that in turn cause referential anomalies.

RQ1: *Are misspellings a common cause of variability anomalies?*

Our second research question is based on our observation that several anomalies are caused by previous incomplete patches. Specifically, a patch renames or removes a feature in KCONFIG without renaming/removing all its uses in the rest of the kernel. We call these *incomplete* KCONFIG *patches.*

RQ2: *Are incomplete* KCONFIG *patches a common cause of variability anomalies?*

Our observations from the exploratory dataset are mainly concerned with what causes the anomalies. However, it is also important to know how they eventually get fixed. To explore this, we need to analyze how long anomalies last in the Linux, and how they get fixed. We therefore raise the following additional questions.

RQ3: *How are variability anomalies fixed?*

RQ4: *How long do variability anomalies remain unfixed in Linux?*

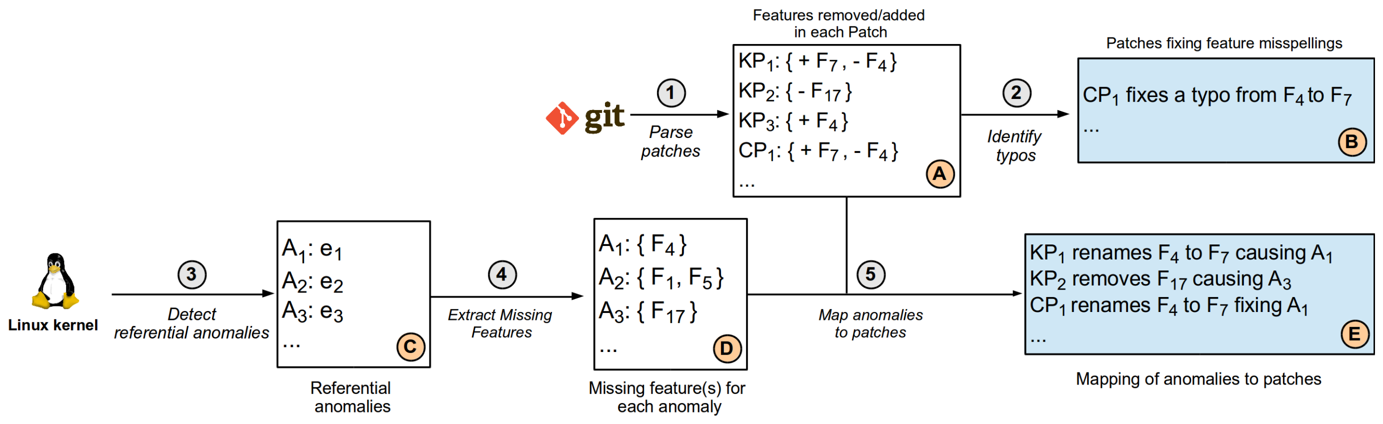
IV. CONFIRMATORY CASE STUDY

The goal of our confirmatory case study is to answer the four research questions from Section III. In this section, we explain the procedure we use to answer these research questions. We describe the steps we perform to find typos as well as mappings from anomalies to patches to identify potential causes and fixes as illustrated in Figure 2. We analyze the variability anomalies in 10 recent releases of the Linux kernel (v2.6.37–v3.6) which spans a period of almost 1 year and nine months.

Our analysis is mainly implemented through several Python scripts which we run on a machine with two quad-core Intel Xeon 2.67GHz CPUs and 16GB RAM.

Step 1: Extract and parse patches

Step 1 identifies and analyzes the patches stored in the GIT repository that are of interest to our analysis. These are KCONFIG patches and CPP patches (explained in Section II-C).



Features removed/added in each Patch
KP1: { + F7 , - F4 }
KP2: { - F17 }
KP3: { + F4 }
CP1: { + F7 , - F4 }
...
A
git
1
Parse patches
2
Identify typos
Patches fixing feature misspellings
CP1 fixes a typo from F4 to F7
...
B
Linux kernel
3
Detect referential anomalies
A1: e1
A2: e2
A3: e3
...
C
Referential anomalies
4
Extract Missing Features
A1: { F4 }
A2: { F1, F5 }
A3: { F17 }
...
D
Missing feature(s) for each anomaly
5
Map anomalies to patches
KP1 renames F4 to F7 causing A1
KP2 removes F17 causing A3
CP1 renames F4 to F7 fixing A1
...
E
Mapping of anomalies to patches

Fig. 2. An overview of our analysis during the confirmatory case study to identify potential causes and fixes for referential variability anomalies. KP_n: KCONFIG patches, CP_n: CPP patches, A_n: Anomalies, e_n: Boolean expressions, F_n: KCONFIG features.

Box A in Figure 2 shows a representation of our extracted patches where KCONFIG patches are denoted by KP_n and CPP patches by CP_n. For each patch, we identify the features added or removed found through the + and – notation described in Section II-C. For example, the notation $KP_1\{+F_7, -F_4\}$ in Box A means that KP_1 removes Feature F_4 and adds Feature F_7. Similarly, KP_2 removes F_{17} and KP_3 adds F_4 while CP_1 adds F_7 and removes F_4 from some CPP conditions.

It is straightforward to parse KCONFIG patches. This is done by locating feature declarations that are inserted or deleted and transcribing them into plus and minus notation (see Box A). It is more complicated to parse CPP patches. This is done by searching for CPP guarded code blocks (blocks surrounded by #IFDEF, #IFNDEF, etc.) and identifying each feature F_i used in the CPP condition (the CPP guard). If patch CP_k deletes the line containing the condition, then F_i is also considered to be deleted, so we produce output such as $CP_k : -F_i$. Conversely, if CP_k adds the condition, we produce output such as $CP_k : +F_i$. For CPP patches, we also record the source file in which the CPP condition was added or removed.

If a patch removes one feature F_i and then adds another F_j in the next step, we consider that F_j *renames* F_i. For example, patch KP_1 in Box A renames F_4 to F_7.

The extraction and parsing of patches in Step 1 is performed only once, on the most recent kernel release (v3.7[2]), to extract the whole history of the ten Linux releases examined (v2.6.37–v3.6). We extract the KCONFIG patches and CPP patches separately. It takes approximately 39 minutes to extract KCONFIG patches from GIT, and 44 minutes to extract the CPP patches. The results are saved to be queried in Steps 2 and 5.

Step 2: Identify misspelled features

Step 2 determines if typos (misspelled features) are a common cause of variability anomalies (RQ1). It does this by locating CPP patches that rename the feature used in the CPP condition. Based on these renames, we develop heuristics to automatically compare the names of the old and new features to determine if the change is correcting a typo.

[2]v3.7 is used to ensure that any fixes that happen to anomalies found in v3.6 are also caught.

We consider that the renaming is correcting a misspelling if the first name of the pair is within one or two edit distances from the second name (i.e., a difference of one or two characters) or if the first name is a permutation of the words separated by underscores in the second name (e.g., `CONFIG_USB_SUPPORT` vs `CONFIG_SUPPORT_USB`). This automatic classification provides us with a set of CPP patches that could potentially be correcting misspelled features. For example, consider Patch CP_1 in Box A; if feature F_4 is classified as an apparent typo of feature F_7, then we classify CP_1 as correcting a misspelling as shown in Box B.

We then manually verify each of these identified patches by looking at the developer's commit messages to judge if the change was actually correcting a spelling mistake or not. This manual verification is necessary to avoid false positives because there are features in Linux which have similar names (e.g., `X86_32` and `X86_64`), but are implementing different functionalities, and so a replacement of one feature with another one may be an intentional logic change and not a typo. We choose to design our analysis this way and not to match missing features in the anomalies to possible misspellings in all defined KCONFIG features, because there would be no way to verify if the identified features are indeed typos or not.

Since this step, Step 2, only depends on the extracted CPP patches, we perform it only once after extracting Linux's history. This takes 1hr 38 minutes to run. The performance bottleneck here is the algorithms used to detect similar words.

The following steps (3, 4 and 5) are then performed for each kernel release examined to answer RQ2 and RQ3 dealing with patches causing and fixing anomalies.

Step 3: Detect referential anomalies using UNDERTAKER

In Step 3, we run the latest version of UNDERTAKER, 1.3, to detect referential variability anomalies in the ten releases of the Linux kernel we examine. These referential anomalies are shown in Box C in Figure 2 where each anomaly A_k has a boolean expression e_k which is the boolean formula that was not satisfied (see Section II-B). Detecting anomalies on the whole Linux kernel using UNDERTAKER takes approximately 45 min for each kernel release using 4 parallel threads.

Step 4: Extract missing features

In Step 4, we analyze the boolean formula e_k for each referential anomaly extracted to automatically identify the missing feature causing the anomaly. This allows us to generate a mapping from anomalies to missing features. For example, in Section II-B, we identify `CONFIG_USB_GADGET_AT91` as the missing feature. Box D in Figure 2 shows such mappings. This step takes about 3 seconds for each release.

Step 5: Map anomalies to patches

Step 5 correlates anomalies with their potential causes and fixes; see Box E in Figure 2. Since the Linux kernel has many patches (the GIT repository contains over 300,000 commits), we need to develop heuristics to automatically identify these causes and fixes, which will now be described.

Our heuristics identify two causes of anomalies: *renaming* and *removal* of KCONFIG features without reflecting these changes in the code. The first two lines of Box E in Figure 2 illustrate these two cases. In Box D, Anomaly A_1 is due to the missing feature F_4. Patch KP_1 renames feature F_4 to F_7 in KCONFIG, thus removing the definition of F_4 from KCONFIG, and causing anomaly A_1. Similarly, anomaly A_3 in Box D is due to the undefined feature F_{17}. Patch KP_2 removes F_{17} from KCONFIG making F_{17} a missing feature and causing anomaly A_3. In more general terms, there are two kinds of causes of an anomaly A_k that is due to missing feature F_j:

1) Patch KP_i *removes* feature F_j from KCONFIG.
2) Patch KP_i *renames* feature F_j in KCONFIG.

In both cases, patch KP_i must occur *before* anomaly A_k. If more than one matched patch occurs before the anomaly, we choose the patch closest to the date of the occurrence of A_k.

We also identify four ways that patches can fix anomalies. One of these is illustrated by the third line in Box E in Figure 2 which specifies that patch CP_1 renames feature F_4 to F_7 in the CPP condition thus fixing anomaly A_1. A_1 is caused by the undefined feature F_4. CP_1 fixes this by using feature F_7 instead of F_4 in the CPP condition. Recall that the incomplete patch KP_1 renames F_4 to F_7 in KCONFIG without reflecting the change in the code. Thus, CP_1 completes this rename in the code by using F_7 instead of F_4 which fixes anomaly A_1.

Specifically, given an anomaly A_k caused by a missing feature, F_j, we identify four cases where a patch fixes this anomaly as follows (the discussed example is the fourth kind):

1) Patch KP_i *adds* F_j to KCONFIG.
2) Patch KP_i *renames* another feature in KCONFIG to F_j.
3) Patch CP_i *removes* the CPP condition containing F_j.
4) Patch CP_i *renames* F_j in the CPP condition.

All types of fixes essentially aim to ensure that the features used in the CPP conditions have a corresponding definition in KCONFIG. In each of the four kinds of fixes, the matched patch must occur *after* the anomaly A_k. If more than one such fixing patch occurs after the anomaly, we choose the patch closest to the date of the anomaly. Mapping anomalies to KCONFIG patches takes approximately 13 seconds for each release, and mapping anomalies to CPP patches takes approximately 47 seconds for each release.

V. RESULTS OF CONFIRMATORY CASE STUDY

We follow the procedure explained in the previous section (see Figure 2) to analyze the variability anomalies in releases v2.6.37 to v3.6. We apply Step 1 on release 3.7 to extract all the Linux history until its latest release. We extract 10,263 KCONFIG patches and 25,410 CPP patches from the GIT repository. We report the results of our analysis in this section. We structure our results to answer the four research questions, and then provide interpretation of our findings.

A. RQ1: Are Misspellings a Common Cause of Variability Anomalies?

Out of the 25,410 extracted CPP patches, only 1,412 patches rename features in CPP conditions (i.e., patch changes the feature being used in the condition). From these patches, we use our spelling checker heuristics (Section IV, Step 2) to automatically find 203 patches (14%) where the replacement feature seems to be correcting a misspelling of the original feature. We manually verify all 203 patches by checking developers' commit messages to judge if this patch is indeed correcting a misspelling. We are able to confirm that 54 out of the 203 patches (27%) are indeed correcting misspellings (the high number of false positives are due to similar features like `X86_32` and `X86_64` as explained in Step 2 of Section IV). This means that only 4% (54 out of 1,412) of CPP patches renaming features are dealing with misspelled features.

Finding 1: *Typos are not a common cause of variability anomalies. Only 4% of* CPP *patches changing the feature used in the* CPP *condition are correcting misspellings.*

B. RQ2: Are Incomplete KCONFIG *Patches a Common Cause of Variability Anomalies?*

Table III summarizes the results for the mapped anomalies in each release studied. In each release, the table shows the number of anomalies mapped to causing and fixing patches. The second column of the table shows the number of referential anomalies detected by UNDERTAKER in each release. The third column shows the number of anomalies which we were able to automatically map to a causing KCONFIG patch in the GIT history. That is, a patch that occurs before the date of this release removes or renames the missing feature in KCONFIG without reflecting this change in the anomalous file. Since a referential anomaly can be due to more than one missing feature, the same anomaly may be matched to several historic patches based on the different missing features. However, we only count the unique number of anomalies for which we could find a causing patch. We note that an anomaly may span multiple releases of the Linux kernel. Since we count the number of matches in each release, and not throughout all release, we avoid multiple countings of the same anomaly.

TABLE III
NUMBER OF REFERENTIAL ANOMALIES IN EACH RELEASE THAT ARE CAUSED BY INCOMPLETE KCONFIG PATCHES AS WELL AS THOSE FIXED BY KCONFIG AND CPP PATCHES. PERCENTAGES ARE SHOWN IN PARENTHESIS.

Release	Referential Anomalies	Anomalies caused by incomplete KCONFIG patches	Anomalies fixed by	
			KCONFIG Patches	CPP Patches
2.6.37	706	56 (8%)	22 (3%)	383 (54%)
2.6.38	688	62 (9%)	21 (3%)	354 (51%)
2.6.39	658	61 (9%)	28 (4%)	317 (48%)
3.0	618	67 (11%)	12 (2%)	193 (31%)
3.1	528	96 (18%)	12 (2%)	129 (24%)
3.2	478	74 (15%)	12 (3%)	99 (21%)
3.3	490	73 (15%)	42 (9%)	84 (17%)
3.4	485	86 (18%)	4 (1%)	39 (8%)
3.5	425	87 (20%)	5 (1%)	21 (5%)
3.6	420	83 (20%)	0 (0%)	3 (1%)
	Mean	75 (14%)	16 (3%)	162 (26%)
	Median	74 (15%)	12 (3%)	114 (23%)

Table III shows that a mean of 14% of the referential anomalies in each release are caused by incomplete KCONFIG patches with one release having values as high as 20%. We could not automatically map the rest of the anomalies to a causing KCONFIG patch. A quick manual analysis of these unmapped anomalies suggests that many of these code blocks have been anomalous since their inception in the code which suggests that the code block has always been dead/undead.

Column 3 in Table III also shows that the percentage of anomalies with matched causing patches is higher in more recent releases. This is because as we examine more recent releases, we have more previous history to analyze, and thus a higher chance to find patches that occur before the current release that can be identified as causes. Although we cannot conclude that incomplete KCONFIG patches are the only cause of referential anomalies, our results suggest that they are a common cause.

Finding 2: *Incomplete* KCONFIG *patches often cause referential anomalies. An average of 14% of referential anomalies are caused by changes to* KCONFIG *that are not completely propagated to the source code.*

C. RQ3: How are Variability Anomalies Fixed?

We now study patches that fix referential anomalies in order to answer RQ3. A referential anomaly is caused by a feature that appears in the boolean formula of the code block, but has no definition in KCONFIG. Therefore, such an anomaly would either be fixed by (1) adding the feature's definition in the KCONFIG files (either by adding a new feature or renaming another feature), or (2) removing that feature from the code block (either by deleting the whole code block, or using another defined feature instead).

With respect to the first possibility, the fourth column in Table III shows the number of anomalies in each release which are fixed by KCONFIG patches. We can see that a very small percentage of referential anomalies (average of 3%) get fixed by future KCONFIG patches. This suggests that although changes to KCONFIG introduce these anomalies, not many of them also get fixed by changes to KCONFIG.

We now look at the second possibility of future CPP patches fixing referential anomalies. The last column in Table III shows the number of anomalies in each release that are fixed by CPP patches. As shown, an average of 26% of the anomalies are fixed by CPP patches. In some releases (2.6.37-2.6.39), these percentages are as high as 48%-54%. When we analyze those fixes, we find that the majority of them remove the dead code block itself or the CPP condition from around undead code blocks. This indicates that these code blocks should have been originally removed in previous patches (i.e., the incomplete ones) since this is indeed how they got fixed later on. On the other hand, a few CPP fixes rename the features used in the code to be consistent with those in KCONFIG.

The last column in Table III shows that the percentage of anomalies fixed by CPP patches is decreasing over time. This is because with more recent releases, there is not much history beyond that release to be able to identify potential fixes. The only exception is the number of KCONFIG patches fixing anomalies (Column 3) in release 3.3. The reason for the high number of matches is due to a patch that renamed `CONFIG_SPI_BFIN` to `CONFIG_SPI_BFIN5XX` in KCONFIG. This simultaneously fixed 31 anomalies that were due to the missing feature `CONFIG_SPI_BFIN5XX` in different files under the `blackfin` architecture in Linux.

Finding 3: *Referential anomalies are commonly (26% of the time) fixed by* CPP *patches.*

D. RQ4: How Long do Variability Anomalies Remain in Linux?

We have identified that referential anomalies are commonly fixed through CPP patches. We now look at how long it usually takes for developers to fix these anomalies. Since the location of a code block may change over time (thus changing the location of the anomaly), we need a method to track the anomaly's location as it changes. We use Herodotos [8] to accomplish that. Herodotos tracks bugs over different versions of a software system by considering the lines added and removed in patches such that it can find the location of a particular code block in a different version of the system. Using Herodotos, we track the referential anomalies detected to determine when they are no longer detected in the system. We identify the version that introduces an anomaly and the version that fixes it.

We find that on average, referential anomalies remain in Linux for 6 releases (approx. 10 months). Since some anomalies are still not fixed in the last release examined, we consider that the minimum lifetime for those anomalies (i.e., 1 release). The standard deviation of the lifetime of an anomaly is 3.

Finding 4: *Referential anomalies remain unfixed in Linux for an average of 6 releases.*

VI. Discussion

A. Interpretation of Our Findings

Finding 4 suggests that many referential anomalies remain unfixed in Linux for an average of 6 releases. A standard deviation of 3 for the anomaly lifetime suggests that there are anomalies that are easier to find and fix than others. Fixed anomalies provide us insight to how Linux developers address such problems. We attribute the low number of corrections that fix misspellings (4%), found in Finding 1, to the strict Linux review process each change has to undergo before its integration. Anomalies not caused by misspellings are harder to catch by developers during their review process since other places such as KCONFIG files may also need to be checked, which explains the higher percentage of anomalies (14%) caused by incomplete KCONFIG changes found in Finding 2.

There are two explanations for Finding 3 which suggests that Linux developers tend to fix referential anomalies on the variability implementation (the source code), and rather seldom on the variability declaration side (KCONFIG). First, changes to the variability declaration occur less often than code additions because they are less often necessary. This is also seen in the small number of KCONFIG patches (10,263) in Linux's repository (i.e., those changing feature definitions in KCONFIG) when compared to the number CPP patches changing the features used in CPP conditions (25,410). Second, changes to KCONFIG have (potentially) wide cross-cutting effects on the Linux code base. Previous work by Eaddy et al. [9] has shown that a high amount of cross-cutting concerns in a software system increases the number of defects. Keeping in mind that Linux is a very large collaborative project, a developer that edits a KCONFIG feature definition potentially changes the behavior of some code that he does not know about, which introduces the anomaly. It, therefore, makes sense that changes to CPP code are later necessary to fix this anomaly, and make it consistent with the KCONFIG change. This makes the understanding of KCONFIG changes, and the required manual code review, much harder than the more focused changes in C source files.

These findings along with the the observation that many of these anomalies have existed since the code was created provide an indication that tools to aid programmers in understanding the mapping from feature declaration to variability implementation are necessary. Running such tools when making changes in Linux can make sure that variability information is kept consistent. However, further investigation into what difficulties developers have in maintaining this consistency are also needed. This can include surveys or interviews of developers to better identify the problems they face with maintaining variability in order to further improve the tools researchers provide them.

B. Beyond Referential Anomalies and Linux

We currently focus on referential variability anomalies since they are the most common types of anomalies, and finding their causes and fixes can be automated. The generated boolean formulas are usually very long and complicated which makes them impossible to study manually. The challenge with studying logical anomalies not caused by missing features (see Section II-B) is that when a boolean formula fails, it is not easy to automatically identify the conflict which caused the failure. Even when such a conflict is identified, it is often difficult to identify the fix needed to remove the conflict. Let us take the following simple formula for a dead block as an example.

$$(\mathrm{B1} \leftrightarrow \mathrm{CONFIG\_X}) \wedge (\mathrm{CONFIG\_X} \rightarrow \mathrm{CONFIG\_Y} \wedge \mathrm{CONFIG\_Z}) \\ \wedge(\mathrm{CONFIG\_Z} \rightarrow \neg\mathrm{CONFIG\_Y} \wedge \mathrm{CONFIG\_W})$$

`B1` is dead because the formula is not satisfiable since `CONFIG_Y` cannot be defined and undefined at the same time. Several changes can be made in order to fix this anomaly. In `CONFIG_X`'s KCONFIG definition, the dependency on `CONFIG_Z` can be removed which will allow the formula to be satisfiable. Alternatively, the dependencies in the KCONFIG definition of `CONFIG_Z` can be changed by either removing the negation of `CONFIG_Y` or removing the dependency on `CONFIG_Y` altogether. Such solutions cannot be automatically captured which is why analyzing logical anomalies requires a lot of manual effort.

A possible solution for this is to follow a technique similar to that proposed by Śliwerski et al. [10] where you can identify the release that fixed the problem (i.e., a release where the anomaly no longer appears in), and then analyze the changes that occurred between these releases. The challenge here is since we are dealing with multiple artifacts, a change that introduces or fixes the anomaly may not necessarily be in the code, but may be in a related KCONFIG or Makefile. Heuristics can be applied to limit the search space a bit. However, more investigation in this direction is needed.

Although our study is limited to Linux, we are convinced our techniques and results can be applied to other systems. Inconsistencies arise from scattered information, and changes that are not properly propagated to all parts of the system contributing to variability. This applies to many software systems. Additionally, several systems use CPP to control variability, and also use KCONFIG as their variability model (e.g., BUSYBOX, BUILDROOT). Since these systems are similarly structured to Linux, it seems likely that the same observations may apply. Additionally, other systems which implement variability differently (e.g., ECOS) may also have inconsistencies caused by incomplete changes since variability information is still divided among more than one place.

VII. Threats to Validity

A. Internal Validity

Mining GIT. Our work relies on mining the GIT repository in Linux. We only analyze the master repository maintained by Linux Torvalds. This ensures that the commits we analyze have been thoroughly reviewed and that we avoid many of the perils of GIT branching discussed by Bird et al. [11].

Mapping Accuracy. Our results rely on the automatic mapping scheme we have developed. Since we focus on investigating our raised research questions, and not on providing any tools to be directly used by Linux developers, we develop conservative heuristics to avoid false positives in our mapping of anomalies to patches. This explains the large number of unmapped anomalies. However, we manually verify many of the detected mappings to confirm they are correct. Since removed and added features in each patch can be accurately identified from the + and – diff notation, we believe our mappings are accurate.

Misspellings. We conclude that misspellings are not a common cause of variability anomalies. This is based on the fixes we analyzed. However, due to the conservative way we designed our analysis, we would miss any anomalies caused by misspellings, but which have not yet been fixed.

Detected Anomalies. Despite the fact that variability anomalies discussed in this paper do not break the code in a direct way, they do lead to unexpected behavior such as not having some functionality even though it is chosen by the user. This is why we opted to use the term *anomaly* to describe this phenomenon, which is similar to the idea of *bad code smells* [7].

B. Construct Validity

The dataset we use to develop our research questions has been created by some of the authors of this paper in their previous work [4]. To avoid researcher expectancies or over familiarity with the data, the first author of the paper, who was not involved in the original work, studied this dataset to develop the research questions.

Since our study focuses on referential anomalies, the fact that the exploratory dataset we examine only contains referential anomalies does not bias our results. We study all 106 patches solving referential anomalies in the exploratory dataset, and thus avoid the need to do any data sampling. The 337 anomalies for which the patches have been created for have been randomly sampled from all detected referential anomalies, and thus, the dataset does not suffer from any sampling bias.

C. External Validity

This work provides a case study of a single software system, Linux, which does not allow us to generalize our results to other software systems. However, Linux is one of the largest and commonly studied open source software systems that support software variability, and has also been previously studied in terms of variability anomalies. We believe that this case study provides a methodology which can be followed to study causes and fixes of variability anomalies in other systems implementing variability through CPP directives and feature selection such as those discussed by Liebig et al. [12] and Spinellis [13] (e.g., Apache, FreeBSD).

VIII. Related Work

A. Variability in Linux

The analysis of variability in Linux is an important research topic due to its large size and the large number of variants it can produce. Variability in the Linux kernel has been studied in terms of its feature model (or variability model) manifested in the KCONFIG files [1], [2]. KCONFIG features then get used in CPP directives in the source code to control source code compilation. Several studies focused on studying variability in the source code [4], [14], and how the source code can be statically parsed to analyze these variability points [15].

The build system (KBUILD) also uses KCONFIG features to control source file compilation. By including the constraints from the source code, KCONFIG, and KBUILD together, previous work has been able to detect variability anomalies in the Linux kernel [5], [16]. Tartler et al. [4] have also provided patches to fix some of these detected variability anomalies. However, to the best of our knowledge, there has not been any comprehensive work done to understand the causes and fixes of these variability anomalies. Studying the origin of these variability anomalies is important in order to provide tools that support more proactive anomaly prevention.

B. Bug-Introducing Changes

It is a common belief that changes to a system often introduce bugs. Our finding that incomplete KCONFIG patches cause variability anomalies aligns with this belief. Identifying *bug-introducing* changes is a well researched topic (e.g., [17], [18], [19], [20]). Most of these techniques rely on first identifying *bug-fixing* changes by looking for keywords such as *fix* or for a bug number in the report. These identified changes are then used to train models that can be used to predict future bug introducing changes. We choose not to follow such techniques for several reasons. First, we are interested in a specific category of problems which are variability anomalies, and so looking at all bug-fixing changes is unnecessary. Second, since there has not been much work in determining the causes and fixes of variability anomalies, we had to first find the patterns we need to look for. We did this by our exploratory study which allowed us to develop criteria for determining causes and fixes. Finally, as pointed out by previous research [21], the automatically identified bug-fix datasets commonly result in unbalanced datasets since they heavily rely on developers including the linkage information between bugs and fixes in their commit messages. Our work is also different from previous research in that we do not only focus on source code changes, but rather we relate changes in one part of the system (KCONFIG) to anomalies in a different part (source code).

IX. Conclusions and Future Work

This paper addresses the gap between detecting variability anomalies in software systems, and understanding how these anomalies occur and get fixed. We use the Linux kernel as a case study, and analyze causes and fixes of referential anomalies (i.e., those caused by undefined KCONFIG features) detected in the kernel. By performing an *exploratory case study* on an existing set of 106 patches which fix previously detected referential variability anomalies, we develop four research questions about the causes and fixes of variability anomalies. We then study several releases of the whole Linux kernel to answer these questions in a *confirmatory case study*.

We find that variability anomalies typically stay in Linux for an average of 6 releases ($\approx$ 10 months) before they get fixed which suggests that detecting these anomalies is not trivial, and that fixing them as soon as they are introduced is important. Our findings show that referential anomalies are often (14% of the time) introduced by incomplete patches which change KCONFIG files without fully reflecting these changes in the corresponding source code. This indicates that automated anomaly detection, such as that provided by UNDERTAKER, should be incorporated into the change process to detect these inconsistencies as soon as the patches are applied. We find that in 26% of the time, these anomalies get fixed by CPP patches that remove the defective code block or rename the undefined feature being used in it. This indicates that KCONFIG changes often have wide cross-cutting effects on the code that are not detected till later, and must be fixed through CPP changes.

The patterns for anomaly causes and fixes we found can help Linux developers avoid such problems in the future. They also allow consistency checking tool designers to automatically identify causes of their detected anomalies, and provide suggestions to fix them. Additional studies to investigate other types of variability anomalies (e.g., logical anomalies) as well as analyzing further causes (e.g., code cloning) are needed. We also hope to analyze systems other than Linux to be able to generalize our results, and find general patterns of causes and fixes of variability anomalies.

ACKNOWLEDGEMENTS

We would like to thank Christoph Egger for his help in configuring and running Herodotos.

REFERENCES

[1] R. Lotufo, S. She, T. Berger, K. Czarnecki, and A. Wasowski, “Evolution of the Linux kernel variability model,” in *SPLC'10: Proceedings of the 14th International Conference on Software Product Lines: Going Beyond*. Springer-Verlag, 2010, pp. 136–150. [Online]. Available: http://portal.acm.org/citation.cfm?id=1885639.1885653

[2] S. She, R. Lotufo, T. Berger, A. Wasowski, and K. Czarnecki, “Variability model of the linux kernel,” in *VaMoS 2010: Proceedings of the 4th International Workshop on Variability Modeling of Software-intensive Systems*, 2010.

[3] T. Berger, S. She, R. Lotufo, K. Czarnecki, and A. Wasowski, “Feature-to-code mapping in two large product lines,” in *SPLC'10: Proceedings of the 14th International Software Product Line Conference. Poster Session*. Springer Berlin/Heidelberg, 2010.

[4] R. Tartler, D. Lohmann, J. Sincero, and W. Schröder-Preikschat, “Feature Consistency in Compile-Time Configurable System Software,” in *Proceedings of the EuroSys 2011 Conference (EuroSys '11)*, G. Heiser and C. Kirsch, Eds., New York, NY, USA, 2011, pp. 47–60. [Online]. Available: http://doi.acm.org/10.1145/1966445.1966451

[5] S. Nadi and R. Holt, “Mining Kbuild to detect variability anomalies in Linux,” in *CSMR '12: Proceedings of the 16th European Conference on Software Maintenance and Reengineering*, March 2012, pp. 107 –116.

[6] S. Easterbrook, J. Singer, M.-A. Storey, and D. Damian, “Selecting empirical methods for software engineering research,” in *Guide to Advanced Empirical Software Engineering*, F. Shull, J. Singer, and D. I. K. Sjøberg, Eds. Springer London, 2008, pp. 285–311. [Online]. Available: http://dx.doi.org/10.1007/978-1-84800-044-5_11

[7] M. Fowler and K. Beck, *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 1999.

[8] N. Palix, J. Lawall, and G. Muller, “Tracking code patterns over multiple software versions with Herodotos,” in *AOSD '10: Proceedings of the 9th International Conference on Aspect-Oriented Software Development*. New York, NY, USA: ACM, 2010, pp. 169–180. [Online]. Available: http://doi.acm.org/10.1145/1739230.1739250

[9] M. Eaddy, T. Zimmermann, K. Sherwood, V. Garg, G. Murphy, N. Nagappan, and A. Aho, “Do crosscutting concerns cause defects?” *IEEE Transactions on Software Engineering*, vol. 34, no. 4, pp. 497 –515, July-Aug. 2008.

[10] J. Śliwerski, T. Zimmermann, and A. Zeller, “When do changes induce fixes?” in *Proceedings of the 2005 international workshop on Mining software repositories*, ser. MSR '05. New York, NY, USA: ACM, 2005, pp. 1–5. [Online]. Available: http://doi.acm.org/10.1145/1082983.1083147

[11] C. Bird, P. Rigby, E. Barr, D. Hamilton, D. German, and P. Devanbu, “The promises and perils of mining git,” in *MSR '09: Proceedings of the 6th IEEE International Working Conference on Mining Software Repositories*, May 2009, pp. 1 –10.

[12] J. Liebig, S. Apel, C. Lengauer, C. Kästner, and M. Schulze, “An analysis of the variability in forty preprocessor-based software product lines,” in *ICSE '10: Proceedings of the 32nd International Conference on Software Engineering*, vol. 1, may 2010, pp. 105 –114.

[13] D. Spinellis, “A tale of four kernels,” in *ICSE '08: Proceedings of the 30th International Conference on Software Engineering*. New York, NY, USA: ACM, 2008, pp. 381–390. [Online]. Available: http://doi.acm.org/10.1145/1368088.1368140

[14] J. Sincero, R. Tartler, D. Lohmann, and W. Schröder-Preikschat, “Efficient extraction and analysis of preprocessor-based variability,” in *GPCE '10: Proceedings of the 9th international conference on Generative programming and component engineering*. ACM, 2010, pp. 33–42.

[15] C. Kästner, P. Giarrusso, T. Rendel, S. Erdweg, K. Ostermann, and T. Berger, “Variability-aware parsing in the presence of lexical macros and conditional compilation,” in *Proceedings of the 2011 ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, 2011.

[16] C. Dietrich, R. Tartler, W. Schröder-Preikschat, and D. Lohmann, “A robust approach for variability extraction from the linux build system,” in *SPLC '12: Proceedings of the 16th International Software Product Line Conference (to appear)*, 2012.

[17] S. Kim, T. Zimmermann, K. Pan, and E. Whitehead, “Automatic identification of bug-introducing changes,” in *ASE '06: Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering*, Sept. 2006, pp. 81 –90.

[18] L. Aversano, L. Cerulo, and C. Del Grosso, “Learning from bug-introducing changes to prevent fault prone code,” in *Ninth international workshop on Principles of software evolution: in conjunction with the 6th ESEC/FSE joint meeting*, ser. IWPSE '07. New York, NY, USA: ACM, 2007, pp. 19–26. [Online]. Available: http://doi.acm.org/10.1145/1294948.1294954

[19] K. Pan, S. Kim, and E. Whitehead, “Toward an understanding of bug fix patterns,” *Empirical Software Engineering*, vol. 14, pp. 286–315, 2009, 10.1007/s10664-008-9077-5. [Online]. Available: http://dx.doi.org/10.1007/s10664-008-9077-5

[20] D. Posnett, A. Hindle, and P. Devanbu, “Got issues? do new features and code improvements affect defects?” in *WCRE '11: Proceedings of the 18th Working Conference on Reverse Engineering*, Oct. 2011, pp. 211 –215.

[21] C. Bird, A. Bachmann, E. Aune, J. Duffy, A. Bernstein, V. Filkov, and P. Devanbu, “Fair and balanced?: bias in bug-fix datasets,” in *ESEC/FSE '09: Proceedings of the the 7th Joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, ser. ESEC/FSE '09. New York, NY, USA: ACM, 2009, pp. 121–130.