IT UNIVERSITY OF COPENHAGEN

Ph.D. Dissertation

Variability Bugs: Program and Programmer Perspective

Jean Carlos de Carvalho Melo

Supervisors:

Claus Brabrand
Andrzej Wąsowski

June 15, 2017

Abstract

Many modern software systems are highly configurable. They embrace variability to increase adaptability and to lower cost. To implement configurable software, developers often use the C preprocessor (CPP), which is a well-known technique, mainly in industry, to deal with variability in code. Although many researchers suggest that preprocessor-based variability amplifies maintenance problems, there is little to no hard evidence on how actually variability affects programs and programmers. Specifically, how does variability affect programmers during maintenance tasks (bug finding in particular)? How much harder is it to debug a program as variability increases? How do developers debug programs with variability? In what ways does variability affect bugs?

In this Ph.D. thesis, I set off to address such issues through different perspectives using empirical research (based on controlled experiments) in order to understand quantitatively and qualitatively the impact of variability on programmers at bug finding and on buggy programs.

From the program (and bug) perspective, the results show that variability is ubiquitous. There appears to be no specific nature of variability bugs that could be exploited. Variability bugs are not confined to any particular type of bug, error-prone feature, or location. In addition to introducing an exponential number of program variants, variability increases the complexity of bugs due to unintended feature interactions, hidden features, combinations of layers (code, mapping, model), many function calls, etc.

From the programmer (and bug-finding) perspective, I find that the time needed for finding bugs increases linearly with variability, while finding bugs in the first place is relatively independent of variability. In fact, most developers correctly identify bugs, yet many fail to identify the exact set of erroneous configurations. I also observe that developers navigate much more between definitions and uses of program objects when interleaved with variability.

Overall, this Ph.D. thesis shows that variability complicates the complexity of bugs and bug finding, but not terribly so. This is positive and consistent with the existence of highly-configurable software systems with hundreds, even thousands, of features, testifying that developers in the trenches are able to deal with variability.

To my mother and grandmother

Acknowledgments

I am grateful to God for this great achievement in my professional career.

I would like to warmly thank my family and especially my wife, Edinez, for their love, support, and encouragement through all these years.

I would like to thank my supervisors, Claus Brabrand and Andrzej Wąsowski, for their time and guidance during the past three years. I have learned so much observing, talking, and writing papers with them.

I would like to thank all my ITU colleagues, especially the members of the ITU SQUARE group for the great discussions on diverse SE topics and for the feedback on my research. Thanks Ahmad Salim Al-Sibahi, Aleksandar Dimovski, Alexandru F. Iosif-Lazăr, Iago Abal, and Ștefan Stănciulescu! Also, I would like to thank Christian Kästner and my colleagues from my stay abroad at Carnegie Mellon University.

Finally, I would like to thank the Brazilian National Council for Scientific and Technological Development (CNPq), for funding this work under the Science without Borders programme.

Contents

Chapter 1

Introduction

1.1 Context

Nowadays, many software systems are highly configurable. They embrace *variability* as a need to adapt their products to meet requirements of various market segments and to extend portability across different hardware platforms, for example. *Highly-configurable systems* include both large industrial product lines [25, 76, 9] and open-source systems. In some cases, such as the Linux kernel, thousands of configuration options (*features*) are used to control the compilation process [11].

Software variability supports the development of such configurable systems, which can be used to build many related software systems (*program variants*), from a common piece of code, by fixing configuration options as either *enabled* or *disabled*. A configuration option controls whether to include or exclude a certain functionality in a program variant. Software variability is a cost-effective way to develop and maintain a variety of related software systems, increasing adaptability and lowering the cost.

To implement configurable systems, a multitude of technologies can be used: object-oriented patterns, aspects, domain-specific languages and code generation, plug-in mechanisms, and so on. Among these, the conditional compilation directives of the C preprocessor (cpp) are one of the *oldest*, the *simplest*, and the *most popular* [36, 60, 50] mechanisms in use, especially in the systems domain. Preprocessor directives, like `#ifdef` and `#endif`, enclose the variable code that can be *included* or *excluded* for different selections of features (*configurations*).

Although bringing important benefits, (preprocessor-based) variability also comes at a cost. In fact, multiple research indicate that variability amplifies maintenance problems [84, 59, 80, 64]. For instance, it obfuscates the source code and reduces comprehensibility, making maintenance error-prone and costly. As a consequence, *configuration-dependent bugs* (a.k.a., *variability bugs*) appear [2]. (They will play a predominant role in this dissertation.) Figure 1.1 illustrates a configurable program with a bug involving one optional feature. One feature gives rise to *two* possible configurations: a program *with* the assignment statement (`err = -1`) in line 6, and a program *without* it. In general, n features will give rise to 2^n configurations; i.e., 2^n program variants (assuming there are no so-called feature constraints that invalidate certain features combinations). Programming errors now depend conditionally on the configurations selected. Importantly, variability errors thus occur only in

```
 1  int netpoll_setup(void) {
 2     int err;
 3     int ipv4 = 1;
 4
 5     #ifdef CONFIG_IPV6          // DISABLED
 6     err = -1;                   // NOT compiled
 7     #endif
 8
 9     if (ipv4)
10        return err;              // BUG: Uninitialized Variable
11
12     return 0;
13  }
```

Figure 1.1: Example of a *variability* bug in a configurable program.

particular configurations and not in others. For instance, in our example, if the feature `CONFIG_IPV6` is *disabled*, the function returns the value of an *uninitialized variable*, `err`, intended originally to hold an error value in case of unexpected situations. If `CONFIG_IPV6` is *enabled*, then `err` is initialized in line 6 and, consequently, there is no *uninitialized variable* error. A bug like this one is known as *variability bug*, since it occurs in some program variants but not in others [2]. In fact, the error in Fig. 1.1 occurs only whenever we *disable* `CONFIG_IPV6` (which means that the statement in line 6 is not executed; hence, the variable `err`, which was declared and uninitialized in line 2, is never assigned a value. Since the value of the variable `ipv4` is 1 (`true`) in line 3, the `return` statement in line 10 is executed returning the value of the *uninitialized* variable `err`).

Despite the widespread adoption of preprocessor-based variability, there is little to no hard evidence on *how* variability affects programs and programmers. We lack empirical studies and tools to better understand the effect of variability and to efficient analyze all program variants, respectively. Specifically, how does variability affect programmers during maintenance tasks (bug finding in particular)? How much harder is it to debug a program as variability increases? How do developers debug programs with variability? In what ways does variability affect bugs? Are variability bugs limited to specific type of bugs, features, or locations in the code base?

In this Ph.D. thesis, I set off to address such issues through different perspectives using empirical research (based on controlled experiments) in order to understand quantitatively and qualitatively the impact of variability on buggy programs and on programmers at bug finding.

From the program (and bug) perspective, I observe that variability is ubiquitous. There appears to be no specific nature of variability bugs that could be exploited. Variability bugs are not confined to any particular type of bug, error-prone feature, or location. In addition to introducing an exponential number of variants, variability increases the complexity of bugs due to unintended feature interactions, hidden features, combinations of layers (code, mapping, model) involving different languages (e.g., C, cpp, Kconfig for Linux), many function calls, etc.

From the programmer (and bug-finding) perspective, I find that the time needed for finding bugs increases *linearly* with the number of features, while finding bugs in the first place is relatively independent of variability. In fact, most developers correctly identify bugs in programs with variability, yet many fail to identify the exact set of erroneous configurations. This is consistent with earlier hypotheses that programmers introduce errors because it is difficult to reason about all configurations. I also observe that developers navigate much more between definitions and uses of program objects when interleaved with variability.

Finally, I discuss a variability-aware solution, called *rewriting variability*, that enables single-program analysis tools, which do not handle variability by design, to detect successfully and efficiently (variability) bugs in configurable programs.

Overall, this Ph.D. thesis shows that variability complicates the complexity of bugs and bug finding, but not dramatically so. This is positive and consistent with the existence of highly-configurable software systems with hundreds, even thousands, of features, testifying that developers in the trenches are able to deal with variability.

1.2 Contributions

This dissertation makes the following main contributions:

- *Understanding the impact of variability in terms of the time and accuracy of bug finding in highly-configurable systems* (Section 3.1). This study consists of a controlled experiment with N=69 participants to quantify the impact of variability on debugging of preprocessor-based programs. I measure speed and precision for bug finding tasks defined at three different degrees of variability on several subject programs derived from real systems.

- *Providing the first study of variability debugging using eye tracking* (Section 3.2). This study describes an eye-tracking experiment with follow-up interviews to investigate more precisely *how* developers approach and debug programs in presence of variability. I ask developers to debug programs *with* and *without* variability, while recording their eye movements using an eye tracker.

- *Identification and in-depth analysis of 98 bugs from four highly-configurable software systems: Linux, Apache, BusyBox, and Marlin* (Section 4.1). These bugs consist of a wide variety of actual variability errors in configurable software. This study includes an in-depth analysis and presentation for non-experts, resulting in the *Variability Bug Database* (VBDb).[1]

- *Automated analysis and classification of 400,000 configuration-dependent warnings in the Linux kernel* (Section 4.2). This study analyzes more than 20 thousand valid and distinct random configurations, resulting in a total of 400,000 configuration-dependent warnings. Based on this corpus, I provide insights in the distribution of warning types and the location of the warnings.

- *A variability-aware technique, rewriting variability, that enables off-the-shelf single-program analysis tools to effective and efficient find a wide class of variability bugs in configurable software* (Section 5.1). This technique relies on a series of variability-related transformation rules that translate configurable programs into single programs by replacing compile-time variability with run-time variability (non-determinism).

1.3 List of Publications

Here is the list of the research papers that this Ph.D. dissertation is based upon. Notice that all published research papers were peer-reviewed.

Paper 1A *How Does the Degree of Variability Affect Bug Finding?*
Jean Melo, Claus Brabrand, and Andrzej Wąsowski.

[1] http://VBDb.itu.dk

In Proceedings of the 38th International Conference on Software Engineering (ICSE), pages 679–690, 2016.

Paper 1B *Variability through the Eyes of the Programmer.*
Jean Melo, Fabricio Batista Narcizo, Dan Witzner Hansen, Claus Brabrand, and Andrzej Wąsowski.
In Proceedings of the 25th International Conference on Program Comprehension (ICPC), pages 34–44, 2017.

Paper 2A *Variability Bugs in Highly-Configurable Systems: A Qualitative Analysis.*
Iago Abal, Jean Melo, Stefan Stanciulescu, Claus Brabrand, Márcio Ribeiro, and Andrzej Wąsowski.
Accepted for publication in the ACM Transactions on Software Engineering and Methodology (TOSEM).

Paper 2B *A Quantitative Analysis of Variability Warnings in Linux.*
Jean Melo, Elvis Flesborg, Claus Brabrand and Andrzej Wąsowski.
In Proceedings of the 10th International Workshop on Variability Modelling of Software-intensive Systems (VaMoS), pages 3–8, 2016.

Paper 3A *Effective Analysis of C Programs by Rewriting Variability.*
Alexandru F. Iosif-Lazar, Jean Melo, Aleksandar S. Dimovski, Claus Brabrand and Andrzej Wąsowski.
In The Art, Science, and Engineering of Programming Journal (Programming), Vol. 1(1): 1-25, 2017.

1.4 Outline

The dissertation is organized as follows:

- Chapter 2 states the three problems, including research questions, that are addressed in this PhD dissertation.

- Chapter 3 describes two controlled experiments in order to understand the impact of variability on bug finding, and *how* programmers debug with preprocessor annotations.

- Chapter 4 presents a *qualitative* study of 98 variability bugs in four highly-configurable systems: Linux, Apache, BusyBox, and Marlin; as well as a *quantitative* analysis of configuration-dependent warnings in Linux.

- Chapter 5 describes the proposed *rewriting variability* technique for lifting conventional single-program analysis tools to find variability bugs, followed by an evaluation.

- Chapter 6 reviews state-of-the-art related work.

- Chapter 7 provides a sketch of one possible solution for helping developers reason about multiple configurations.

- Chapter 8 draws my final conclusions, summarizes the contributions of this PhD thesis, and provides directions for future work.

Chapter 2

Problem Definition

	Problem Space	Solution Space
Programmer Perspective	– Quadrant 1 – **P1**: No hard evidence on how variability affects programmers. **Q1**: How does variability affect programmers on bug finding? **T1**: Variability increases linearly the bug-finding time and makes it difficult to identify the erroneous configurations.	– Quadrant 4 – **P4**: Programmers fail to reason about configurations. **Q4**: How to support programmers to reason about configurations? **T4**: Beyond the scope of this dissertation. (Chapter 7 provides a sketch of one possible solution.)
Program Perspective	– Quadrant 2 – **P2**: Lack of evidence on how variability affects bugs. **Q2**: How does variability affect bugs? **T2**: Variability increases the complexity of bugs; and these (variability) bugs are not confined to any type of bug, feature, or location.	– Quadrant 3 – **P3**: Infeasible to lift all program analyses to deal with variability. **Q3**: How to lift conventional analysis tools to find variability bugs? **T3**: Rewriting variability enables single-program analysis tools to find bugs in highly-configurable systems.

Table 2.1: Overview of the research Problems, Questions, and Theses.

This chapter scopes the research problems on variability challenges and solutions for programs and programmers that I focus on this dissertation.

Table 2.1 presents an overview of the research problems, questions, and theses that I am interested in. I investigate variability from the *problem* and *solution* space (columns) in combination with the *programmer* and *program* perspective (rows). Each quadrant consists of a triple (P, Q, T) where P stands for Problem, Q for research Question, and T for Thesis. Notice that Quadrant 4 is beyond the scope of this Ph.D. thesis. However, I enumerate its (deliberately open) research problem like for the others and, in Chapter 7, I provide a sketch of one possible solution that might help developers reason about configurations by providing a simplified program with a variability skeleton for a given program point.

In the following, I detail each quadrant element (P, Q, T), beginning with the problems:

P1 ***[problem space, programmer perspective]*** **– No hard evidence on how variability affects programmers on bug finding.** Recent literature is riddled with unsubstantiated claims indicating that variability increases complexity and makes reasoning about programs more difficult. I list a few examples: *"bug-finding is a time-consuming and tedious task in the presence of variability"* [80]; *"managing variability can become complex"* [91]; *"variability specifications and realizations tend to* *erode in the sense that they become overly complex*" [98]; "*understandability and maintainability may be negatively affected*" [40]. However, there is little to no hard evidence for these claims. Specifically, how does variability affect programmers during maintenance tasks (bug finding in particular)? To what extent does variability affect quality of debugging? How much harder is it to debug a program as variability increases? How do developers debug programs with variability? Do developers consider all execution paths (configurations) when debugging programs with variability?

P2 *[problem space, program perspective]* – **Lack of studies for understanding the complexity and nature of variability bugs.** Little effort has been put into understanding what kind of bugs appear in highly-configurable systems, and what are their variability characteristics. Variability is an essential aspect of highly-configurable systems, from which understanding it would help to ground research on variability-sensitive analyses. It would also inspire the creation of programmer support and bug finding tools, contributing to more effective debugging and, ultimately, fewer bugs in highly-configurable systems. Specifically, in what ways does variability affect bugs? Are variability bugs limited to specific type of bugs, features, or locations in the code base?

P3 *[solution space, program perspective]* – **It is infeasible to *lift* every single conventional program analysis and bug finding tools to deal with variability.** Analyzing highly-configurable systems is challenging, since the number of configurations is exponential in the amount of features. This mean that to brute force all variants individually using conventional single-program analysis tools is impractical. However, most of the modern analysis tools are built to analyze one program and thus cannot cope with variability. Recently, a handful of variability-aware analysis tools have been developed such as syntax checking [1, 39], type checking [51, 20], and static analysis [14, 13], to find bugs directly in configurable systems. But, often, these variability-aware tools are rare, experimental, and not fast enough to extensively scan the long history of real software systems like Linux. In general, it is very difficult and infeasible to re-implement all sorts of analyses, which already exist for decades in single-program analysis tools, in a variability-aware manner. In other words, the question is: how to make conventional analysis tools detect bugs in highly-configurable systems, without reinventing the wheel?

P4 *[solution space, programmer perspective]* **– Difficulty of reasoning about the configuration space (all variants) when debugging.** Highly-configurable systems include both large industrial product lines and open-source systems, such as the Linux kernel with thousands of features. Debugging such systems requires understanding the combinations of features, which indeed becomes difficult fast (cf. **Paper 1A**). For instance, as little as 33 independent features yield more possible program variants than there are people on the planet. Thus, debugging in this context is notoriously hard for programmers. In fact, I observe that the time to find a bug increases linearly with the number of features and that most developers fail to correctly identify the exact set of erroneous configurations (cf. **Paper 1A**). But, how to support programmers to correctly reason about all configurations? How to make programmers aware of the variability context when they are modifying or understanding a variable program? These are open research questions which are beyond the scope of this dissertation. I refer to Chapter 7 where I sketch one possible solution to these questions.

2.1 Research Questions and Goals

This thesis aims to tackle the following (deliberately broad) research question:

How does variability affect software maintainability?

In this context, software maintainability refers to *corrective maintenance*, according to Lientz's and Swanson's terminology [61]. Corrective maintenance is the activity of diagnosing and fixing errors in a software system.

In the following, I define detailed questions based on three high-level goals to address this overall research question. The research goals comprise two distinct perspectives on software variability: one of which focuses on bug finding from the programmer standpoint, and the other two goals are concerned with bugs (program artifacts). Note that the three goals correspond to the three questions (Q's) in Table 2.1.

Goal 1 Understand how variability affects programmers on bug finding. (**Programmer Perspective**)

Q1 How does variability affect programmers on bug finding?

Goal 2 Investigate characteristics and identify bugs caused by variability of real highly-configurable systems. (**Program Perspective**)

Q2 How does variability affect bugs?

Goal 3 Explore conventional program analysis tools and propose a technique to make them find bugs in presence of variability. (**Program Perspective**)

Q3 How to *lift* single-program analysis tools to find variability bugs?

The ultimate goal is to understand how variability impacts programmers on bug-finding and programs on bugs. In addition, based on the detailed findings, the objective is also to inspire the creation of programmer support tools addressing the challenges faced by developers when reasoning about configurations, and to lift single-program analysis tools to find variability bugs, contributing to more effective debugging and, ultimately, fewer bugs in highly-configurable systems.

2.2 Theses

Corresponding to the research questions, I elaborate the following theses:

T1 The time needed for finding bugs appears to increase *linearly* with the number of features, while effectiveness of finding bugs is relatively independent of variability. However, identifying the exact set of erroneous configurations is difficult, already for a low number of features. Variability also increases the number of gaze transitions (eye movements) between definition-usages for variables and call-returns for methods.

T2 Variability is ubiquitous. There appears to be no specific nature of variability bugs that could be exploited. Variability bugs are not confined to any particular type of bug, error-prone feature, or location. Variability also increases the complexity of bugs due to unintended feature interactions, hidden features, combinations of layers (code, mapping, model) involving different languages (e.g., C, cpp, Kconfig for Linux), many function calls, etc.

T3 Conventional program analysis tools, which do not handle variability, are able to effectively and efficiently find bugs in configurable programs by rewriting variability. Rewriting variability appears to be a cost-efficient solution to *lifting* program analyses.

In summary, this Ph.D. thesis shows that:

> Variability complicates bug finding and bug complexity, but not terribly so. Also, by rewriting the variability in the source code, off-the-shelf single-program analysis tools are able to detect successfully and efficiently variability bugs.

Chapter *3*

Variability Challenges for Programmers

This chapter gives an overview of the results from the two publications dedicated to my first research question (**Q1**), as described in Chapter 2. Here, I provide evidence related to the problem-programmer quadrant, as highlighted in Table 3.1, by summarizing the publication contributions to **Goal 1** and **Q1**. The first quadrant is about the intersection between problem space and programmer perspective. That is, it regards the problems of variability on bug finding from the programmer point of view. In the following, I describe **Paper 1A** and **Paper 1B**, which gather evidence on this matter.

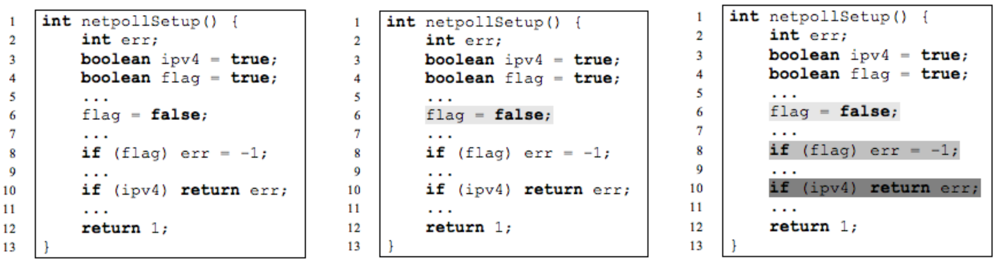
	Problem Space	Solution Space
Programmer Perspective	– Quadrant 1 – P1: No hard evidence on how variability affects programmers. Q1: How does variability affect programmers on bug finding? T1: Variability increases the bug-finding time and makes it difficult to identify the erroneous configurations.	– Quadrant 4 – P4: Programmers fail to reason about configurations. Q4: How to support programmers to reason about configurations? T4: Beyond the scope of this dissertation. (Chapter 7 provides a sketch of one possible solution.)
Program Perspective	– Quadrant 2 – P2: Lack of evidence on how variability affects bugs. Q2: How does variability affect bugs? T2: Variability increases the complexity of bugs; and these (variability) bugs are not confined to any type of bug, feature, or location.	– Quadrant 3 – P3: Infeasible to lift all program analyses to deal with variability. Q3: How to lift conventional analysis tools to find variability bugs? T3: Rewriting variability enables single-program analysis tools to find bugs in highly-configurable systems.

Table 3.1: Research problem, question, and thesis of the Quadrant 1.

3.1 How Does the Degree of Variability Affect Bug Finding? (Paper 1A)

Summary

This paper investigates the impact of variability on bug finding from the developer standpoint. Although software projects embrace variability to increase adaptability and to lower cost, many researchers and practitioners blame variability for increasing complexity and making reasoning about programs more difficult. However, there is little to no hard evidence on how exactly variability affects maintenance tasks, bug



```
int netpollSetup() {
    int err;
    boolean ipv4 = true;
    boolean flag = true;
    ...
    flag = false;
    ...
    if (flag) err = -1;
    ...
    if (ipv4) return err;
    ...
    return 1;
}
```

(a) Zero features (NO).

```
int netpollSetup() {
    int err;
    boolean ipv4 = true;
    boolean flag = true;
    ...
    flag = false;
    ...
    if (flag) err = -1;
    ...
    if (ipv4) return err;
    ...
    return 1;
}
```

(b) One feature (LO).

```
int netpollSetup() {
    int err;
    boolean ipv4 = true;
    boolean flag = true;
    ...
    flag = false;
    ...
    if (flag) err = -1;
    ...
    if (ipv4) return err;
    ...
    return 1;
}
```

(c) Three features (HI).

Figure 3.1: A program with an *uninitiliazed variable* error with progressively increasing degrees of variability.

finding in particular. I therefore carry out a controlled experiment with N=69 participants to quantify the impact of variability on debugging of preprocessor-based programs. I measure speed and precision for bug finding tasks defined at three different degrees of variability on several subject programs derived from real systems.

The results indicate that the speed of bug finding appears to decrease linearly with the degree of variability, while effectiveness of finding bugs is relatively independent of the degree of variability. Additionally, identifying the set of configurations in which the bug manifests itself is difficult already for a low degree of variability. Surprisingly, identifying the exact set of affected configurations appears to be harder than finding the bug in the first place. The difficulty in reasoning about several configurations is a likely reason why the variability bugs are actually introduced in configurable programs in the first place.

Context & Motivation

The C preprocessor (CPP) is one of the *oldest*, the *simplest*, and the *most popular* [36, 60, 50] mechanisms in use to handle variability at the code level. For this reason, I consider preprocessor-based variability (a.k.a. #ifdefs) in this study.

Figure 3.1 illustrates how the task of debugging becomes more complex as the number of features increase. This code example is extracted from the Netpoll module of the LINUX kernel, slightly adapted to JAVA syntax using coloured lines instead of preprocessor [50]. The original function, called netpoll_setup in C, contains variability and is used to initialize the module. It is about 100 lines long and involves one optional feature. Historic versions of the function, contained an *error*.[1] If the feature is disabled, the function returns the value of an *uninitialized variable*, `err`, intended to hold an error value in case of unexpected situations.

Figure 3.1a shows a version of the bug as a conventional program without variability. It is fairly easy to establish that the function returns the value of an uninitialized variable in line 10. (In line 6, the variable `flag` is assigned `false` which means that the conditional statement in line 8 is not executed; hence, the variable `err` which was declared and uninitialized in line 2, is never assigned a value. Now, since the value of the variable `ipv4` is `true` (line 3), the conditional statement in line 10 returns the value of the *uninitialized* variable `err`.)

Figure 3.1b contains the same program, but now involving *one feature* shown in light gray background color (line 6). A feature, such as light gray in this example, can be configured either as *enabled* or *disabled*. Features are used in compile-time conditional directives (`#ifdefs`) to control whether to *include* or *exclude* code fragments in a program. Obviously, a feature thus gives rise to *two* possible configurations: a program *with* the light gray statement, and a program *without* it. In general, n features will give rise to 2^n configurations; i.e., 2^n program variants. Now, bugs conditionally depend on configurations. That is, they become *variability* bugs (i.e., bugs that occur in some configurations and not in others). In fact, the error in Figure 3.1b occurs only whenever the light gray feature is *enabled*.

Figure 3.1c shows the same programs as before, but now with *three* features: light gray, gray, and dark gray. The three features yield eight configurations. (Here, I assume that there is no feature constraints for simplicity.) In debugging the program, the developer must somehow consider all configurations. Combinatorial problems are difficult for humans. Indeed, for our program in Figure 3.1c, the error now occurs in exactly three (out of eight) configurations.

But, how are maintenance tasks (bug finding in particular) affected by variability? How much harder is it to debug a program as variability increases? Does variability affect speed or also quality of debugging? In this paper, I set off to understand such issues using a controlled experiment designed to quantify the impact of the degree of variability on program code on bug finding.

[1] `http://git.kernel.org/cgit/linux/kernel/git/stable/linux-stable.git/commit/?id=e39363a9def53dd4086be107dc8b3ebca09f045d`

Method

In the experiment I use simplifications of real bugs extracted from Linux, BusyBox, and BestLap. Given a program and a degree of variability, I ask the participants to debug the programs. In other words, I perform a range of classic *"find the bug"* experiments [72] and measure the *time* and *accuracy* of the bug-finding task. I settled on using three distinct degrees of variability. Let $\mathbb{F}$ denote the set of features (conditional compilation symbols used in a program). First, to establish a baseline, I consider programs with no variability (degree `NO`, $\mathbb{F} = \emptyset$). Then, I consider programs that use one feature (degree `LO`, $|\mathbb{F}| = 1$) and programs with three features (degree `HI`, $|\mathbb{F}| = 3$). The number of configurations grows from one for degree `NO` programs, two for `LO` programs, to eight for `HI`. This should make any performance differences manifest themselves clearly. Even though it would be interesting to study higher degrees, the limitation to three features has one important advantage: it leaves us with programs sufficiently small to be used in a time-delimited controlled experiment. Lastly, I measure the time for them to find the bug, the number of correct vs incorrect identifications of bugs, and whether they can pinpoint the exact set of erroneous configurations.

I derive programs of lower degrees by taking an erroneous program with three features and appropriately fix features as either *enabled* or *disabled* retaining the original error (cf. **Paper 1A**). I thus obtain three versions of each program: "`NO`", "`LO`", and "`HI`" (much like in Fig. 3.1).

I ran the experiment with N=69 participants: 31 M.Sc. students, 32 Ph.D. students and 6 post-docs from three Danish universities: IT University of Copenhagen (ITU), University of Copenhagen (KU), and Technical University of Denmark (DTU).

Results

In this paper, I make eight observations addressing two research questions—the impact of variability on the *time* and *accuracy* of bug finding. Here, I detail five main observations. I refer to **Paper 1A** for more details (including a few more observations).

> OBSERVATION 1: *Mean bug-finding time appears to increase linearly with the degree of variability.*

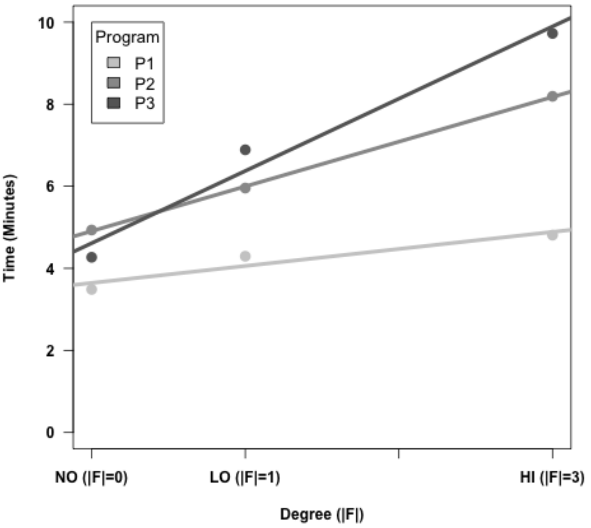


Figure 3.2: Mean bug-finding time (along the y-axis in minutes) as a function of the degree of variability (x-axis).

Figure 3.2 plots the mean bug-finding times (in minutes, along the y-axis) for each of our three benchmark programs. Each dot depicts the mean time to find the bug, for a particular program (P1, P2 and P3), for a particular degree of variability, i.e., NO ($|\mathbb{F}| = 0$), LO ($|\mathbb{F}| = 1$), and HI ($|\mathbb{F}| = 3$). For instance, the fastest mean bug-finding time is about $3\frac{1}{2}$ minutes (for program P1 with NO variability), whereas the slowest mean bug-finding time is a bit less than 10 minutes (for program P3 with a HI variability degree of $|\mathbb{F}| = 3$). For each program, I fit a regression line to its respective points. The lines suggests that the mean bug-finding time increases *linearly* with the degree of variability. According to an ANOVA test, the difference between bug-finding times for distinct degrees of variability is statistically significant, with p-value $= 2.0 \times 10^{-8}$. Also bug-finding time is a linear function of programs and degrees, with p-value $= 3.6 \times 10^{-9}$, by F-test for regression.

Recall that the number of variant programs to be considered by a participant grows *exponentially* with the degree of variability (i.e., $|\mathbb{K}| = 2^{|\mathbb{F}|}$). Clearly, a developer has to somehow consider each of the $2^{|\mathbb{F}|}$ variants in order to make an accurate diagnosis of the bug. After all, each of the variants may or may not harbour a bug. One might then, in fact, suspect that bug-finding time ought to increase *exponentially* with the degree of variability.

The post-treatment interviews provide qualitative insights into how the participants approached the problem and what difficulties they faced in understanding programs with a `HI` variability degree. The participants agree that finding bugs in the `NO` programs, so without variability, required less effort than in programs with `HI` degree of variability. One participant explains:

> *"I tried to keep all different paths in mind, but it was especially difficult with multiple colors [`HI`]."*

Along the same lines, another participant says:

> *"With more variability [`HI`] you need to build up exponentially more traces in your head."*

The participants analyze programs as one unit despite variability. They do *not* split the task into analysis of exponentially many independent programs, one variant at a time. An unconscious use of brute force would yield a $2^{|\mathbb{F}|}$ factor slow down in overall bug-finding time.

Hick's Law [44] from psychology, based on so-called *choice-reaction-time experiments*, explains that the amount of time for a human response increases logarithmically with the number of possible choices. Compared to a baseline program with `NO` variability, programs with higher degrees of variability involve exponentially more choices to be made. Obviously, composing an exponential function with a logarithmic one yields a linear function. We thus hypothesize that the seemingly linear increase in bug-finding time, in spite of the exponential blow up, can be attributed to Hick's Law.

In summary, the first observation indicates that an increase in variability (e.g., by adding features) complicates bug finding, but not dramatically and not prohibitively so. This is a very positive finding, that is consistent with existence of software products with hundreds, even thousands, of features, testifying that developers in the trenches *are* able to deal with variability.

> Observation 2: *The variance of bug-finding time appears to be amplified by the degree of variability.*

Figure 3.3 shows the distribution of bug-finding times for each program and variability degree. Each box encapsulates the middle 50% data points. The lower and upper limit of the box respectively represent the lower and upper quartiles (the 25% and 75% percentiles). The upper and lower whiskers represent the data above and below the middle half of

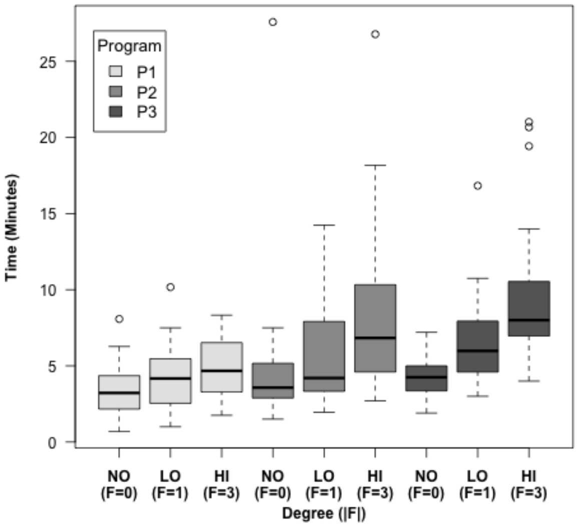


Figure 3.3: The distribution of bug-finding time.

the data. The horizontal line within the box draws up the *median* of the data points. Finally, the circles above the boxes visualize outliers. For instance, for program P3 (the three rightmost boxes), the middle half of the participants spent between $3\frac{1}{3}$ and 5 minutes to find the bug with NO variability, whereas, for HI variability, the middle half spent from about 7 to $10\frac{1}{2}$ minutes. Again, considering only participants that found the bug yields a similar diagram, consistent with the above.

Amplification of variance is a predictable consequence of our first finding. For the *variance* of a stochastic variable, X, multiplied by a constant factor, c (depending on the degree of variability), we have that: $\text{Var}(c\,X) = c^2\,\text{Var}(X)$.

In popular terms, this observation means that differences in bug-finding competences are amplified when working with variability. Ultimately, this may mean that getting talented developers on such projects is important.

> Observation 3: *Most developers correctly identify bugs in programs regardless of the degree of variability.*

Figure 3.4 shows what percentage of developers were able to find the bugs correctly. The *incorrect* answers are black, and the *correct* ones are gray. The data is presented for each degree of variability separately. The

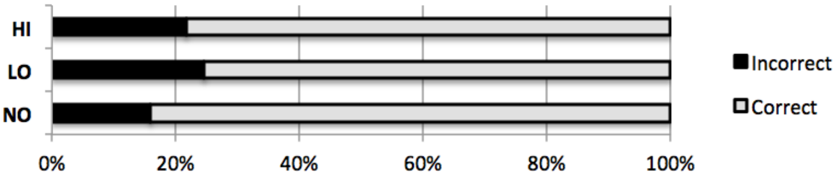


Figure 3.4: Ratio of *incorrectly* vs. *correctly* identifying a bug.

frequency of incorrect answers is consistently low, with around a fifth being the incorrect answers. For programs with NO variability, 16% of subjects (11 out of 69) did not find the bug. Even for the HI variability programs, only 22% of the subjects (15 out of 69) answered incorrectly.

Generally, developers seem to be good at finding bugs in programs—and in programs with variability (at least, up to three features). Interestingly, more than half (38 out of 69) of the participants correctly identified the bug in *all* three tasks. On average, disregarding the variability degrees, 79% of the participants were able to correctly find the bug. All in all, I conclude that the ability to find bugs in programs seems not to be significantly affected by the degree of variability (at least for $|\mathbb{F}| \leq 3$).

> Observation 4: *Many developers fail to exactly identify the set of erroneous configurations, already for a low degree of variability.*

Let us now look a little closer at accuracy and split the *correct* answers in two categories. If the participant got the set of erroneous configurations exactly right, I classify her answer as *fully correct*. Similarly, I classify answers as *partially correct*, if the developer has correctly identified the bug, but failed to correctly specify the set of configurations in which the error occurred (missing some configurations or listing too many). I ignore incorrectly identified bugs for this part of the analysis, as it is hard to interpret the identification of configurations for them. For instance, program P3 with HI variability, contains an *assertion error* that occurs in two (out of eight) configurations. For this task, some participants found only one of the erroneous configurations and others listed extra configurations for which the error does not occur. Figure 3.5 presents the numbers of fully and partially correct answers at different degrees of variability.

Obviously, partial correctness does not make sense for programs without variability (for NO we have only one possible configuration).

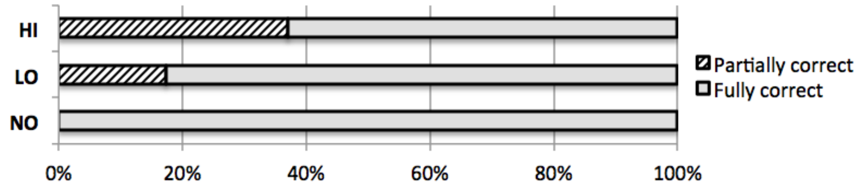


Figure 3.5: Ratio of *partially correctly* vs. *fully correctly* identifying a bug.

Already for LO variability (one feature), notice that the number of partially correct answers quickly rises to 17% (9 out of 52). For HI variability, this number escalates to almost 40% (20 out of 54).

Identifying the exact set of erroneous configurations seems to become difficult already for $|\mathbb{F}| = 3$ (HI variability). Doing this requires understanding the combinations of features that enable the incriminated execution paths—a form combinatorial reasoning, which apparently becomes difficult fast. Such problems are notoriously hard for humans. For realistic systems, where a feature model additionally shapes the set of legal configurations, this task would presumably be *even harder* (as one needs to reason about feature model constraints, in addition).

From prior qualitative studies [2, 64], it is known that programming errors related to variability appear due to inability of programmers to correctly reason about all variations of the program that they are modifying. Those findings are consistent with the above: it is plausible that developers mis-identify the sets of configurations during programming tasks and during debugging tasks for the same reasons. To the best of my knowledge, this study presents the first quantitative confirmation that indeed reasoning about multiple configurations is a challenge, even for relatively small sets.

> Observation 5: *For higher degrees of variability, it appears to be more difficult to correctly identify the set of erroneous configurations than to find the bug in the first place.*

For HI variability, 22% (15 out of 69) participants did not find the bug (see Figure 3.4). Among the ones that did, a staggering 37% (20 out of 54) erred on set of erroneous configurations (cf. Figure 3.5).

Although the participants were only asked to *find* the bugs, not (also) *fix* them, I find that these results are consistent with studies of creating and fixing bugs. Yin and coauthors report that in general bug fixers "*may* *forget to fix all the buggy regions with the same root cause.*" [97]. Another study also reports that bugs are introduced because the programmers do not realize the complexity of all the configurations in which their code will run [2]. This is also confirmed by another interview study that many developers only check a few configurations of the source code in practice [64].

Contributions

This paper contributes to **Goal 1**, which aims to understand how variability affects programmers on bug finding. Research question **Q1** asks for the impact of variability on debugging. Responding to this question, the results of this paper show that bug-finding time appears to increase linearly with the degree of variability. This conclusion is both positive and negative. An increase in variability complicates bug finding (negative), but not dramatically so (positive) – if developers reasoned about each of the variants separately I would have observed an exponential, not linear, growth. The practical implication is that it is beneficial to introduce variation points into programs from the debugging perspective: It is beneficial to pay a linear price for bug finding, if the alternative is to maintain a super-linear set of variants (at least up to three variations in a file). However, there might be benefits in selecting designs (architectures and algorithms) that require less variability, if possible.

Somewhat expectedly, the variance in bug-finding time is amplified by variability. In other words, differences in bug-finding competences of developers appear to be amplified when working on software projects with variability. Getting talented developers for such projects might be important.

I also find that most participants correctly identify bugs in programs with accuracy regardless of the degree of variability. However, developers often fail to exactly identify the set of erroneous configurations (in which the bug manifests itself), and this happens already for a rather low number of features, and gets worse when variability increases. Clearly, reasoning about multiple configurations is a challenge. This is consistent with earlier qualitative indications that variability bugs appear, when developers unintentionally ignore an execution that is enabled by an unexpected (for them) configuration of features [2, 64].

```java
import java.util.Random;

public class Http {
    String subject = null;
    int totalLength = 600;
    final int HTTP_UNAUTHORIZED = 401;
    final int HTTP_NOT_IMPLEMENTED = 501;
    boolean LARGE_FORMAT = false;
    String REQUEST_GET = "GET";


    public void sendHeaders(int responseNum) {
        if (LARGE_FORMAT) {
            int buf = 0;
            buf = totalLength - responseNum;
            subject = "response header";
        }
        if (subject.isEmpty())
            subject = "Void response";
        System.out.println("done");
    }

    private void handleIncoming(String requestType) {

        boolean http_unauthorized = new Random().nextBoolean();
        if (http_unauthorized)
            sendHeaders(HTTP_UNAUTHORIZED);


        if (!requestType.equals(REQUEST_GET))
            sendHeaders(HTTP_NOT_IMPLEMENTED);

    }

    public static void main(String[] args) {
        Http http = new Http();
        http.handleIncoming("POST");
    }
}
```

(a) *Without* variability.

```java
import java.util.Random;

public class Http {
    String subject = null;
    int totalLength = 600;
    final int HTTP_UNAUTHORIZED = 401;
    final int HTTP_NOT_IMPLEMENTED = 501;
    #ifdef CONFIG_FEATURE_HTTPD_CGI
    String REQUEST_GET = "GET";
    #endif

    public void sendHeaders(int responseNum) {
        #ifdef CONFIG_LFS
        int buf = 0;
        buf = totalLength - responseNum;
        subject = "response header";
        #endif
        if (subject.isEmpty())
            subject = "Void response";
        System.out.println("done");
    }

    private void handleIncoming(String requestType) {
        #ifdef CONFIG_FEATURE_HTTPD_BASIC_AUTH
        boolean http_unauthorized = new Random().nextBoolean();
        if (http_unauthorized)
            sendHeaders(HTTP_UNAUTHORIZED);
        #endif
        #ifdef CONFIG_FEATURE_HTTPD_CGI
        if (!requestType.equals(REQUEST_GET))
            sendHeaders(HTTP_NOT_IMPLEMENTED);
        #endif
    }

    public static void main(String[] args) {
        Http http = new Http();
        http.handleIncoming("POST");
    }
}
```

(b) *With* variability.

Figure 3.6: Program `P` *without* and *with* variability.

3.2 Variability through the Eyes of the Programmer (Paper 1B)

Summary

In my previous paper (**Paper 1A**), I investigate the impact of variability on bug finding in terms of time and accuracy. However, I focus only on *quantitative* aspects of debugging, and not on *how* developers debug programs with variability. Additionally, little is known about the cognitive process of debugging programs with variability. Thus, to better account for the effect of variability on debugging, I carry out a follow-up eye-tracking experiment to understand *how* developers debug programs with variability. I ask developers to debug programs *with* and *without* variability, while recording their eye movements using an eye tracker.

The results indicate, not surprisingly, that debugging time increases for code fragments containing variability. Interestingly, debugging time also seems to increase for code fragments without variability in the proximity of fragments that do contain variability. The presence of variability correlates with an increase in the number of gaze transitions between definitions and usages for fields and call-returns for methods. Variability also appears to prolong the "initial scan" of the entire program that most developers initiate debugging with.

Context & Motivation

Configurable software systems are challenging for developers because code fragments may be conditionally *included* or *excluded* depending on whether particular features are *enabled* or *disabled*. This means that developers need to reason about several different configurations (versions of the program), each with different data- and control-flow in order to understand a program with variability. This impacts debugging. In programs with variability, some errors occur conditionally, only in certain *erroneous configurations* (i.e., when certain combinations of features are enabled/disabled).

Previous studies have demonstrated that debugging is overall difficult and time consuming in the presence of variability (e.g., **Paper 1A** and [80]). In this paper, I use eye tracking to study more precisely *how* developers debug programs with variability. I compare how the developers look at a program *with* variability against a version of it *without* variability (as a baseline).

Figure 3.6 presents a code scenario extracted from BusyBox which is an open-source highly-configurable system with about 600 features that provides several essential Unix tools in a single executable file. We have adapted the extracted example from C to Java to widen the audience of potential participants for the experiment.

Figure 3.6a shows the version of this program *without* variability derived from the original version with variability shown in Fig. 3.6b. The program in Figure 3.6a contains an error in line 18 where evaluation of the expression `subject.isEmpty()` causes a *null-pointer exception* because `subject` has the value `null`. The entry point `main` calls `handleIncoming` in line 37 which, in turn, calls `sendHeaders` in line 27. This method then skips past the statements in lines 14–16 because the variable `LARGE_FORMAT` has the value `false` (line 8). Hence, when we reach line 18, the variable `subject` has never been assigned a proper value aside from its initialization to `null` in line 4.

Figure 3.6b depicts the original version of the program *with* variability. Notice that the program now contains three so-called *features*: `LFS`, `AUTH`, and `CGI` (names abbreviated). Each of these three features can be designated as either *enabled* or *disabled*. Features are used in conditional compilation directives (`#ifdefs`), which control whether to *include* or *exclude* code before compilation, depending on whether features are *enabled* or *disabled*. For instance, the fragment in lines 14–16 (wrapped in an `#ifdef` and `#endif` in lines 13 and 17) is to be *included* in the code if LFS is enabled; and *excluded* if LFS is disabled. Since n features yield 2^n distinct configurations, our variability program with three features then comes in eight (2^3) distinct configurations, each corresponding to a different version of the program.

The *null-pointer exception* from before now only appears in specific configurations: whenever we *disable* the feature LFS as well as *enable* either AUTH or CGI. The exception thus occurs in exactly three (out of eight) configurations. The error no longer occurs if we, for instance, *enable* LFS; then subject is indeed assigned a non-null value in line 16. Also, if we do not *enable* either AUTH or CGI, sendHeaders is no longer invoked in line 27 or 31. The developer must thus somehow consider *all* configurations when debugging a variability program. Further, for a program with variability it is not enough to simply find an error in some configuration. In order to *fix* a bug, a developer must thus not only identify the error, but also correctly identify the exact set of erroneous configurations (combinations of feature enablings/disablings). If the developer gets the configurations wrong, the bug may only be partially fixed. Clearly, this is a difficult task due to its combinatorial characteristics.

For these reasons, a developer has to be highly alert and conscious of the features and #ifdefs in the code. In **Paper 1A** I have demonstrated to what extent variability complicates debugging. In this paper, I consider *how* variability impacts debugging.

Method

In the experiment, I use the Tobii EyeX Controller integrated into the EyeInfo Framework and an open-source tool called OGAMA[2] to record all of the eye movements. I performed this eye-tracking experiment with N=20 participants: seven undergraduate students, one M.Sc. student, seven Ph.D. students, and five post-docs at the IT University of Copenhagen (ITU). I expose each participant to programs *with* and *without* variability, while controlling for noise factors such as learning effect, developer competence, and program complexity. Technically, the experiment is a *within-group design* in which all participants are exposed to every treatment.

[2] http://www.ogama.net/

	Program P	Program Q
without variability	$5\frac{3}{4}$ min	5 min
with **variability**	$10\frac{1}{2}$ min	$10\frac{1}{2}$ min

Figure 3.7: Average total debugging times.

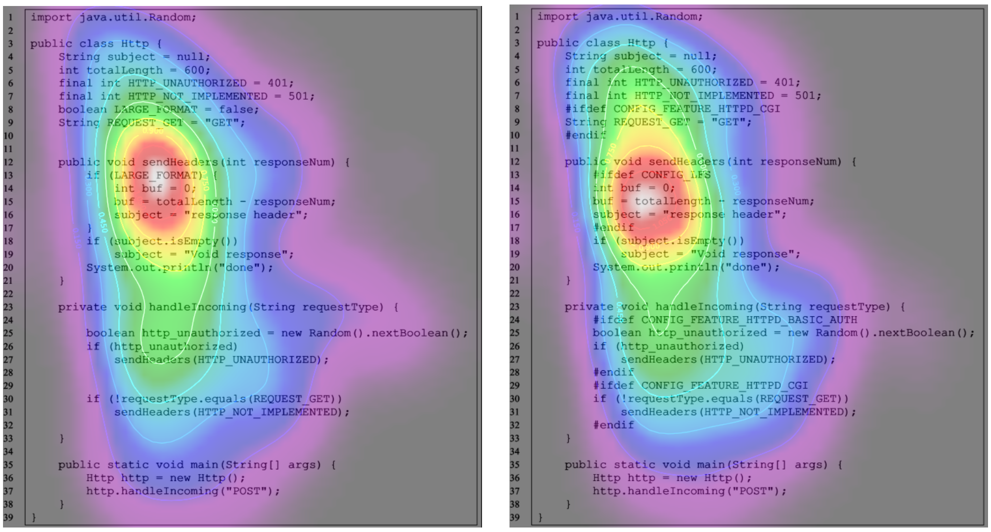
Results

Figure 3.7 shows the average total debugging time for each of the two programs P and Q, *without* variability (zero features) vs. *with* variability (three features). For both programs, the average total debugging time goes up from roughly *five* to *ten* minutes when the programs involve variability; i.e., the debugging time is *doubled*. Using the eye tracking data we can investigate deeper *where* developers are spending all this extra debugging time. In the following, based on this eye-tracking study, I present four main observations on *how* developers debug programs with variability. (The paper contains more findings. I refer the reader to **Paper 1B** for more details/information.)

> Observation 1: *Variability appears to increase debugging time of the areas of the program that contain variability.*

Figure 3.8 shows the aggregated heat maps for the program P *without* variability (to the left) versus *with* variability (to the right). Aggregated heat maps are produced by first normalizing (with respect to time) and then superimposing all individual heat maps such that contributions from each developer will be accounted for equally. (Since we have N=20 participants, each aggregated heat map is derived from ten individual heat maps.) Aggregated heat maps give an overall picture of the focus of the developers; i.e., how much they were looking at each part of the program, on average. Importantly, in contrast to Figure 3.7 that considers *absolute* time, Figure 3.8 considers *relative* time: how attention is distributed among the program parts.

The *hot spots* (red regions) indicate areas where most of the attention was directed. Not surprisingly, most attention was awarded to the method containing the bug, `sendHeaders` (specifically, lines 12 to 18). Recall that the bug is in line 18 where the condition `subject.isEmpty()`



```
1  import java.util.Random;
2
3  public class Http {
4      String subject = null;
5      int totalLength = 600;
6      final int HTTP_UNAUTHORIZED = 401;
7      final int HTTP_NOT_IMPLEMENTED = 501;
8      boolean LARGE_FORMAT = false;
9      String REQUEST_GET = "GET";
10
11
12     public void sendHeaders(int responseNum) {
13         if (LARGE_FORMAT) {
14             int buf = 0;
15             buf = totalLength - responseNum;
16             subject = "response header";
17         }
18         if (subject.isEmpty())
19             subject = "Void response";
20         System.out.println("done");
21     }
22
23     private void handleIncoming(String requestType) {
24
25         boolean http_unauthorized = new Random().nextBoolean();
26         if (http_unauthorized)
27             sendHeaders(HTTP_UNAUTHORIZED);
28
29
30         if (!requestType.equals(REQUEST_GET))
31             sendHeaders(HTTP_NOT_IMPLEMENTED);
32
33     }
34
35     public static void main(String[] args) {
36         Http http = new Http();
37         http.handleIncoming("POST");
38     }
39 }
```

(a) *Without* variability.

```
1  import java.util.Random;
2
3  public class Http {
4      String subject = null;
5      int totalLength = 600;
6      final int HTTP_UNAUTHORIZED = 401;
7      final int HTTP_NOT_IMPLEMENTED = 501;
8      #ifdef CONFIG_FEATURE_HTTPD_CGI
9      String REQUEST_GET = "GET";
10     #endif
11
12     public void sendHeaders(int responseNum) {
13         #ifdef CONFIG_LFS
14         int buf = 0;
15         buf = totalLength - responseNum;
16         subject = "response header";
17         #endif
18         if (subject.isEmpty())
19             subject = "Void response";
20         System.out.println("done");
21     }
22
23     private void handleIncoming(String requestType) {
24         #ifdef CONFIG_FEATURE_HTTPD_BASIC_AUTH
25         boolean http_unauthorized = new Random().nextBoolean();
26         if (http_unauthorized)
27             sendHeaders(HTTP_UNAUTHORIZED);
28         #endif
29         #ifdef CONFIG_FEATURE_HTTPD_CGI
30         if (!requestType.equals(REQUEST_GET))
31             sendHeaders(HTTP_NOT_IMPLEMENTED);
32         #endif
33     }
34
35     public static void main(String[] args) {
36         Http http = new Http();
37         http.handleIncoming("POST");
38     }
39 }
```

(b) *With* variability.

Figure 3.8: Aggregated heat maps for the program P.

produces a null-pointer exception since the variable subject has the value null. (In the case *with* variability, this happens in certain configurations.[3]) Overall, the red regions appear quite similar. Without variability, developers dedicate 12% of all fixations to this area (752 out of 6,355). With variability, the dedication to this area is comparable in relative terms with 15% fixations (although using more fixations in absolute terms: 1,249 out of 8,339). The Kullback-Leibler Divergence test confirms that the similarity between the two hot spots is highly significant (divergence value = 0.05, in a scale [0,1]—0 meaning that there is no divergence, i.e., they are similar; and 1 means that they are totally different). I observe the same phenomenon for the hot spots in the other program Q (divergence value = 0.07).

Table 3.2 details the total time spent looking at each of the four designated *areas of interest* of the program: the field declarations (lines 4–9); the method sendHeaders (lines 12–21); handleIncoming (lines 23–33); and main (lines 35–38). For instance, the attention devoted to the method sendHeaders goes up from about a minute (63 seconds) to two minutes (120 seconds) in the presence of variability; i.e., an increase factor of 1.9 (almost twice as much attention). Overall, it appears that the extra (roughly double) debugging time is spent on all areas of the

[3] The bug occurs when LFS is *disabled* and either AUTH or CGI is *enabled*; i.e., $\neg$LFS $\wedge$ (AUTH $\vee$ CGI).

area of interest		variability		increase
lines	area	*without*	*with*	factor
4–9	*fields*	26 s	58 s	2.2 x
12–21	`sendHeaders`	63 s	120 s	1.9 x
23–33	`handleIncoming`	56 s	98 s	1.8 x
35–38	`main`	8.2 s	5.3 s	0.7 x
Σ	*all four areas*	153 s	281 s	1.8 x

Table 3.2: Average debugging time for four *areas of interest* of the program `P` *without* vs. *with* variability.

sub-area of interest		variability		increase
lines	sub-area	*without*	*with*	factor
`P:12–17`	- *with* variability	38 s	77 s	2.0 x
`P:18–21`	- ***without*** **variability**	**25 s**	**43 s**	**1.7 x**
Σ	`sendHeaders`	63 s	120 s	1.9 x
`Q:18–20`	- ***without*** **variability**	**24 s**	**45 s**	**1.9 x**
`Q:21–33`	- *with* variability	48 s	130 s	2.7 x
Σ	`gc_computeLevelScore`	72 s	175 s	2.4 x

Table 3.3: Average debugging time for fragments without variability in proximity of fragments with variability.

program that involve variability: the field declarations and the two methods `sendHeaders` and `handleIncoming` all double debugging time. In contrast, no extra time is spent on `main` that does not involve variability. In fact, attention to this area appears to drop slightly in the presence of variability.

Please note that the attention awarded to the four *areas of interest* (last line in Table 3.2) does not add up to the total debugging time of Figure 3.7. This is because the four elements do not cover everything (e.g., imports, blank lines, class definitions, and even areas beyond the screen), gaze transitions (rapid eye movements) are not accounted for in Table 3.2, and the total debugging time also involves answering questions about the bugs on a sheet of paper (i.e., not looking at the screen).

> OBSERVATION 2: *Debugging time also increases for code fragments without variability in proximity of code fragments that do contain variability.*

Consider the body of the `sendHeaders` method in program `P` with variability (cf. Fig. 3.8b). It consists of a code fragment *with* variability (lines 13–17) followed by a fragment *without* variability (lines 18–20). A similar phenomenon occurs in program `Q` in the function `gc_computeLevelScore`, where the top part (lines 18–20) does not contain variability followed by a fragment (lines 21–33) with variability.

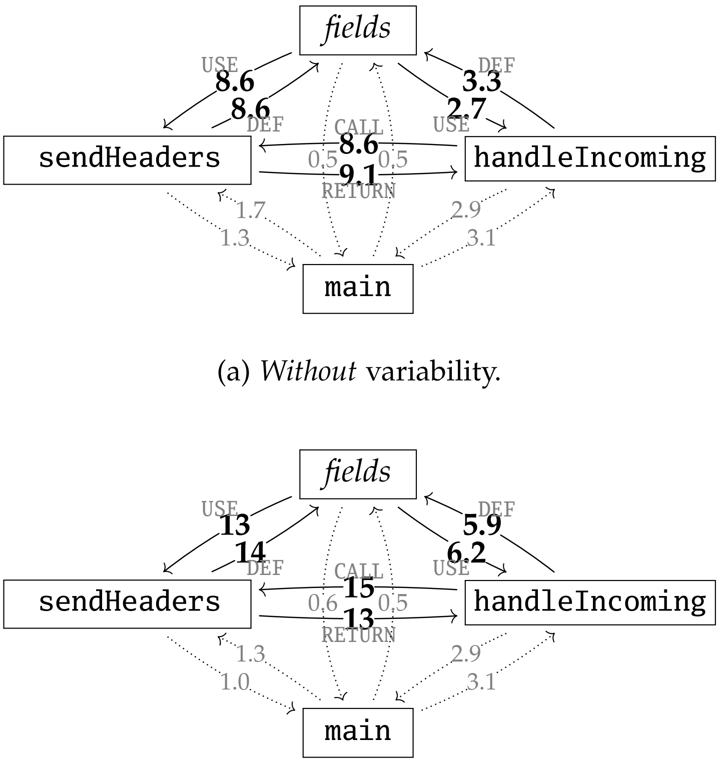
Designating these as our *sub-areas of interest*, we can thus zoom in and study the impact of code fragments *with* variability on code fragments *without* variability within the same method.

Table 3.3 splits these two methods into their sub-areas of interest with versus without variability. The sub-areas without variability "in proximity" of variability are shown in bold face. Variability appears to be "contagious" along the flow of control, within a method. Even though lines (18–21) in `P` do not have variability, they go from 25 seconds to 43 seconds to debug in the presence of variability (i.e., debugging takes 1.7 times longer). Similarly, for lines 18–20 in `Q`; they go from 24 seconds to 45 seconds (i.e., debugging takes 1.9 times longer).
I therefore hypothesize that this is because the developers are considering different configurations while debugging (more on this in OBSERVATION 4 later).

> OBSERVATION 3: *Variability appears to increase the number of gaze transitions between definition-usages for fields and call-returns for methods.*

Figure 3.9 depicts the average number of *gaze transitions* between the four previously introduced areas of interest. Without variability there are, for instance, on average 8.6 navigations from `handleIncoming` to `sendHeaders` and 9.1 back again (see Fig. 3.9a). Navigations between two methods are annotated with *call* and *return* according to invocations in the program (e.g., `sendHeaders` is called from `handleIncoming` in line 27 and 31). The gaze transition diagrams confirm that the eye movements proceed along method invocations. Similarly, *method-to-field* navigations are annotated with *def* and *use* as developers navigate from a field variable usage to its *definition* and back again to the *use*. For instance, we see on average 8.6 navigations from `sendHeaders` to the *fields* area of interest (def) and exactly the same number going back again to the usage within the method (use).

With variability, all gaze transitions out of methods containing variability increase significantly (cf. Figure 3.9b compared to Figure 3.9a).



(a) *Without* variability.

(b) *With* variability.

Figure 3.9: Average number of gaze transitions (eye switches) between the differents elements of program P.

The *method-to-method* navigation along call-return from `handleIncoming` to `sendHeaders` goes up to 15 and 13 (from 8.6 and 9.1). For *method-to-field*, the (def-use) navigations out of `sendHeaders`, for instance, goes up to 13 and 14 (from 8.6 and 8.6). For navigations out of the method `main` that does not contain any variability (shown as dotted gray edges), there is little change.

Thus, the participants make significantly more gaze transitions in the presence of variability. Again, I hypothesize that developers are exploring and re-exploring different configurations while debugging, as we shall see next.

> Observation 4: *Developers appear to debug programs with variability by considering either one configuration at a time (consecutively) or all configurations at the same time (simultaneously).*

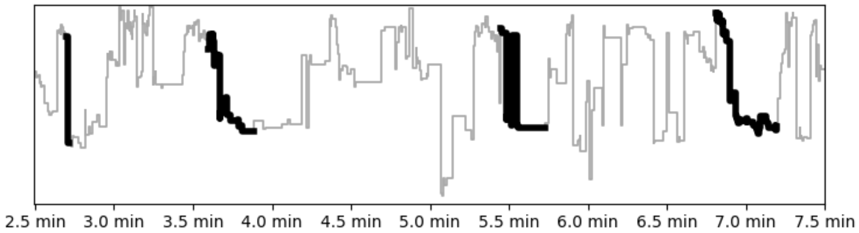


Figure 3.10: Gaze transition diagram for a developer using a consecutive strategy and repeatedly considering a method (highlighted in black).

The interviews give some qualitative insights into how the subjects debug programs with variability. Most participants complained that they had trouble finding the bug in the presence of variability. One subject explains that he is using a *consecutive strategy* by considering one configuration at a time:

> *"I began with all features enabled, then I removed one-by-one."*

Along the same lines, another explains:

> *"After I get a good understanding of the code, I started to enable/disable features one at a time to see if the bug appears."*

This approach manifests itself on his gaze transition diagram which contains repetitions corresponding to the method `sendHeaders` with variability (cf. Figure 3.10).

Another subject claims to adopt a *simultaneous strategy* by considering all configurations at the same time:

> *"I tried to keep track of everything by compiling every combination in mind."*

The two strategies are also well-known in *automated* program analysis of programs with variability [14].

Independent of strategies, all developers agreed that debugging programs *without* variability required much less effort. This finding aligns with the study of Medeiros et al. [64] in which they observed that bugs involving variability are easier to introduce and harder to debug and fix than ordinary bugs.

Contributions

This paper addresses primarily the research question **Q1**— which asks for how variability affects programmers on bug finding— and, as a result, it contributes to **Goal 1**. In fact, besides the above-mentioned observations, this paper also *confirms* previous hypotheses (from **Paper 1A**), regarding the *accuracy* of debugging programs with variability:

> CONFIRMATION: *Most developers correctly identify bugs in programs with variability; however, many developers fail to identify exactly the set of erroneous configurations (already for three features).*

This is also consistent with previous research reporting that developers admit that when fixing programs with variability, they "check only a few configurations of the source code" [64].

It is worth noting that my previous study (**Paper 1A**) uses *colored annotations*, whereas in this study I use actual preprocessor directives (`#ifdefs`). Therefore, I think that the stability of the results across multiple annotative mechanisms is a good indication that I am, in fact, studying an underlying phenomenon, which is variability.

Observation 3 (above) stated that developers perform more navigation in the presence of variability. Knowing that, I encourage the programmers using variability to structure the code in a way that minimize the distance between uses and definitions of field variable declarations or between methods calling each other, especially for those declarations and uses that involve `#ifdefs`. At the same time, the builders of development environments shall consider providing convenient ways to navigate from uses to definitions and back again and along call-returns for method invocations. An IDE equipped with continual eye tracking could even automatically "pop up" relevant definitions next to uses as they are being considered by the developer. Clearly, as shown in our data, these pop-ups might be more useful, in areas of code that involve variability (so intensive variability could activate them). Emergent interfaces [78] and emergent feature interfaces [67] are examples of tooling that attempt to simplify reasoning about variability. This study confirms the need for more research on such tools.

Observations 1–3 indicate that it is worth to contain variability in as few methods as possible to keep other methods variability free. Observation 2 hints that it is advantageous to hoist code fragments without variability "in proximity" of variability out of the method. For instance, in program `P` with variability, lines 18–20 could be moved into a fresh method.

All observations 1–4 may indicate that there are potential gains from *projectional editing* of program with variability. Developers could work separately on particular configurations (programs without variability) which would then be automatically synchonized with the entire variability program (spanning all configurations) [93]. This could be activated/suggested automatically for programmers who work following the consecutive (brute-force) process, as this process can be presumably detected automatically as multiple scans in the eye-tracking data. Of course, I do not know to what extent, or whether at all, these suggestions improve debugging programs with variability. However, the findings of this study do provide indications that these are the directions that might be worth exploring.

Chapter 4

Variability Challenges for Programs

In this chapter, I give an overview of two research papers that address the research question **Q2**, which asks for the characteristics of bugs caused by variability. In the following, I summarize **Paper 2A** and **Paper 2B** which belong to the second quadrant (problem-program), as shown in Table 4.1. This quadrant focuses on the problem space from the program (and bug) perspective.

	Problem Space	Solution Space
Programmer Perspective	– Quadrant 1 – P1: No hard evidence on how variability affects programmers. Q1: How does variability affect programmers on bug finding? T1: Variability increases the bug-finding time and makes it difficult to identify the erroneous configurations.	– Quadrant 4 – P4: Programmers fail to reason about configurations. Q4: How to support programmers to reason about configurations? T4: Beyond the scope of this dissertation. (Chapter 7 provides a sketch of one possible solution.)
Program Perspective	– Quadrant 2 – P2: Lack of evidence on how variability affects bugs. Q2: How does variability affect bugs? T2: Variability increases the complexity of bugs; and these (variability) bugs are not confined to any type of bug, feature, or location.	– Quadrant 3 – P3: Infeasible to lift all program analyses to deal with variability. Q3: How to lift conventional analysis tools to find variability bugs? T3: Rewriting variability enables single-program analysis tools to find bugs in highly-configurable systems.

Table 4.1: Research problem, question, and thesis of the Quadrant 2.

4.1 Variability Bugs in Highly-Configurable Systems (Paper 2A)

Summary

This paper extends a prior *exploratory* study on 42 variability bugs in Linux published previously by Abal *et al.* [2]. That study produced a method design and a list of nine initial hypotheses based on analyzing the Linux kernel. I then join Abal and co-authors along with Márcio Ribeiro and Stefan Stanciulescu to execute three independent *confirmatory* case studies validating the previous hypotheses. We replicated the same data collection process and analysis for the three new subject systems (Apache, Marlin, and BusyBox), which significantly differ from Linux. In the end, we confirm all previous hypotheses from the original Linux-only study. This attests to the stability and generalizability of our findings.

In this paper, we present a qualitative study of 98 diverse variability bugs (i.e., bugs that occur in some variants but not in others) collected from bug-fixing commits in the Linux, Apache, BusyBox, and Marlin repositories; 55 of these bugs have been added using the *confirmatory* part of the study. We analyze each of the bugs, and record the results in the Variability Bugs Database[1] (VBDb). 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.

Context & Motivation

Many software projects adopt variability to tailor development of individual software products to particular market niches [4]. Other software projects, such as the Linux kernel, embrace variability and use configuration options known as *features* [46] to tailor functional and non-functional properties to the needs of a particular user. Such systems are often referred to as *highly-configurable systems* and can get very large and encompass large sets of features. There exist reports of industrial systems with thousands of features [10], and extensive open-source examples are documented in detail [11].

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. 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 infeasible to analyze each configuration separately.

[1] `http://vbdb.itu.dk/`

Family-based analyses [88] tackle this problem by considering all configurable program variants as a single unit of analysis, instead of analyzing the individual variants separately. 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 [5, 13, 14, 27, 49, 55] and model checking [6, 23, 24, 58, 42, 77] based techniques have been developed.

Although several variability-aware techniques have been proposed to analyze exponentially many configurations of highly-configurable systems, little effort has been put into understanding the characteristics of bugs in these large software systems. In fact, researchers still lack examples of concrete bugs induced by variability, occurring in real large-scale systems. Gaining such understanding is needed to ground research in actual problems and to evaluate tool implementations of variability-sensitive analyses.

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.

Method

The objective of this work is to understand the complexity and nature of *variability 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.

For each of the four subject systems, 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 organize our findings.

To do so, we take the Linux[2], Apache[3], BusyBox[4] and Marlin[5] reposi-

[2] http://git.kernel.org/

[3] http://git.apache.org/httpd.git

[4] http://git.busybox.net/busybox/

[5] http://github.com/MarlinFirmware/MarlinDev

tories 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.

Results

We begin by investigating the diversity of the 98 variability bugs in terms of bug type, location, and configuration options involved. The numbers are used solely to describe the collected sample—no statistical conclusions about the broader bug population should be drawn from them. That is, the figures presented here serve exclusively to characterize population of bugs we found, not to hint at any representative bug distribution.

Diversity of bugs in VBDb

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

Figure 4.1 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[6] 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 [2]. In Figure 4.1, we

[6] *Common Weakness Enumeration* (CWE) – a catalog of numbered software weaknesses and vulnerabilities.

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

Figure 4.1: Types of variability bugs in Linux and all of VBDb. (L is for Linux, M is for Marlin, B is for BusyBox and A is for Apache.)

see that the variability bugs in Linux span 21 different kinds of bugs, falling into seven categories of the CWE taxonomy.

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

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

Figure 4.2: Features involved in variability bugs in all of VBDb.

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.

Observation 2: *Variability bugs are not restricted to any specific error prone feature.*

Figure 4.2 summarizes the distribution of features in presence conditions of bugs in our collection. We see that the Linux bugs of our collection 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. The collection is not biased for a particular feature, and no particular feature seems to be responsible for majority of bugs. Thus, there are no obvious particularly error-prone features in Linux.

We validate this hypothesis with the other three subject systems (the columns: M, B, and A). For example, for BusyBox, we see only one feature, CLEAN_UP that is involved in two bugs. 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.

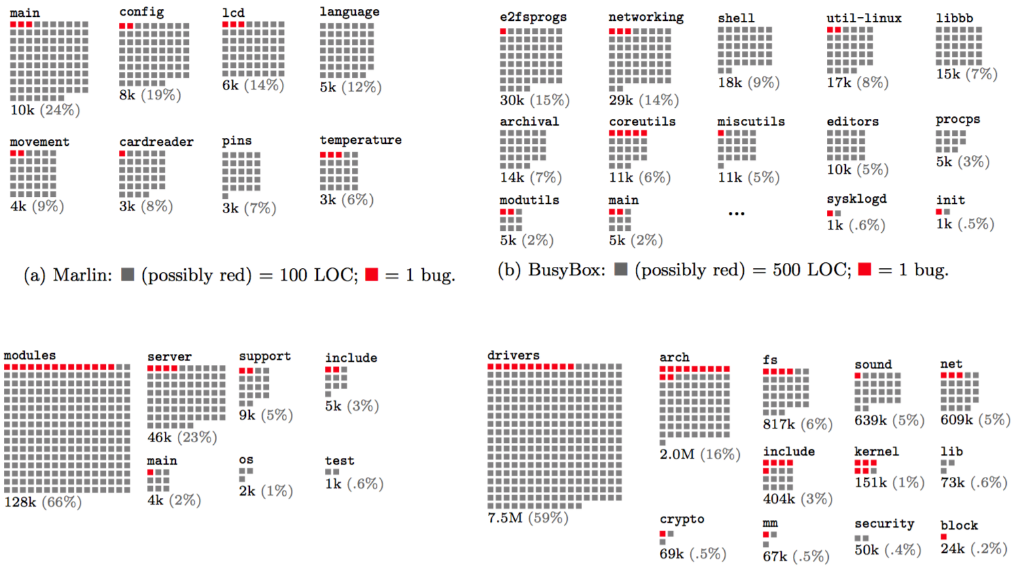
> OBSERVATION 3: *Variability bugs are not confined to any specific* location *(file or subsystem).*

Figure 4.3 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. Normalizing to a single unit would make the amount of red squares invisible.

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*[7] 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. 4.3(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.

[7] 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 KLOC; 🟥 = 1 bug.

Figure 4.3: Project structure and relative size of subsystems vs location of bugs in VBDb.

For testing the hypothesis, we collected the corresponding data for the other cases (cf. Figures 4.3(a), 4.3(b) and 4.3(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 (details in **Paper 2A**). As for Linux, *smaller* subsystems accounting for less than 0.1% are abstracted away (e.g., `serial` with 0.3k in Marlin) and *infrastructure* subsystems (e.g., `testsuite` (4k) in BusyBox); none of which harbored any of our bugs.

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. Also in BusyBox, the bugs are spread out over many distinct directories; from low-level networking (`networking/`) to user-oriented core utilities (`coreutils/`). Only three out of 14 subsystems are not represented in VBDb. Similarly, for Apache, the bugs appear to not be confined to any specific location; 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.

To summarize the above three observations, we conclude that:

> SUMMARY: *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. In other words, 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 kinds of variability bug during family-based analysis, they would thus fail to detect large classes of errors (the kinds not focused 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). Even though all of this is not so surprising, we can now *confirm* these folkloric hypotheses with *evidence* in terms of hard data.

Variability characteristics of bugs in VBDb

> 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` 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 have 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 should not simply zoom in on one subsystem (or file) without taking the features defined in other subsystems into consideration.

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

We investigate *configuration-dependent definitions* (functions, variables, macros, and types) which 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 even 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 when `CRYPTO_BLKCIPHER` is *disabled*.

For an example of different definitions in different configurations, consider Linux bug `0988c4c7fb5`. Figure 4.4 shows an excerpt 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.

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

Figure 4.4: 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).

(When the code reaches BUG(), the kernel prints out the contents of the registers and a stack trace, and then the current process dies.) 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 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*[8] 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 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). Bug 5cd6461, still in BusyBox, 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).

> <u>Observation 6</u>: *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.*

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

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

Figure 4.5: Bug-fixing layers.

A bug can be fixed in the *code*, *mapping* (feature expression–`#ifdef`), and (feature) *model*. Since bug fixes often involve multiple locations, variability bugs can occur in multiple layers. Figure 4.5 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). 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. 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.

Figure 4.5 shows that the variability bugs in Marlin, BusyBox, and Apache are also not only fixed in the *code*, but in several layers. Although, like for Linux, the variability bugs appear to be fixed predominantly in the *code* and *mapping* layers. In BusyBox, commit `199501f2a00` fixes a null pointer dereference error in the *code*. 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.

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 [75], 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

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

Figure 4.6: Variability degrees.

research on analysis of feature models does exactly that [8]. Similarly, for bug fixing techniques [41], 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.

> 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 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 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 4.6 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. For instance, Linux bug `6252547b8a7` 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 (cf. **Paper 1A** and **Paper 1B**), and outright infeasible in practice, as confirmed to us by a professional Linux developer, whom we interviewed.

Looking at the data for our three replication studies, we see 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; our study uncovered another nine such high-degree bugs. For instance, in 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 "`Accept-Encoding: gzip`", then the HTTP error response would be incorrectly marked as being gzip encoded ("`Content-Encoding: gzip`") even though it is not. Another example of a 3-degree bug in BusyBox is `b62bd7b261b`, which contains an unused pointer variable whenever `MDEV_CONF` and `MDEV_RENAME` are enabled and `MDEV_RENAME_REGEXP` is disabled. 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 worth noting that more than half of the bugs in our VBDb corpus are, in fact, *feature-interaction bugs* (cf. the Σ column in Figure 4.6). While most feature-interaction bugs have been identified, documented, and published in telecommunication domain [19], this study provides a documented collection of feature-interaction bugs in the context of a wider collection of highly-configurable systems (involving 3D printer firmware, UNIX utilities, web servers, and operating systems).

Feature-interaction bugs are inherently more complex to find and reason about (cf. **Paper 1A** and **Paper 1B**) 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.

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

Figure 4.7 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

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**

Figure 4.7: Presence conditions under which the bugs occur.

more features and disabling *exactly one* feature. 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 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. 4.6 and Fig. 4.7 may appear inconsistent.)

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 collection, then testing maximal configurations only, will miss a significant amount of bugs.

> 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.*

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

Figure 4.8: Effectiveness (cost/benefit) of various testing strategies if applied to our collection of bugs.

We propose a *one-disabled* configuration testing strategy, where we test configurations in which *at least one* feature is disabled (and preferably exactly one, if the feature model permits it).

Figure 4.8 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 since feature models are underconstrained in practice [68], a small number of configurations will suffice for real systems (thus constant in practice). For *one-disabled*, the size of the sample is always at most $|\mathbb{F}|$, the maximum is obtained if all features are 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 half of the bugs in our sample; 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 [68].

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 [26, 28, 16]), as well as other heuristic strategies such as *all-enabled*, *all-disabled*, *code-coverage* [86, 87, 85] and *random sampling* strategies. Medeiros *et al.* [63] executed a comparative quantitative study of effectiveness of various sampling strategies for testing and analysis of configurable systems, including all the above, one-disabled 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 [63]. 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.

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

> SUMMARY: *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.

Contributions

This paper is the main contribution to **Goal 2** (Program Perspective) as we set off to gain understanding on the complexity and nature of variability bugs of four highly-configurable systems. Research question **Q2** asks for how variability affects bugs. Based on the aforementioned observations, this paper shows that variability bugs are not confined to any particular *type of bugs*, error-prone *features*, or specific *locations*. 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. Indeed, perhaps we should work on methods to *lift* any analysis tools to configurable systems. We look into one method in this direction in Chapter 5. Another study attempt is the work of Dimovski *et al.* on automatic variability abstraction [34].

The paper also characterizes in what ways variability affects bugs. In addition to introducing an exponential number of program variants, variability increases the complexity of bugs along several dimensions: unintended feature interactions, hidden features, different layers and languages, many function calls, etc.

Furthermore, the tremendous variation among the bug in the VBDb collection itself (each with simplified and single-function versions and patches) provides a useful resource for further research on variability bugs and bug finders. In fact, this work has already influenced a quantitative study on the effectiveness of sampling strategies for configurable systems [63]. Al-Hajjaji *et al.* [3] 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.

4.2 A Quantitative Analysis of Variability Warnings in Linux (Paper 2B)

Summary

This paper[9] analyzes one of the largest open source projects, the Linux kernel, in order to quantify basic properties of configuration-related warnings. I use warnings as a proxy for unintended quality issues. Warnings are undesirable in mature code; it is a common practice to disallow code with compilation errors from being committed. Additionally, it is difficult to collect errors automatically, since classifying the cause of error cannot be done automatically, while it is entirely feasible for warnings. This work automatically analyzes more than 20 thousand valid and distinct random configurations, in a computation that lasted more than a month. Then, I count and classify a total of 400,000 warnings to get an insight in the distribution of warning types, and the location of the warnings. I run both on a *stable* and *unstable* version of the Linux kernel.

The results reveal that the most common warnings involve *dead code* (warnings: `unused-function` and `unused-variable`) and *uninitializations*. Interestingly, it appears that *no warnings* are configuration independent in Linux. All warnings that survive the development process, and are committed to the repositories, even in stable releases, are configuration dependent. I also find that the `drivers/` and `include/` subsystems contain most warnings, whereas there is much less warnings in core subsystems `kernel/` and `security/` of Linux. Additionally, the unstable version contains more warnings than the stable version, indicating that the Linux process for preparing stable releases does help to reduce configuration-dependent warnings.

[9] This is a joint work with Elvis Flesborg, who concluded his M.Sc. project under the supervision of Claus Brabrand and mine.

Context & Motivation

It is commonly assumed that developing highly-configurable systems is more difficult than developing single-variant software. A clear challenge is that highly-configurable software can only be handled one variant at a time by conventional software development tools (static analyzers, compilers, testing tools, etc). Using preprocessor (variability) just adds to this difficulty (cf. **Paper 1A**). Despite widespread adoption, preprocessors obfuscate the source code, reduce comprehensibility and increase error-proneness [84, 57].

In this paper, I report on a simple, but extensive experiment that investigates main properties of compilation warnings appearing in different configurations (and two different trees) of the Linux kernel project. I use sampling [88] across many configurations to investigate what kind of compilation warnings are most common, and configuration-dependent, to see in which subsystems they appear, and whether they are more likely to appear in unstable source trees, than in stable releases.

I use warnings as a proxy for unintended quality issues. Warnings are undesirable in mature code; it is a common practice to disallow code with compilation errors from being committed. Maintainers see warnings as a heuristic indication of low-quality code. Unfortunately, eliminating warnings from highly-configurable code is difficult, because compilers only report them for one configuration at a time. Also, warnings are often produced using the same, or very similar, static analysis techniques as used for detecting errors. Quantitative characteristics of warnings are thus interesting for tool builders, who work on family-based static analysis tools. First, they show that evaluating analysis tools on systems of the size of the Linux kernel using sampling is feasible. Second, they show what kind of warnings appear frequently, which allows to scope and prioritize work on lifted analyses for these warnings (so that sampling is not necessary, and warnings can be produced with higher reliability).

Warning statistics can be efficiently collected for a large number of configurations. The very small number of compilation errors, especially in stable releases, makes designing quantitative studies difficult. Frequency of warnings is higher than of errors. Moreover, compilation errors are often caused not by mistakes, but by deficiencies of the build environment (for instance absence of configuration-dependent dependencies). Since classifying the cause of error cannot be done automatically, collecting error distribution data automatically is difficult, while it is entirely feasible for warnings.

Therefore, the objective of this study is to quantitatively analyze configuration-dependent warnings in the Linux kernel by checking a large number of randomly generated configurations.

Method

This paper follows a three-step method: First, generation of random configurations. Second, collecting the warning messages returned by a compilation. Third, reflecting on the aggregated data to gather the findings. I discuss details of these steps below.

Generation of Random Configurations. I generate random configurations using `randconfig`, a built-in facility of the Linux kernel build system. It produces a file with configuration (so called `.config` file), with values for all features decided. This file is an input for the build system.

As a rule, generating uniformly distributed solutions to a constraint system over 15 thousand variables is not feasible with state of the art PSAT tools, which leaves `randconfig` as a viable approximation. While (possibly) not uniformly distributed, `randconfig` is a reproducible mechanism. The `randconfig` mechanism guarantees generating valid configurations, and thus so very efficiently (within seconds).

Compiling and Data Collection. I compile each generated configuration using the `make all` command with the GCC compiler, activating all warnings (the `-Wall` option), which is a common and recommended practice in open source development in C. Then, I collect the warning messages. A warning output contains a *bug type*, a *filename*, *line number*, and a *message* describing the warning.

The experiment is repeated for two different versions of the Linux kernel: a latest stable, v4.1.1, and a two months old in-development from the linux-next tree.[10] The random configurations are selected from the `x86` architecture of the Linux kernel, which decreases the amount of technical problems with building them on a single machine.

Data Analysis. After collecting all the data, I analyze the warning messages, classifying them by type and location (sub-system).

[10] next-20150402-22
`https://git.kernel.org/cgit/linux/kernel/git/next/linux-next.git`

#	Warning	Percent
1	unused-function	59 %
2	maybe-uninitialized	45 %
3	unused-variable	29 %
4	uninitialized	19 %
5	pointer-to-int-cast	17 %
6	frame-larger-than=	14 %
7	array-bounds	11 %
8	return-type	8 %
9	int-to-pointer-cast	8 %
10	overflow	7 %
11	implicit-function-decl	6 %
12	unused-label	5 %
13	deprecated-declarations	5 %

Figure 4.9: Most common warnings in the *stable* Linux Kernel (according to how many percent of configurations produce the given kinds of warnings, when compiled).

Operation. The experiment has been carried out for a month on two machines, a 32 core, 2.8MHz, 128GB RAM server (average time to generate and compile one configuration of around 1 minute and 35 seconds) and a conventional laptop with a 4 core, 2.5 MHz CPU and 4 GB of RAM.

Results

I now present the results of compiling 42,060 kernels (21K in a *stable* version and 21K in an *unstable* version) of Linux using GCC with all warnings enabled. All compilations produced a total of 400,000 warnings (i.e., an average of about ten warnings per compilation). The highest number of warnings produced by a single compilation was 111 (warnings) and 226 configurations compiled without warnings. Obviously, many of the same warnings are found over and over because the same code base in which they occur is included in many different configurations. The experiment materials are available online[11] (including scripts, notes, and reports).

Most Common Warnings

> OBSERVATION 1A: *The most common warnings in the* stable *Linux kernel involve* dead code *(warnings:* `unused-function`

[11] `https://github.com/models-team/quantify_linux_errors`

> *and* `unused-variable`*) and* uninitializations *(warnings:* `uninitialized` *and* `maybe uninitialized`*).*

Figure 4.9 shows the most common warnings occuring in the *stable* version of Linux. A warning is said to *occur* in a configuration of Linux, if a corresponding warning message appears as a result of compiling the configuration of Linux. The most abundant warning is `unused-functions`; functions that are declared, but not used. Such functions are technically *dead code*, but do occupy memory. In fact, Gcc does *not* remove dead code such as *unused functions* and *unused variables*.[12] To remove *dead code*, the gcc compiler needs to be invoked with options `-fdata-sections` and `-ffunction-sections` in order to keep data and functions in separate sections; subsequently, the *linker* needs the `-gc-sections` flag to be able to finally remove unused sections. Interestingly, the removal of such dead code is *not* available among the optimizations commonly employed in the Linux kernel; it is included in *neither* `-O0` (optimization level *zero*), `-O1` (*one*), *nor* -O2 (*two*). Note that Linux ubiquitously runs on many small embedded devices such as TiVo and similar DVR devices, network routers, and smartwatches —even credit-card sized single-computer boards such as Raspberry Pi—where memory is a limited resource.

The second most common warning, `maybe-uninitialized`, occurs in 45% of configurations; it occurs whenever gcc determines the existence of *an* execution path from a variable declaration to a usage without prior initialization. If the uninitialization occurs along *all* paths, the warning is strengthened to a (definitely) `uninitialized` which happens in 19% of configurations. Such warnings are quite serious. The third most common warning is `unused-variable` (occurring in 29% of configurations); variables that are declared, but not used. Such variables constitute *dead code* and will take up space on the heap or stack and may thus translate to wasted memory. In total, Figure 4.9 details the frequencies of 13 types of warnings commonly occurring in Linux.

In terms of variability, it is interesting to note that none of the warnings are at 100% which would be the case for configuration-independent warnings. This means that all warnings in the stable Linux seem dependent on configurations (choice of enabled/disabled features). Hence:

> <u>Observation 1b</u>: *All warnings in Linux appear to be* configuration-dependent *(i.e., warnings that occur in some configurations and* not *in others).*

[12] `https://gcc.gnu.org/ml/gcc-help/2003-08/msg00128.html`

#	Subsystem Directory	Absolute Size	Relative Size	Warning Percentage
1	drivers/	7,713	59 %	64 %
2	include/	423	3 %	40 %
3	crypto/	69	1 %	17 %
4	fs/	831	6 %	14 %
5	net/	631	5 %	10 %
6	arch/x86/	235	2 %	9 %
7	lib/	74	1 %	9 %
8	mm/	68	1 %	8 %
9	kernel/	155	1 %	6 %
10	sound/	659	5 %	4 %
11	block/	24	0 %	1 %
12	security/	50	0 %	0 %

Figure 4.10: Rank of subsystems in the *stable* Linux Kernel (according to how many percent of configurations produce warnings, when compiled). Size is given in KLOC.

Finally, I observe that:

> Observation 1c: *Most configurations appear to contain warnings.*

In fact, among our 21,030 configurations compiled in the *stable* version, only 226 did not produce warnings. Even though the Linux kernel developers try to improve code quality[13] —making sure that the code follows the coding style, and eliminating the static code checker errors and warnings—, it seems that dealing with variability (i.e., maintaining thousands of features and their interactions) is complicated. This is consistent with the results in **Paper 1A** and **Paper 1B** discussed before, which also show that variability makes reasoning about programs more difficult.

Subsystems with Most Warnings

> Observation 2: *The* `drivers/` *subsystem and* `include/` *header files produce warnings in around* half of all configurations*; whereas* core *subsystems such as* `kernel/` *and* `security/` *rarely produce warnings.*

Figure 4.10 shows the frequency of warnings in the *stable* version of Linux, according to the *subsystems* in which they occur. (Note that I use directories as proxies for *subsystems*.) It presents both absolute and

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

Warning	Stable	In-Dev.
unused-variable	29 %	51 %
int-to-pointer-cast	8 %	25 %
implicit-function-decl	6 %	23 %
frame-larger-than=	14 %	8 %

(a) Kinds of warnings.

Subsystem	Size	Stable	In-Dev.
arch/x86/	235	9 %	14 %
mm/	68	8 %	13 %
kernel/	155	6 %	3 %
sound/	659	4 %	2 %

(b) Subsystems with most warnings.

Figure 4.11: Significant differences of configuration-dependent warnings (in percentage) between the *stable* and *in-development* version of Linux.

relative size for each subsystem, in which I normalize the subsystem size by dividing it by the total size (= 13 MLOC). A warning is said to *occur* in a subsystem, if a corresponding warning message appears designating a location within the given subsystem. In the following, I disregard *smaller* subsystems below ten thousand lines of code: `virt/` (6.8 KLOC), `ipc/` (6.4k), `init/` (2.0k), and `usr/` (0.6k). I also disregard Linux *infrastructure* such as `tools/` (102k), `scripts/` (44k), and `samples/` (2.1k).

The subsystem most frequently producing warnings is also the largest subsystem of Linux with 7 MLOC: `drivers/`. This subsystem produces warnings in more than half (64%) of configurations. The subsystem `include/` (a directory with header files) causes warnings in almost half (40%) of configurations.

In particular, this means that sampling using `randconfig` is likely to hit warnings in `drivers/` and `include/`, but unlikely to hit warnings in core subsystems such as `kernel/` and `security/`. This is important as sampling is often proposed as a viable method for analysis of configurable systems. In case of using another sampling technique, the result is still unclear (as the probability distribution of `randconfig` is hard to describe).

Stable vs. In-Development Version

Figure 4.11 shows a comparison of differences in warnings in the *stable* versus the *in-development* version of Linux. Figure 4.11a charters significant differences in frequencies of warning kinds in the two version.

Variables declared, but not used (`unused-variables`) proliferate in the *in-development* version, occurring in twice as many configurations as that of the *stable* version. Presumably, the lack of rigorous testing that the *stable* version undergoes before "release" does not reveal such superfluous declarations. Also, (integer to pointer) *type casts* and *implicit function declarations* seems to occur more frequently in configurations in the (*un-stable*) development version. Only one kind of warning, `frame-larger-than=`N, occurs less frequently in the *in-development* version. This kind of warning is reported when a function seems to require allocation of more memory on the stack than a compile-time given constant, N. The remaining kinds of warnings do not spawn significant differences between the two versions.

Figure 4.11b shows differences in the locations of warning among the two versions of Linux. The subsystems that has a percentage point difference lower than 2% are not shown. As we can see, warnings occur more frequently in the `arch/x86/` and `mm/` (memory management) subsystems. Also, I observe, not surprisingly, that:

> OBSERVATION 3: *There appear to be more warnings in the* in-development *version than the* stable *version of Linux (especially,* unused variables, type casts*, and* implicit function declarations*).*

This is interesting for two reasons. First, it shows that developers do fix many problems before the code becomes stable, and if they had the right tools, this process could possibly be speeded up. In fact, official Linux kernel patch submission guidelines encourage removing warnings to avoid clutter of messages from the compiler.[14] The Linux foundation also urges the developers to heed the warnings produced by the compiler:[15]

> *"Contemporary versions of GCC can detect (and warn about) a large number of potential errors. Quite often, these warnings point to real problems. Code submitted for review should, as a rule, not produce any compiler warnings. When silencing warnings, take care to understand the real cause and try to avoid "fixes" which make the warning go away without addressing its cause."*

Second, it also shows that the errors survive all the way to the stable version, so the variability-aware tools that would be more precise or more

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

[15] `https://www.linuxfoundation.org/content/how-participate-linux-community-0`

accurate than the developers, could help to substantially decrease the bug density in Linux.

Overall, it seems that Linux developers are good at fixing issues in in-development versions for stable version releases.

Contributions

Similarly to **Paper 2A**, this paper contributes to **Goal 2** since it investigates characteristics of configuration-dependent warnings in Linux, which are proxies for bugs.

The results show 13 different types of warnings appearing in the Linux kernel. The majority of these are regarding *dead code* (e.g., unused variable) and *uninitializations*. This finding aligns with the study of Medeiros *et al.* [66] in which they found 39 configuration-related issues, including 14 (36%) undeclared functions, 2 (5%) undeclared variables, 7 (18%) unused functions, and 23 (41%) unused variables. In other words, they noticed that 59% of the issues are related to unused functions and variables. Here, I complement this finding by generating randomly thousands of configurations from the Linux kernel, in which I observe that the most abundant warning is unused function and the third most is unused variable.

I also find that most configurations (and subsystems) appear to contain warnings. Specifically, the `drivers/` and `include/` subsystems contain warnings in about half of all configurations, whereas much fewer warnings originate from the *core* subsystems `kernel/` and `security/`. This result harmonizes with **Paper 2A** and with the study of Abal *et al.* [2] by confirming and complementing with quantitative data that variability bugs are not confined to any particular type of bug, (error-prone) feature, or source code location. Additionally, I observe that there are generally more warnings in the in-development version of Linux than in the stable version.

Finally, I hope that indication of which areas are hot in warnings can be useful for further research on bug finding in Linux, showing which subsystems are good candidates as analysis subjects. Also, this study seems to confirm the, commonly held but rarely followed, recommendation to focus bug-finding studies on unstable trees of Linux, as opposed to stable. Moreover, this study shows that the errors survive all the way to the stable version, so the variability-aware tools could help to substantially decrease the bug density in Linux.

Chapter 5

Variability-Aware Solution for Lifting Single-Program Analysis

This chapter summarizes **Paper 3A** which contributes to **Goal 3** and, consequently, to **Q3**. In the following, I address the solution-program quadrant (see Table 5.1), where I provide evidence on how to *lift* conventional single-analysis tools to handle program families implemented using preprocessor directives.

	Problem Space	Solution Space
Programmer Perspective	– Quadrant 1 – P1: No hard evidence on how variability affects programmers. Q1: How does variability affect programmers on bug finding? T1: Variability increases the bug-finding time and makes it difficult to identify the erroneous configurations.	– Quadrant 4 – P4: Programmers fail to reason about configurations. Q4: How to support programmers to reason about configurations? T4: Beyond the scope of this dissertation. (Chapter 7 provides a sketch of one possible solution.)
Program Perspective	– Quadrant 2 – P2: Lack of evidence on how variability affects bugs. Q2: How does variability affect bugs? T2: Variability increases the complexity of bugs; and these (variability) bugs are not confined to any type of bug, feature, or location.	– Quadrant 3 – P3: Infeasible to lift all program analyses to deal with variability. Q3: How to lift conventional analysis tools to find variability bugs? T3: Rewriting variability enables single-program analysis tools to find bugs in highly-configurable systems.

Table 5.1: Research problem, question, and thesis of the Quadrant 3.

5.1 Effective Analysis of C Programs by Rewriting Variability (Paper 3A)

Summary

This paper proposes a series of variability transformations for translating program families into single programs by replacing compile-time variability with run-time variability (non-determinism). The obtained transformed programs can be subsequently analyzed using the conventional off-the-shelf single-program analysis tools such as type checkers, symbolic executors, model checkers, and static analyzers. In particular, our[1] variability transformations are *outcome-preserving*, which means that the relation between the outcomes in the transformed single program and

[1] This is a joint work with Alexandru F. Iosif-Lazar and Aleksandar S. Dimovski.

the union of outcomes of all variants derived from the original program family is *equality* in general.

As a proof of concept, the paper presents C Reconfigurator, a prototype tool that implements variability transformations for the C language. All transformations are implemented using Xtend.[2] The C Reconfigurator tool is available online at `https://github.com/models-team/c-reconfigurator`. It calls SuperC [39], a variability-aware parser, to parse code with preprocessor annotations, which uses Binary Decision Diagrams (BDD's) for encoding feature expressions and for decisions during the parsing process. SuperC returns an Abstract Syntax Tree (AST) with variability, in which variability is reflected with choice nodes over feature expressions [37]. A choice node is a node with two children, such that the left child of the choice node is included in the result of those configurations for which the given feature expression is satisfied; the right child of the choice node is included otherwise. We apply variability transformation rules to the AST with variability obtaining an ordinary AST, which is subsequently translated into a single C program (pretty printed).

We show the transformation rules and discuss their correctness with respect to a minimal core imperative language IMP. We also discuss our experience of implementing and using the transformations for efficient and effective analysis and verification of real-world C program families. We report on some interesting variability bugs that we were able to detect using various state-of-the-art single-program analysis tools, such as Frama-C [56], Clang [21] and LLBMC [69].

Context & Motivation

Due to the increasing popularity of program families, formal verification techniques for proving their correctness are widely studied [88]. Analyzing program families is challenging (cf. **Paper 1A**). With very few compile-time configuration options, exponentially many variants of a system can be derived. Thus, for large variability-intensive software systems, a brute-force approach that derives and analyzes all variants individually one by one using existing single-program analysis tools is infeasible.

Recently, many dedicated family-based (variability-aware) analysis tools have been developed, which operate directly on program families. They

[2] `http://www.eclipse.org/xtend/`

$$\begin{tikzcd}
\text{Variability program} \arrow[r, "\textit{transform}"] \arrow[d, dotted, "\textit{variability-aware}"', "\textit{analyzer}"] & \text{Single program} \arrow[d, "\textit{single-program}"', "\textit{analyzer}"] \\
\text{Variability-aware results} & \text{Single-program results} \arrow[l, "\textit{interpret results}"']
\end{tikzcd}$$

Figure 5.1: The overview of our transformation-based approach for verification of program families. The single-program analyzer can be any verification oracle for single programs, such as: symbolic executor, type checker, static analyzer, model checker.

produce results for all variants at once in a single run by exploiting the similarities between the variants. Examples of successful family-based analysis tools are applied to syntax checking [1, 39], type checking [51, 20], static analysis [14, 13], model checking [22, 30], etc. Although they are more efficient than the brute-force approach, still their design and implementation for each particular analysis and language is tedious and error prone. Often, these family-based tools are research prototypes implemented from scratch. So, it is very difficult to re-implement all optimization algorithms in them that already exist for their single-program industrial-strength counterparts, which have been under development for several decades.

Another approach for efficient variability-aware verification would be to replace compile-time variability with run-time variability (or non-determinism), a method also known as "variability simulation" [92]. In particular, in this work we consider a class of variability transformations that transform a program family into a single program, whose outcomes are equal to the union of all outcomes of individual variants. We call the corresponding transformations outcome-preserving. Subsequently, existing single-program analysis tools (verification oracles) that can handle non-determinism (run-time variability) can be used to analyze the generated single program. Finally, the obtained results are interpreted back on the individual variants. The overview of this approach is given in Figure 5.1. Instead of using specialized variability-aware tools to analyze program families (which would be tedious and labor intensive), we use the standard off-the-shelf single-program analysis tools to achieve the same goal by employing our variability transformations.

```c


int foo()
{
  int x = 1;

  #ifdef A
  x = x + 1;
  #endif
  #ifdef B
  x = x - 1;
  #endif

  return 2/x;
}
```

```c
int A = rand() % 2;
int B = rand() % 2;
int foo()
{
  int x = 1;

  if(A)
    x = x + 1;

  if(B)
    x = x - 1;


  return 2/x;
}
```

Figure 5.2: Before (left) and after (right) our transformations.

In the following, I illustrate how our variability transformations work on C program families. Consider a preprocessor-based family of C programs shown in Figure 5.2 (left column), which uses two (Boolean) features A and B. Only two features give rise to a family of four program variants defined by the set of configurations $\mathbb{K} = \{A \wedge B, A \wedge \neg B, \neg A \wedge B, \neg A \wedge \neg B\}$ (assuming none are deemed invalid). For example, the variant for $A \wedge B$, which has both features A and B enabled (set to true), and the variant for $\neg A \wedge \neg B$ are, respectively:

```c
int foo()
{
  int x = 1;
  x = x + 1;
  x = x - 1;
  return 2/x;
}
```

```c
int foo()
{
  int x = 1;


  return 2/x;
}
```

Figure 5.3: Variants: $A \wedge B$ (left) and $\neg A \wedge \neg B$ (right).

In such program families, errors (also known as *variability bugs* [2]) can occur in some variants (configurations) but not in others. In our example, the variant $\neg A \wedge B$ will crash at the `return` statement when we attempt to divide by zero. At the same time, the other variants avoid the division-by-zero. The value of `x` at the `return` statement is 1 for variants $A \wedge B$ and $\neg A \wedge \neg B$, and 2 for $A \wedge \neg B$.

To detect the variability bugs, we would either have to analyze each variant individually, or to use a family-based analysis tool that can parse and process the `#ifdef` and `#endif` directives accordingly. We instead transform the code in such a way that it can be analyzed by tools that cannot handle C Preprocessor directives. Figure 5.2 (right column) shows a single program obtained by applying our transformation on the family shown in the left part of the figure. All features are first declared as ordinary global variables and non-deterministically initialized to 0 or 1, then all `#ifdef` statements are transformed into ordinary conditional statements (`if`-s) with the same guard conditions. The division-by-zero is still present in this single program and happens when A is initialized to 0 (*disabled*) and B to 1 (*enabled*). The set of outcomes of the transformed program (Figure 5.2, right column) is equal to the union of outcomes of all individual variants from the family (Figure 5.2, left column).

In general, the transformed program that we obtain from the original program family can be analyzed by various single-program verification tools, in order to find variability errors or to confirm the absence of errors in the given program family.

Method

To demonstrate correctness of our transformations, we define them formally using IMP, a small imperative language. To model compile-time variability, we extend IMP with an "`#ifdef`" construct for encoding multiple variants, which we call $\overline{\text{IMP}}$ language. To encode run-time variability, we extend IMP with an "`or`" construct for encoding non-determinism, which we call IMPor language. We define transformations that translate any given $\overline{\text{IMP}}$ program into a corresponding IMPor program. In the paper, we show the relation between the semantics of the input and output programs for each transformation. In the following, I give an overview of the formal model for our transformations. (I refer the reader to the paper for a more detailed description on this.)

A Formal Model for Transformations

IMP is an imperative language with two syntactic categories: expressions and statements. Expressions include integer constants, variables, and binary operations. Statements include a "do-nothing" statement `skip`, assignments, statement sequences, conditional statements, while loops, and local variable declarations. Its abstract syntax is summarized using the following grammar:

$$\begin{array}{lll} e & ::= & n \;|\; \mathtt{x} \;|\; e_0 \oplus e_1 \\ s & ::= & \mathtt{skip} \,|\, \mathtt{x} \;\mathtt{:=}\; e \,|\, s_0 \;\mathtt{;}\; s_1 \,|\, \mathtt{if}\; e \;\mathtt{then}\; s_0 \;\mathtt{else}\; s_1 \,|\, \mathtt{while}\; e \;\mathtt{do}\; s \,|\, \mathtt{var}\; \mathtt{x}\mathtt{:=}e \;\mathtt{in}\; s \end{array}$$

In the above, n stands for an integer constant, x stands for a variable name, and $\oplus$ stands for any binary arithmetic operator.

The language IMPor is obtained by extending IMP with a non-deterministic choice operator 'or' which can non-deterministically choose to evaluate either of its arguments.

$$e \quad ::= \quad ... \;|\; e_0 \;\mathtt{or}\; e_1$$

With this non-deterministic construct 'or', it is possible for an expression to evaluate to a set of different values in a given store.

The programming language $\overline{\text{IMP}}$ also extends IMP (thus, $\overline{\text{IMP}}$ does not contain the 'or' construct). Its abstract syntax includes the same expression and statement productions as IMP, but we add the new compile-time conditional statements for encoding multiple variants of a program. The new statements "#if (ϕ) s #endif" and "#if (ϕ) var x:=n in #endif s" contain a feature expression $\phi \in \mathit{FeatExp}$ as a presence condition, such that only if ϕ is satisfied by a configuration $k \in \mathbb{K}$ then the code between #if and #endif will be included in the variant for k.

$$s ::= ... \;|\; \mathtt{\#if}\,(\phi)\; s\; \mathtt{\#endif} \;|\; \mathtt{\#if}\,(\phi)\; \mathtt{var}\; \mathtt{x}\mathtt{:=}n\; \mathtt{in}\; \mathtt{\#endif}\; s$$

A finite set of Boolean variables $\mathbb{F} = \{A_1, \ldots, A_n\}$ describes the set of available *features* in the program family. Each feature may be *enabled* or *disabled* in a particular variant. A *configuration* k is a truth assignment or a valuation which gives a truth value to each feature, i.e. k is a mapping from $\mathbb{F}$ to $\{\text{true}, \text{false}\}$. If a feature $A \in \mathbb{F}$ is enabled for the configuration k then $k(A) = \text{true}$, otherwise $k(A) = \text{false}$. We write $\mathbb{K}$ for the set of all *valid* configurations defined over $\mathbb{F}$ for a family. The set of valid configurations is typically described by a feature model [47], but in this work we disregard syntactic representations of the set $\mathbb{K}$. Note that $|\mathbb{K}| \leq 2^{|\mathbb{F}|}$, since, in general, not every combination of features yields a *valid* configuration. We define *feature expressions*, denoted *FeatExp*, as the set of well-formed propositional logic formulas over $\mathbb{F}$ generated using the grammar: $\phi ::= \text{true} \,|\, A \in \mathbb{F} \,|\, \neg\phi \,|\, \phi_1 \wedge \phi_2 \,|\, \phi_1 \vee \phi_2$.

The semantics of $\overline{\text{IMP}}$ has two stages: first, given a configuration k compute a single IMP program without #if-s; second, evaluate the obtained variant using the standard IMP semantics. The first stage is a simple *preprocessor* specified by the projection function π_k mapping an $\overline{\text{IMP}}$ program family into a single IMP program corresponding to the configuration k. The projection π_k copies all basic statements of $\overline{\text{IMP}}$ that are also in IMP, and recursively pre-processes all sub-statements of compound statements. For example, $\pi_k(\mathsf{skip}) = \mathsf{skip}$ and $\pi_k(s_0;s_1) = \pi_k(s_0);\pi_k(s_1)$. The second step is standard for IMP.

Results

First, I summarize our variability transformations. Then, I present the evaluation results.

Variability Transformations

The aim of our transformation-based approach is to rewrite an input $\overline{\text{IMP}}$ program family $\bar{s}$ into an output IMPor program $\overline{s'}$. In a pre-transformation phase, we first convert each feature $A \in \mathbb{F}$ into the variable A, which is non-deterministically initialized to 0 or 1 (i.e., false or true). Let $\mathbb{F} = \{A_1, \ldots, A_n\}$ be the set of available features in the family $\bar{s}$, then we have the following initialization fragment in the resulting pre-transformed program pre-t$(\bar{s})$:

$$\texttt{pre-t}(\bar{s}) = \texttt{var}\ A_1 := 0\ \texttt{or}\ 1\ \texttt{in} \ldots \texttt{var}\ A_n := 0\ \texttt{or}\ 1\ \texttt{in}\ \bar{s}$$

After translating features into variables, a series of rewrite rules are applied on the program. The rules have the following form: $\psi \vdash s \rightsquigarrow s'$, which means that if the current program family being transformed matches any abstract syntax tree (AST) node of the shape s nested under #if-s with the resulting presence condition that implies $\psi \in \mathit{FeatExp}$ (i.e., in context ψ) then *replace s by s'*. Formally, applying the rule $\psi \vdash s \rightsquigarrow s'$ to a family:

$$\ldots \texttt{\#if}\ (\phi_1) \ldots \texttt{\#if}\ (\phi_n) \ldots; s; \ldots \texttt{\#endif} \ldots \texttt{\#endif} \ldots$$

where $\phi_1 \wedge \ldots \wedge \phi_n \implies \psi$, then results in the transformed program:

$$\ldots \texttt{\#if}\ (\phi_1) \ldots \texttt{\#if}\ (\phi_n) \ldots; s'; \ldots \texttt{\#endif} \ldots \texttt{\#endif} \ldots$$

The function $Rewrite(\overline{s}, \psi \;\vdash\; s \rightsquigarrow s')$ represents the final transformed program $\overline{s'}$ obtained by repeatedly applying the rule $\psi \;\vdash\; s \rightsquigarrow s'$ on $\overline{s}$ and its transformed versions until a point where this rule can not be applied is reached (a fixed point of the rule).

In the following, I present two transformation rules which handle conditional variable declarations and uses. They involve duplicating code and variable renaming. The most straightforward way to handle renaming of variables in different contexts is by adding an *environment* δ as a parameter to the statements being transformed. We define an environment $\delta : \mathit{Var} \times \mathit{FeatExp} \rightarrow \mathit{Var}$ as a function mapping a given pair of a variable and a feature expression to a variable name. We write $\delta^{\mathtt{fe}}(\mathtt{x}) \subseteq \mathit{FeatExp}$ for the set of all feature expressions ϕ such that $\delta(\mathtt{x}, \phi)$ is defined, i.e. $\delta^{\mathtt{fe}}(\mathtt{x}) = \{\phi \in \mathit{FeatExp} \mid (\mathtt{x}, \phi) \in \mathit{dom}(\delta)\}$. We write s, δ to denote the result of simultaneously substituting $\delta(\mathtt{x}, \phi)$ for each occurrence of any variable x in s in the context (presence condition) that implies ϕ.

Conditional variable declaration. Figure 5.4 (left) shows a variable declared with different types depending on the features A and B, and how it is rewritten to include both versions in a single program (right side).

```
#if (A) int x;  #endif
#if (B) long x; #endif
```

```
int x_A;
long x_B;
```

Figure 5.4: Rewriting conditional variable declarations.

This rule transforms a local variable that is declared conditionally within a given context $\psi \in \mathit{FeatExp}$:

$$\psi \;\vdash\; \mathtt{\#if}\ (\phi)\ \mathtt{var\ x{:=}}n\ \mathtt{in\ \#endif}\, s, \delta \;\rightsquigarrow\; \mathtt{var}\ \mathtt{x}_{new}\mathtt{{:=}}n\ \mathtt{in}\ s, \delta[(\mathtt{x}, \phi) \mapsto \mathtt{x}_{new}] \tag{5.1}$$

where $\mathtt{x}_{new}$ is a fresh variable name that does not occur as a free variable in s. The square brackets in δ (cf. $\delta[(\mathtt{x}, \phi) \mapsto \mathtt{x}_{new}]$) means that the environment is update if the variable already exists in the mapping, otherwise a new entry is created in the environment.

Conditional variable use. Figure 5.5 contains variable x that is declared conditionally when A and !A, two mutually exclusive presence conditions. At its usage, we rewrite it by introducing a ternary conditional operator in order to split the execution and retain the original paths.

```
#if (A)  int x;  #endif
#if (!A) long x; #endif
y = x + 1;
```

```
int x_A;
long x_notA;
y = (A ? x_A : x_notA) + 1;
```

Figure 5.5: Rewriting conditional variable use.

This rule handles the case when a local variable is used within a context $\psi \in \mathit{FeatExp}$. In other words, it transforms expressions whose variables depend on features. There are three cases to consider here.

$$\psi \ \vdash \ \texttt{y:=}e[\mathrm{x}], \delta \rightsquigarrow \texttt{y:=}e[\delta(\mathrm{x}, \phi)], \delta \tag{2.1}$$

if there exists an unique $\phi \in \delta^{\texttt{fe}}(\mathrm{x})$, such that $\psi \models \phi$. Here $e[\mathrm{x}]$ means that the variable x occurs free in the expression e.

The second case is when there are several $\phi_1, \ldots \phi_n \in \delta^{\texttt{fe}}(\mathrm{x})$, such that $\texttt{sat}(\phi_1 \wedge \psi), \ldots, \texttt{sat}(\phi_n \wedge \psi)$:

$$\psi \vdash \texttt{y:=}e[\mathrm{x}], \delta \rightsquigarrow \texttt{\#if } (\phi_1) \ \texttt{y:=}e[\delta(\mathrm{x}, \phi_1)] \ \texttt{\#endif;} \ldots \texttt{\#if } (\phi_n) \ \texttt{y:=}e[\delta(\mathrm{x}, \phi_n)] \ \texttt{\#endif,} \delta \tag{2.2}$$

Otherwise, meaning that for all $\phi \in \delta^{\texttt{fe}}(\texttt{x})$ it follows that $\texttt{unsat}(\phi \wedge \psi)$, we have:

$$\psi \ \vdash \ \texttt{y:=}e[\mathrm{x}], \delta \rightsquigarrow \texttt{y:=}e[\mathrm{x}], \delta \tag{2.3}$$

I refer to the paper for the other rules, including a few normalization rules, as well as the proof of the theorem of outcome preservation for IMP programs. The theorem also holds for C programs in the subset of C corresponding to IMP.

Evaluation

We evaluate our reconfiguration technique based on variability transformations and single-program verification oracles on several real-world C case studies.[3] The evaluation aims to show that we can use state-of-the-art single-program verification tools to verify realistic C program families using variability transformations. To do so, we ask the following research questions:

RQ1: How precise is our technique?

[3] All experiment materials are available online at `https://github.com/models-team/c-reconfigurator-test`.

RQ2: How efficient is the verification oracle to identify variability bugs after transforming the code using our technique?

In particular, we want to show that single-program verification tools able to find bugs in variant code (erroneous configurations), are also able to find the same bugs in reconfigured code, which are obtained using our tool. We use FRAMA-C [56], CLANG [21] and LLBMC [69] as our verification oracles. FRAMA-C is a framework for modular static (dataflow) analysis of C programs. The CLANG project includes the Clang compiler front-end and the Clang static analyzer for several programming languages, including C. LLBMC (the low-level bounded model checker) is a software model checking tool for finding bugs in C programs. We study only *known* variability bugs that are detectable by the three tools.

Subject Files and Experimental Setup. All transformations are applied using the C RECONFIGURATOR tool. We investigate precision and performance in finding real variability bugs extracted from three benchmarks: Linux, BusyBox and Libssh. In particular, we use simplified bugs from the VDBb database that are found in the Linux kernel files and in BusyBox (cf. **Paper 2A**). Simplified bugs are independent of the kernel code and the corresponding programs were derived systematically from the error trace. Additionally, we use real variability bugs from Libssh.

Table 5.2 presents the characteristics of the subject files we analyzed in our empirical study. We list: the file id, bug type, number of features ($|\mathbb{F}|$), number of valid configurations ($|\mathbb{K}|$), lines of code, the size in KB of the files before (with `#ifdef`-s) and after (without `#ifdef`-s) our transformations, and commit hash (clickable) for each project. This collection consists of a diverse set of bug types such as null pointer dereferences, buffer overflow, and uninitialized variable. In total, we have 11 distinct kinds of bugs. The number of features per file varies from one to seven. In addition, the number of lines of code ranges from 12 to 165 for the simplified files (from VBDb), and from 1404 to 2959 for real files (from Libssh). After the transformation, the biggest increase in size of almost 8 times can be observed for FILE ID 7. This is due to the fact that this file has seven different features and several variability patterns that depend on them. In most of the other cases the size increase is not very big.

All experiments were executed on a Kubuntu VM (64bit, 4 CPUs), Intel®Core^{TM} i7-3720QM CPU running at 2.6GHz with 12GB RAM memory. The performance numbers reported constitute the median runtime of 50 independent executions.

File	Bug type	$\lvert\mathbb{F}\rvert$	$\lvert\mathbb{K}\rvert$	LOC	Size KB		Hash
id					before	after	
VBDb Linux files							
1	null pointer deref.	5	24	165	2.9	4.3	76baeeb
2	null pointer deref.	3	6	112	1.9	2.5	f7ab9b4
3	null pointer deref.	4	8	55	0.9	1.0	ee3f34e
4	null pointer deref.	3	6	34	0.5	0.6	6252547
5	buffer overflow	1	2	58	1.0	1.2	8c82962
6	buffer overflow	1	2	33	0.6	0.7	60e233a
7	read out of bounds	7	63	69	1.1	8.4	0f8f809
8	uninitialized var.	2	4	54	0.8	1.0	7acf6cd
9	uninitialized var.	1	2	54	1.0	1.1	bc8cec0
10	uninitialized var.	1	2	53	0.8	1.0	30e0532
11	uninitialized var.	2	4	38	0.9	1.2	1c17e4d
12	uninitialized var.	2	4	26	0.3	0.5	e39363a
13	undefined symbol	4	14	25	0.4	0.6	7c6048b
14	undefined symbol	2	4	20	0.3	0.5	2f02c15
15	undefined symbol	2	4	20	0.3	0.5	6515e48
16	undefined symbol	2	4	19	0.3	0.5	242f1a3
17	undeclared identifier	3	8	37	0.6	1.0	6651791
18	undeclared identifier	2	4	20	0.3	0.4	f48ec1d
19	wrong # of args	1	2	12	0.2	0.4	e67bc51
20	multiple funct. defs	2	4	21	0.3	0.8	e68bb91
21	dead code	1	2	19	0.2	0.3	809e660
22	incompatible type	2	4	27	0.4	0.7	d6c7e11
23	assertion violation	2	4	79	1.5	1.8	63878ac
24	assertion violation	2	4	75	1.1	1.2	657e964
25	assertion violation	2	4	41	0.6	0.7	0988c4c
VBDb BusyBox files							
26	null pointer deref.	1	2	28	0.4	0.7	199501f
27	null pointer deref.	2	4	24	0.4	0.6	1b487ea
28	uninitialized var.	2	4	28	0.4	0.7	b273d66
29	undefined symbol	1	2	42	0.8	0.9	cf1f2ac
30	undefined symbol	2	4	27	0.4	0.6	ebee301
31	undeclared identifier	1	2	35	0.5	0.8	5275b1e
32	undeclared identifier	1	2	19	0.3	0.4	b7ebc61
33	incompatible type	3	8	46	0.9	1.5	5cd6461
Real Libssh files							
34	null pointer deref.	6	48	1404	34.8	32.6	0a4ea19
35	null pointer deref.	4	4	1428	44.1	31.9	fadbe80
36	uninitialized var.	3	4	2959	72.4	77.6	2a10019

Table 5.2: Characteristics of the benchmark files.

ID	Frama-C				
	buggy variant		reconfigured		all
	y/n	time	y/n	time	time
VBDb Linux files					
1	✓	218	✓	235	5602
2	✓	220	✓	225	1394
3	✓	215	X	236	1918
4	✓	218	✓	224	1379
5	✓	218	✓	227	488
6	✓	213	✓	227	463
7	✓	218	✓	225	14381
8	✓	241	✓	250	918
9	✓	224	✓	230	462
10	✓	216	inc	224	460
11	✓	234	✓	224	917
12	✓	216	inc	227	914
13	✓	239	✓	248	3194
14	✓	237	✓	244	905
15	✓	224	✓	248	906
16	✓	213	✓	222	910
17	✓	216	✓	230	3823
18	✓	210	✓	224	901
19	✓	210	✓	224	452
20	✓	213	X	228	907
21	✓	239	X	240	458
VBDb BusyBox files					
26	✓	230	✓	234	484
27	✓	224	✓	234	959
28	✓	237	inc	237	957
29	✓	230	✓	236	481
30	✓	231	✓	228	968
31	✓	220	✓	228	486
32	✓	216	✓	224	477

(a) VBDb files using Frama-C.

ID	Clang/LLBMC				
	buggy variant		reconfigured		all
	yes/no	time	yes/no	time	time
VBDb Linux files					
22	✓	21	✓	23	91
23	✓	4	✓	10	10
24	✓	3	✓	7	11
25	✓	3	✓	5	8
VBDb BusyBox files					
33	✓	27	✓	31	222

(b) VBDb files using Clang (files 22 and 33) and LLBMC (files 23, 24, and 25).

ID	Clang/LLBMC				
	buggy variant		reconfigured		all
	yes/no	time	yes/no	time	time
34	✓	1526	✓	1702	17029
35	✓	1591	✓	1804	5917
36	✓	112	✓	144	448

(c) Libssh files using Clang (file 36) and LLBMC (files 34 and 35).

Table 5.3: Verification results for the benchmark files. Times in milliseconds (ms).

Simplified files. Table 5.3a shows the results of verifying our benchmark files which contain known bugs by using Frama-C. The table has three main columns: buggy variant, reconfigured, and all that depict the tool results on the buggy variant code, on the reconfigured program family code, and on all valid variants from $\mathbb{K}$ analyzed one by one (in a brute force fashion). Each column contains two results. The first is a checkmark (✓), which means that the same bug was found in both the buggy variant and reconfigured program by the verification tool. The checkmark is replaced by X if the bug was not found, or *inc*—inconclusive which means that FRAMA-C was able to detect a bug in the reconfigured program, but a different one from the bug found in the buggy variant. The second result in each column is the analyzer time. In the case of brute force approach (ALL), we include the analysis time of all valid variants regardless of whether they contain a bug or not.

We observe that C RECONFIGURATOR tool transforms the family code by preserving the erroneous traces from the buggy variant in most cases. For instance, FRAMA-C could detect 22 (78%) bugs from the simplified benchmark files (28 in total) after reconfiguring the files using our tool. Besides that, the C RECONFIGURATOR preserves a variety of bug types such as buffer overflow and uninitialized variable. It is worth noting that there is a trade-off between BUGGY VARIANT, which requires luck to hit the exact erroneous configuration among the entire set of valid configurations, and ALL, which is slow and thus infeasible to analyze the entire program family. This way, the main benefit of our technique is that we are able to speed-up the analysis of all variants (versus brute-force), but also the analysis tools are more likely to find a bug in reconfigured code (versus generating random configurations).

However, the success rate of our transformation depends on the tool which may or may not detect different types of bugs. For example, our technique is able to transform a file containing a memory leak error, but FRAMA-C does not have any analysis to identify it. In three specific cases (cf. FILE IDS 10, 12 and 28), FRAMA-C did not report the original bug as an error, but it did detect that some variable might be uninitialized in some conditions. This happens because FRAMA-C performs a *may* value analysis for finding uninitialized variables. A *may* analysis describes information that may possibly be true along one path to the given program point and, thus in our case, computes a superset of all uninitialized variables in all variants. So the reported variable may not match with the one in the buggy variant. We marked these three cases as *inc*—inconclusive in the table. Still the verification oracle reports that there might be an error in the reconfigured code.

In addition, the verification tool could not identify the required bug in the reconfigured file in three occasions (cf. FILE IDS 3, 20 and 21). For ex-

```c
int do_sect_fault()
{
  return 0;
}

int main()
{
  #ifndef ARM
  do_sect_fault();
  #endif
  return 0;
}
```

```c
int do_sect_fault()
{
  return 0;
}

int main()
{
  if(!ARM)
    do_sect_fault();

  return 0;
}
```

Figure 5.6: File 21 - Before (left) and after (right) our transformations

ample, file 21 contains dead code, which is a function (`do_sect_fault()`) that is never called when feature ARM is enabled (see the code snippet in Fig. 5.6, left column). The C Reconfigurator transforms the code by changing the `#ifdef` into ordinary `if` condition, making the function available for the transformed single program (i.e., the function is not dead any more), as shown in the code snippet in Fig. 5.6 (right column). The other two cases are similar to this one in the sense that the C Reconfigurator makes feature code explicit to the entire program family.

Generally speaking, if one variant does not use a variable/function, but another does, then the reconfigured code will use the variable/-function and the error will be hidden (like in the example above). This happens due to the limitations of variability encoding, especially because we cannot preprocess the reconfigured code to filter out the irrelevant features for a particular variant. In a reconfigured code, all variants are encoded as a single program (see **Paper 3A** for more discussion).

Let us now consider the remaining simplified files. We use Clang and LLBMC to analyze only the other types of bugs (incompatible type and assertion violation) that Frama-C cannot handle. We treat Clang and LLBMC as one verification oracle, since we first need to compile and emit llvm code with Clang in order to analyze it using LLBMC. So, we do not make difference in reporting whether the bug was found by Clang during the compilation or afterwards by LLBMC.

Table 5.3b, similarly to Table 5.3a, shows the results of verifying both the buggy variant and the reconfigured code using Clang and LLBMC. We also report the analysis time of the brute force approach in the column

ALL. As we can see, all bugs were found by CLANG/LLBMC in the reconfigured version. We can thus confirm that our C RECONFIGURATOR tool transforms the family code by preserving the erroneous traces from the buggy variant. Based on analyzing 33 simplified variability bugs from Linux and BusyBox, we find that:

ANSWER RQ1 (PRECISION)

The C RECONFIGURATOR enables single-program verification tools such as FRAMA-C, CLANG, and LLBMC to **successfully** detect *most* of the simplified variability bugs on the reconfigured code, obtained from the Linux and BusyBox benchmark files.

We also evaluate performance of the verification tools to identify the given variability bugs. Tables 5.3a and 5.3b show time needed for the verification tools to analyze the buggy variant code (BUGGY VARIANT column) and the reconfigured program family code (RECONFIGURED column). We can see that the analysis times in both cases are similar although reconfigured code is bigger in size (not terribly so). In fact, FRAMA-C takes less than half a second to analyze each file regardless of whether it is a variant or a reconfigured file. For instance, FRAMA-C analyzes file 1 in 218 and 235 milliseconds on the variant code and on the reconfigured program family code, respectively. File 1 contains a null pointer dereference and gives rise to 24 different configurations. Applying the brute force approach (ALL column), which analyzes all variants individually one by one, on this file using FRAMA-C takes 5,602 ms. In this way, we obtain significant speed-up to verify the program family using our technique. We also obtain similar results in terms of performance using CLANG/LLBMC (see Tables 5.3b and 5.3c). In general, the performance of analyzing a reconfigured code is similar to analyzing only one variant, which gives us a speed-up proportional to the number of valid variants of a program family. Overall, we observe that:

ANSWER RQ2 (PERFORMANCE)

The C RECONFIGURATOR speeds-up the family-based analysis via single-program verification tools, so that we can **efficiently** detect simplified variability bugs on the reconfigured code, obtained from the VBDb benchmark.

Real files. Lastly, we consider real files to confirm our previous observations with respect to precision and performance. Table 5.3c presents the results of analyzing three real files from the Libssh project using CLANG and LLBMC.[4] These files contain two types of bugs: null pointer dereference and uninitialized variable. Each file has at least three distinct features.

Table 5.3c shows that our C RECONFIGURATOR transforms the family code by preserving the erroneous traces from the buggy variant even for complex and large files. In fact, the verification tool (CLANG/LLBMC) found the same bug (from the buggy variant code) on the reconfigured code in all three cases. From this preliminary evidence, we thus confirm that our technique enables single-program verification oracles to successfully detect variability bugs on the reconfigured code, obtained from complex and real files.

Regarding performance, we can still see the similarity in verifying a variant code and a reconfigured one. For example, CLANG/LLBMC took 1,5 sec to analyze file 34 in the single variant version, whereas in the reconfigured version, the tool analyzed it in 1,7 sec. We can also observe a speed-up of the family-based analysis using the C RECONFIGURATOR and single-program verification tools by a factor of the number of valid variants compared to the brute force approach. Therefore, we conclude that:

CONCLUSION

The C RECONFIGURATOR enables analyses of programs at the coverage close to a brute force analysis, but with the efficiency of a single program analysis. It is also much easier to implement comparing the cost of reimplementing specialized tools such as FRAMA-C, CLANG, and LLBMC in a variability-aware fashion.

Contributions

This paper contributes mainly to **Goal 3** as propose variability transformations to translate program families into single programs without variability. Research question **Q3** asks for how we could *lift* single-analysis

[4] We do not report results from FRAMA-C on the real files because FRAMA-C could not handle them.

tools to find variability bugs. We define a series of variability transformations rules and prove their correctness with respect to a minimal core imperative language IMP.

The evaluation shows that the transformed programs can then be effectively and efficiently analyzed using various single-program analyzers, and that some interesting variability bugs can be found in realistic C programs by rewriting variability. Finally, I hope that our technique will be useful for future builders of analysis tools.

Chapter *6*

Related Work

In this chapter, I present work related to the main chapters of this dissertation (Chapters 3, 4, and 5). First, I discuss previous work on studying *programmers* debugging programs (Section 6.1), and the need for understanding the impact of variability on bug finding. Second, I describe previous studies of *programs* (Section 6.2), software bugs in particular, but I also motivate the study of the nature of variability bugs occurring in highly-configurable software systems. Third, and finally, I present related work on tooling support to deal with variability on the code analysis level (Section 6.3), specifically, to find bugs that occur in only certain program variants.

6.1 Empirical Studies of Programmers Debugging Programs

This section discusses work on *bug finding* and *program comprehension* related to **Paper 1A**, and then relates eye-tracking studies on debugging to **Paper 1B**.

6.1.1 Bug Finding & Program Comprehension

In the late 1980es, Oman *et al.* [72] compared debugging abilities of *novice*, *intermediate*, versus *skilled* student programmers using two Pascal programs. Not surprisingly, they found, among other things, that the ability of programmers to find errors increases with general programming experience. Experienced programmers found errors faster than less experienced programmers. In fact, they concluded that experienced programmers become faster and make fewer mistakes. Comparing to **Paper 1A**, I did not notice any difference in terms of bug-finding time between Ph.D. and M.Sc. students. But, the former appeared to be more careful and meticulous when debugging than the latter. This can be explained by the educational level of subjects in that study. They considered only undergraduate students, separating them according to the amount of computer science courses taken, whereas I ran the experiment with graduate students, who would likely be considered as skilled programmers in their setup. Furthermore, I studied a different phenomenon, which is variability, that might be challenging independently of educational level. However, I stress that I did not design my experiment to directly compare novices versus experts even though I discuss some indications.

Feigenspan *et al.* [38] in a series of controlled experiments showed that use of distinct background colors (in place of `#ifdefs`) improves comprehension of programs with preprocessor directives, independently of size and programming language of the underlying product. Additionally, they found that programmers generally favor background colors. This is one important reason why I used background colors in my experiment (cf. **Paper 1A**), instead of preprocessor directives. In other words, I assigned colors to features by showing conditional statements using background colors rather than `#ifdefs`. I observed that the speed of bug finding decreases linearly with the number of features. For instance, developers spent a bit less than 10 minutes to find a bug in a program with three features, on average.

Ribeiro *et al.* [78] conducted a controlled experiment to evaluate whether emergent interfaces (EI's) reduce effort and number of errors during code-change tasks involving feature code dependencies. Emergent interfaces are an example of tooling that attempts to simplify reasoning about variability. In general, they found a decrease in code-change effort and number of errors when using their EI tool support. Based on my two controlled experiments, I can confirm the need for more research on such tools. I observed that developers perform more code navigation in the presence of variability. Knowing that, the builders of debugging and developer support tools should consider providing convenient ways to navigate from uses to definitions and back again and along call-returns for method invocations. This kind of information might be more useful in areas of code that involve variability (i.e., `#ifdefs`), as suggested in **Paper 1B**.

Schulze *et al.* [80] studied the influence of the discipline of preprocessor annotations on program comprehension. They considered preprocessor annotations to be disciplined only those which align with the syntactic structure of the programming language; otherwise, they are not. Thus, disciplined annotations encompass only code fragments that belong to entire subtrees in the corresponding abstract syntax tree. For instance, preprocessor directives wrapping an entire function definition is a disciplined annotation. In the paper, they suggest that finding bugs in the presence of variability is time-consuming and difficult. The results of my experiments are consistent with this finding and complement it with hard evidence. In fact, I was able to quantify the increase of difficulties as the number of features grows. They also found that the discipline of preprocessor annotations has no influence at all on program comprehension. However, another study by Malaquias and co-authors [62], on the very same subject, indicates the contrary, i.e., that the discipline of preprocessor annotations does matter. In my studies I did not focus on this particular issue, which would require another design and setup.

Another controlled experiment with 17 programmers applied *functional magnetic resonance imaging* (fMRI) to measure program comprehension [81]. The participants were asked to understand simple code snippets. They found that five different brain regions associated with working memory, attention, and language processing become activated for comprehending source code. However, variability was not in their focus. I, in turn, designed and ran two controlled experiments to quantify the effect of variability on debugging, as well as understand qualitatively *how* programmers approach and debug programs with preprocessor directives.

Recently, Medeiros and co-authors interviewed 40 developers to study their perceptions of the C preprocessor [64]. The developers assess that preprocessor-related bugs are easier to introduce, harder to fix, and more critical than other bugs. Many admit that they check only a few configurations of the source code in practice when testing their implementations. My experiments confirm these qualitative insights and complement them with quantitative data.

6.1.2 Eye-Tracking Studies

A few studies have used eye tracking to study debugging and program comprehension in ordinary programs (i.e., without variability).

Hansen *et al.* [43] used eye tracking to investigate factors that impact code comprehension. They exposed ten small Python programs to the participants to predict the exact output. They found that even subtle notation changes can have impact on performance, and that notation can also make a simple program more difficult to read. Relating to my work, "subtle notations" (preprocessor directives in my case) might have impact on how programmers approach and perceive programs with variability in comparison with ordinary programs. However, I cannot separate the effect of notation (`#ifdef, #endif`) from the underlying complexity of variability.

Busjahn *et al.* [18] conducted eye-tracking studies on small programs to investigate how programmers read code compared to natural language text. They found that the fixation durations increased when reading source code in comparison with natural language text. Busjahn *et al.* [17] also studied linearity (sequential reading) and whether or not the linearity effect in reading natural languages transfers to reading of source code. They asked programmers to read and comprehend snippets of natural language text versus Java programs. They observed that expert programmers read code less linearly than novices which, in turn, read code less linearly than natural language text. They also suggested that non-linear reading skills increase with expertise. While I conduct my study in a similar fashion, I have focused on *how* developers debug programs with variability. In terms of linearity, I found that variability appears to prolongs the "initial scan" of the program (first line to last line) that most developers initiate debugging with. Interestingly, I also identified that developers seem to debug programs with variability by considering either one configuration at a time (consecutively, a sort of linearity) or all configurations at the same time (simultaneously, a sort of non-linearity).

Rodeghero *et al.* [79] conducted an eye-tracking study of ten Java programmers. They asked the programmers to read Java methods and to write English summaries of those methods. They noticed that the programmers looked more at a method's signature than its body in order to summarize it in plain English. In my study setup, I designed an eye-tracking experiment to "see" how developers approach programs with variability during debugging tasks. Among other things, I observed that the presence of variability correlates with increase in the number of gaze transitions between definitions and usages for fields and methods.

None of the above eye-tracking studies investigated debugging in the presence of variability. In other words, variability was not in their focus. I, in turn, focused on the interplay between variability and debugging from the programmers' perspective. To the best of my knowledge, this is the first study of variability debugging using eye tracking. I could draw a number of observations (cf. **Paper 1B**). However, I believe that further research (either *confirmatory* or *exploratory*) using eye tracking on variability debugging is important and required to confront my findings, and to draw new ones.

6.2 Empirical Studies of Software Bugs

This section discusses work related to the study of 98 variability bugs in four highly-configurable systems (**Paper 2A**), and to the quantitative analysis of configuration-dependent warnings in Linux (**Paper 2B**); both studies are described in Chapter 4.

ClabureDB [83] is a database of classified bug-reports to accommodate various kinds of errors from diverse projects and project versions, e.g., the Linux kernel. The database is automatically populated using existing bug finders, e.g., Clang and Stanse. The objective is to support research and development in the area of bug-finding techniques and tools by providing data for their automatic evaluation. In terms of size, the VBDb database is comparatively small, since we populated it manually, as no suitable bug finders handling variability that scale to large projects exist (which also means that none of the VBDb bugs are covered in ClabureDB adequately). As of May 2017, ClabureDB contains 221 confirmed Linux bugs and 850 false positives, whereas VBDb has only 98 confirmed variability bugs and zero false positives (by design) from Apache, BusyBox, Linux and Marlin. VBDb is unlikely to contain false positives as we studied bug already found and fixed by developers. Although ClabureDB has a similar purpose to that of VBDb, it does not consider variability. Also, unlike ClabureDB, VBDb offers 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 where irrelevant details are abstracted away, along with explanations and references intended for researchers with limited kernel experience.

Palix *et al.* [74] reproduced an old empirical study (from 2001) on Linux to reevaluate and investigate the evolution of bugs in Linux over the last decade. The results are available in a public archive.[1] This study has identified a series of bugs and rule violations such as "do not use floating point in the Linux kernel". They hunted for Linux-specific kinds of bugs in the code base. Additionally, the bugs were not corroborated by Linux developers, so it is unclear how many of these bugs are, in fact, real bugs. I, along with my co-authors, focused on qualitatively understanding the complexity and nature of variability bugs based on four different open-source software systems.

Hyunsook Do *et al.* [35] provided an infrastructure to help the execution of controlled experiments related to software testing techniques. The main purpose is to support reproducible experimentation and minimize some challenges when performing a new study, such as the high costs when gathering proper artifacts for the controlled experiment. The infrastructure provides elements to execute test cases (e.g., oracles, test

[1] `http://faultlinux.lip6.fr/`

classes, stubs, etc.) and inputs to reveal faults. Likewise, 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—i.e., configurations—that trigger the bugs. In addition, the VBDb database might be used to conduct empirical studies to better understand how developers introduce variability bugs in highly-configurable systems. The work that introduces the infrastructure [35] also includes a list of research already using and benefiting from it. Similarly, VBDb has already been used in a variety of recent publications [63, 3].

Nadi *et al.* [71] mined the Linux repository to study the causes and fixes of *variability anomalies*. An *anomaly* is a *mapping* error, such as mapping code to an invalid configuration or code mapped to nonexistent features, which can be detected by checking satisfiability of Boolean formulas over features. While we conducted our study in a similar way, we have focused on a broader class of semantic errors in code, including data- and control-flow bugs. They could analyze several commits automatically, we however performed an in-depth manual analysis for every single commit.

Apel *et al.* [6] used a model checker to find feature interactions in a simple email client, using a technique known as *variability encoding* (*configuration lifting* [77]). In variability encoding, features are encoded as Boolean variables and conditional compilation directives (e.g., `#if`) are transformed into ordinary conditional statements (e.g., `if`). We, in turn, focused on understanding the nature and complexity of variability bugs widely. This cannot be done with a model checker looking for a specific class of feature interactions. In fact, searching for bugs with tools only finds cases that these tools cover. Thus, an automated bug hunt would be heavily biased against a few kinds of bugs for which the tools were designed. Understanding variability bugs should lead to building scalable bug finders, enabling studies (e.g., [6]) to be run on Linux in the future.

Medeiros *et al.* [65] investigated *syntactic* variability errors in preprocessor-based systems. They used a variability-aware C parser [54] to automate their bug finding and exhaustively find *all* syntax errors. They noticed that developers introduce syntax errors when changing existing code and adding preprocessor directives, and that some of the relevant errors survive in the life cycle all the way to the release stage. But, they found only few tens of errors in 41 program families, suggesting that syntactic variability errors are not common in committed code. In contrast to our work, the difference is that we are not concerned with syntactic errors, but rather with the wider category of more complex *semantic* errors.

Medeiros *et al.* [66] have also conducted an empirical study on configuration-related issues, specifically, investigating undeclared and unused identifiers. They found 39 configuration-related issues, including 14 (36%) undeclared functions, 2 (5%) undeclared variables, 7 (18%) unused functions, and 23 (41%) unused variables. In other words, they noticed that 59% of the issues are related to unused functions and variables. We complement this finding by generating randomly thousands of configurations from the Linux kernel, in which we observed that the most abundant warning is unused function and the third most is unused variable. Besides that, we ranked both warnings and subsystems, e.g., the `drivers/` subsystem and `include/` header files produced warnings in around half of all configurations we compiled; whereas core subsystems such as `kernel/` and `security/` rarely produced warnings. In addition, we focused on configuration-related bugs and warnings widely, i.e., not targeting a particular type of bug.

Tian *et al.* [90] studied the problem of distinguishing bug fixing commits in the Linux repository. They used semi-supervised learning to classify commits according to tokens in the commit log and code metrics extracted from the patch contents. They remarkably improved recall over keyword-based methods, without lowering precision. In contrast, we used the keyword-based method for two reasons. First, our main emphasis was on *analyzing* commits; not in finding commits to be analyzed, which was secondary and not difficult for our study. Second, it can easily be applied in any project that stores historical information on changes. Thus, we found a simple keyword-based method sufficient for our purpose.

Yin *et al.* [96] studied more than 500 configuration errors (or *misconfigurations*) in open source and commercial software. They target systems in which parameters are read from configuration files, as opposed to systems configured statically. More importantly, they document errors from the *user* perspective, whereas we provide a documentation from the *programmer* standpoint.

Padioleau *et al.* [73] studied collateral evolution of the Linux kernel, especially device drivers, following a research method close to ours. Collateral evolution happens when existing code is adapted to changes in the kernel interfaces. They automatically identified potential collateral evolution candidates by analyzing bug fixes using heuristics, and then manually selected 72 for a more meticulous analysis. Similarly, they classified and performed an in-depth analysis of their data. Yet, the main difference to our work is that we performed an *exploratory* study on variability bugs widely (not in a particular subsystem), once we did not have any prior knowledge about the nature of these bugs.

6.3 Techniques for Finding Variability Bugs

This section examines related work to the proposed rewriting variability technique (**Paper 3A**) for lifting conventional single-program analysis tools to find variability bugs, as described in Chapter 5.

Recently, formal analysis and verification of program families have been a topic of considerable research. The challenge is to develop efficient techniques that work at the level of program families, rather than the level of single programs. There are two main approaches to *lifting* analysis to program family level [88]: (1) to develop dedicated variability-aware (family-based) techniques and tools; (2) to use specific simulators and encodings which transform program families into single programs that can be analyzed by the standard single-program verification tools. The two approaches have different strengths and weaknesses. The advantage of (1) is that precise (conclusive) results are reported for every variant, but the disadvantage is that their design and implementation can be tedious and labor intensive. On the other hand, the approaches based on (2) re-use existing tools from the single-program world, but some precision may be lost when interpreting the obtained results.

Various variability-aware techniques have been proposed which *lift* existing single-program analysis techniques to work on the level of program families. This includes variability-aware syntax checking [52, 39], type checking [51, 20], static analysis [14, 13, 70], model checking [22], and so forth. For instance, TypeChef [52] and SuperC [39] are able to parse C code with preprocessor directives. The results are ASTs with variability (choice) nodes [37]. Representationally, the difference between these two approaches is that feature expressions are represented as formulae in TypeChef, and as BDD's in SuperC. TypeChef also has implemented some variability-aware dataflow analyses. Several approaches have been proposed for type checking program families directly. For example, Kästner *et al.* [51] presented a variability-aware type checking for Featherweight Java, and Chen *et al.* [20] studied variational lambda calculus. Additionally, Classen *et al.* [22] introduced lifted model checking for verifying variability intensive systems. SNIP, a specifically designed family-based model checker, is implemented for efficient verification of temporal properties of such systems. The input language to SNIP is fPromela, which represents a variability-aware extension of the Promela language for the (single-system) SPIN model checker [45]. fPromela uses an `#ifdef`-like statement for encoding multiple variants, which represents a nondeterministic "`if`" statement guarded by features expressions that are used to specify what system parts are included (resp., excluded) for which variants. Brabrand *et al.* [14] and Midtgaard *et al.* [70] have showed how to lift any single-program dataflow analysis from the monotone framework to work on the level of program families. The obtained lifted dataflow analyses are much faster than ones based on the naive (brute-force) approach that generates and analyzes all program variants individually. Another efficient implementation of lifted analysis formulated within the IFDS framework for inter-procedural distributive environments has been proposed in SPL$^{\text{LIFT}}$ [13]. Dimovski *et al.* [32] have proposed variability abstractions to speed up the lifted verification techniques. These abstractions tame the exponential blowup caused by the large number of features and variants in a program family. This way, variability abstractions enable deliberate trading of precision for speed in the context of lifted (monotone) data-flow analysis [32, 33] and lifted model checking [31].

Another technique to analyze program families efficiently is called *variability encoding* [92] (or *configuration lifting* [77]), which is based on generating a so-called *variant simulator* which simulates the behavior of all variants in a program family. Then, an existing off-the-shelf single-program analyzer is used to verify the generated simulator, which represents a single program. Rhein *et al.* [92] have defined variability encoding on top of TypeChef for C and Java program families. They have applied the results of variability encoding to testing [53], model checking [7], and deductive verification [89]. Compared to [92], our work has the following distinguished characteristics. First, our C Reconfigurator tool is aimed to transform C program families. Second, we showed variability-related transformation rules and their correctness with respect to a minimal C-like imperative (state-based) language, whereas in [92] the rules and their correctness are demonstrated to Featherweight Java. C is a language much wider used in industry for variability than (Featherweight) Java. Third, we did not have to rely on object-oriented encodings to make the variability-transformations work. Furthermore, we provided an evaluation of our technique with several state-of-the-art single-program analysis tools for finding variability bugs on realistic C programs (both on large and sanitized files). In the end, we provided evidence that all studied single-program analysis tools—that do not normally handle variability—could detect *successfully* and *efficiently* most of the variability bugs in the transformed (reconfigured) code, after rewriting the variability of the source code. Thus, I believe that rewriting variability is a cost-efficient strategy for lifting analysis tools to program family level.

Chapter 7

Discussion: Variability Skeleton for Understanding the Impact of Changes

This is a short chapter in which I will only sketch one possible idea to the open research question **Q4**, which asks for solutions to minimize the difficulty that software developers have when reasoning about (all) configurations in a program with variability (cf. Table 7.1). I stress that this chapter does not belong to the main contributions of this Ph.D. thesis, but it solely describes a preliminary idea in the form of a technique to support programmers when maintaining code in the presence of variability.

	Problem Space	Solution Space
Programmer Perspective	– Quadrant 1 – P1: No hard evidence on how variability affects programmers. Q1: How does variability affect programmers on bug finding? T1: Variability increases the bug-finding time and makes it difficult to identify the erroneous configurations.	– Quadrant 4 – P4: Programmers fail to reason about configurations. Q4: How to support programmers to reason about configurations? T4: Beyond the scope of this dissertation. (Chapter 7 provides a sketch of one possible solution.)
Program Perspective	– Quadrant 2 – P2: Lack of evidence on how variability affects bugs. Q2: How does variability affect bugs? T2: Variability increases the complexity of bugs; and these (variability) bugs are not confined to any type of bug, feature, or location.	– Quadrant 3 – P3: Infeasible to lift all program analyses to deal with variability. Q3: How to lift conventional analysis tools to find variability bugs? T3: Rewriting variability enables single-program analysis tools to find bugs in highly-configurable systems.

Table 7.1: Research problem and question of the Quadrant 4.

In the problem statement (Chapter 2), I describe the problem **P4**, which focuses on the intersection between *solution space* and *programmer perspective*. Debugging highly-configurable systems, which include both large industrial product lines and open-source systems with thousands of features, requires understanding the combinations of features along with their data and control flows. In fact, I provide evidence that many developers fail to identify exactly the set of erroneous configurations, already for only three features (cf. **Paper 1A** and **Paper 1B**).

7.1 Motivating Scenario

Configurable software systems are challenging for developers because code fragments may be conditionally *included* or *excluded* depending on whether particular features are *enabled* or *disabled*. This means that developers need to reason about several different configurations (versions of the program), each with different data- and control-flow in order to understand a program with variability. In fact, I have observed that developers navigate much more between definitions and uses of program objects when interleaved with variability, and that variability appears to be "contagious" along the flow of control (cf. **Paper 1B**).

Figure 7.1 presents a code scenario extracted from a commercial highly-configurable race game, BestLap, with about 15 KLOC. For clarification, I have simplified and adapted the running example. The program contains two functions (`computeLevel` and `setScore`) responsible for computing the score, involving two optional features, namely, `ARENA` and `NEG_SCORE`. Feature `ARENA` is responsible for publishing the scores on a network server, while the variable `totalScore` stores the player's total score, which is greater than or equal to zero.

In this scenario, consider the following user requirement: the game should compute negative scores as well. To add penalties when the player crashes the car, the developer modifies the score computation right after the `totalScore` assignment. The changed code is shown in dark green in lines 18-20 (prefixed with a plus sign). This patch appears to be correct and in the appropriate location. For instance, compiling the configurable program with only `NEG_SCORE` *enabled*, the program variant runs properly. So, the developer can commit and push the change into the repository.

However, looking closer at this patch, we can see that the change is incomplete for configurations with `ARENA` *enabled*. The function `setScore` contains a condition prohibiting negative scores, thus breaking the specification of the program (the program is supposed to publish negative scores). This is a semantic (logical) error. To find this bug, the developer should consider all configurations and somehow see that the variable `totalScore` is always equal to zero in the end of `setScore` computation, when passing negative values to the function. For our program in Figure 7.1, the error occurs in only *one* configuration (out of four); whenever `ARENA` and `NEG_SCORE` are both *enabled*.

Therefore, the developer must somehow consider *all* configurations when debugging a configurable program. Moreover, for a program with variability it is not enough to simply find an error in some configuration. (Although, in this simplified scenario, it actually happens to be.) In order to *fix* a bug, a developer must thus not only pinpoint the error, but also

```
int totalScore = 0;
...
#ifdef NEG_SCORE
int penalty = totalCrashes * TIME_MULTIPLIER;
#endif

#ifdef ARENA
void setScore(int score)
{
  totalScore = (score < 0) ? 0 : score;
}
#endif

void computeLevel()
{
  totalScore = perfectCurvesCounter * PERFECT_CURVES_BONUS + ... ;

+  #ifdef NEG_SCORE
+  totalScore = totalScore - penalty;
+  #endif

  #ifdef ARENA
  setScore(totalScore);
  #endif
  ... // Publish scores on the network
}
```

Figure 7.1: *Simplified patch* in a configurable program.

be aware of the exact set of erroneous configurations (combinations of feature enablings/disablings). If the developer gets the configurations wrong, the bug may only be partially fixed, as presented in our scenario. In fact, **Paper 1A** provides empirical evidence that many developers fail to identify the exact set of affected configurations. Clearly, this is a difficult task. A professional senior Linux kernel developer from RedHat also attests to the difficulty of reviewing patches in presence of variability. During an interview with him, he mentioned that predicting the impact of changes in the presence of variability is a difficult challenge for him, e.g., what parts of the program a change affected and was affected by. For these reasons, a developer has to be highly alert and conscious of the features (#ifdefs) and configurations in the code.

In the following, I present the general idea of a solution that might help developers reason about configurations.

7.2 Variability-Aware Program Slicing

Yin *et al.* [97] conducted a comprehensive study on incorrect bug-fixes from large operating systems including Linux, OpenSolaris, FreeBSD and also a mature commercial OS. They noticed that 27% of the incorrect fixes are made by developers who have never touched the source code files associated with the fix. The study indicates that both developers and reviewers of incorrect fixes usually do not have enough knowledge about the involved code for debugging or changing it. This may mean that developers make mistakes during bug fixing because they do not know all the potential impact of the code they are changing. If the main potential impact (control- or data- dependencies) are plainly presented to developers, they could better detect the cause of a bug and, consequently, provide a correct fix to it. Program slicing can be adapted to analyze such information, making the patch as the slicing criterion. Then, its mostly a matter of extracting the most relevant program parts that most directly affect or is affected by the program point. Proximity is likely to be a key aspect in determining the most relevant parts to be extracted into a skeleton. Earlier indication that variability is difficult makes us believe that it would be advantageous to keep most of the variability structure intact in the skeleton (at least the variability that has to do with the modification point).

Program slicing [94] is an analysis that establishes which part of the program is relevant for a given criterion. For instance, it can compute which program elements could possibly participate in reaching a program point exhibiting an error. This can be used to speed up bug finding analyses. Slicing has applications in debugging [95], regression testing [12], program comprehension [29], among others.

However, we still lack family-based extensions of program slicing. There is only one work that tries to *lift* classic program slicing [48], but there is no stable implementation. In the context of variability, developers agree that variability bugs are easier to introduce, harder to fix, and more critical than other bugs [64]. Many still admit that they check only few configurations of the source code in practice when testing their implementations. (I confirmed these qualitative insights and provided quantitative evidence in **Paper 1A** and **Paper 1B**.) With

```
1  #include<stdio.h>
2
3  #ifdef A
4  void foo(int i){...}
5  #endif
6  #ifdef B
7  void bar(int i, int j){...}
8  #endif
9
10 int main(void)
11 {
12   int x = 2;
13   int y = 3;
14
15   #ifdef A
16   int z = 7 + y;
17   foo(x);
18   #endif
19
20   #ifdef B
21 -   foo(y);
22 +   bar(y,z);
23   #endif
24
25   printf("%d", x);
26   return 0;
27 }
```

```
1
2
3
4
5
6  #ifdef B
7  void bar(int i, int j){...}
8  #endif
9
10 int main(void)
11 {
12
13   int y = 3;
14
15   #ifdef A
16   int z = 7 + y;
17
18   #endif
19
20   #ifdef B
21
22   bar(y,z); //criterion
23   #endif
24
25
26
27 }
```

Figure 7.2: Example of a code change (commit) in a configurable program; and of a variability skeleton based on `bar(y,z)` in line 22.

this lack of awareness, developers can introduce errors when changing a feature code, making maintenance even more expensive. To tackle this issue, in the following I present the idea of *contextual variability skeleton*.

7.2.1 General Idea

The main idea is to generate a simplified program with a variability skeleton for a code change, helping developers understand the impact of the change. I believe that a slicing technique that deals with both the C code and the preprocessor directives (`#ifdefs`) has a potential to improve reasoning about features and configurations. In this way, developers might be able to comprehend, debug, and fix configurable programs. To illustrate this idea, let us consider the code example displayed in Figure 7.2. The left-hand side illustrates a commit (patch) in a configurable program involving one change; the function call (`foo(y)`) in line 21 is removed, while `bar(y,z)` is added in line 22. Suppose that a maintainer gets this code change to review. So, instead of analyzing the entire program, she can focus on the variability skeleton as a *stepping stone* for understanding the impact of the modification on the *real* program. To run a slicing technique, one needs first to select a slicing criterion, which serves to compute the data and control dependencies accordingly. By making the patch as the slicing criterion, she pinpoints the function call (`bar(y,z)`) in line 22 as her program point of interest. The right-hand side of Figure 7.2 shows the program slice without the parts that are deemed irrelevant with respect to the variability skeleton computed. Now, looking only at the (relevant) potential impacts of the change, she may have better chances to detect that this change is broken in certain configurations. Variable `z` in declared only inside feature `A` (line 16), while it is used in feature `B` (line 22). Thus, this program causes an undeclared variable error when feature `A` is *disabled* and feature `B` is *enabled*, respectively. I therefore believe that a variability-aware program slicing might make it easier to understand the potential impacts of a code change and to further identify problematic situations like changes that introduce bugs.

7.2.2 Preliminary implementation

I have a preliminary implementation of a prototype tool, *Variability Skeleton Generator*, which takes as input a C program with `#ifdefs` along with a program point (line number), and generates a simplified version of the configurable program based on the given criterion. The early version of the prototype tool is available online.[1] I have implemented it on top of TypeChef [54], which provides an infrastructure to work with preprocessor-based variability. TypeChef parses the source code containing variability in a variability-aware fashion, i.e., without generating particular configurations. As a result, it returns an abstract syntax tree (AST) *with variability*, in which variability is reflected with choice nodes over feature expressions [37]. TypeChef also has a variability-aware type system, as well as control- and data-flow analyses.

Technically speaking, the prototype tool works as follows: First, the *Variability Skeleton Generator* calls TypeChef to parse the source code with preprocessor annotations, and to perform type checking on the resulting AST with variability. Second, it runs the reaching definitions analysis and gets the def-use and use-def chains including variability

[1] `https://github.com/jccmelo/TypeChef/tree/master/CSlicer`

context. Subsequently, this step provides the definitions and uses of every variable and function in the configurable program. We now can query what are the definitions (or uses) of a particular variable using these mappings. Third, and finally, the tool performs a classic but *lifted* program slicing (cf. [94]) based on a program point, taking into account all the variability. As a result, a simplified program is generated with a variability skeleton, which might ease in understanding the impact of particular code elements including features.

However, this is just one idea among many others to support programmers reasoning about configurations. Perhaps, the combination of Braz *et al.* [15] work (change-centric approach) with the *Variability Skeleton Generator* has the potential to help developers and maintainers who commit, review, and merge variability code every day. I acknowledge that this idea should be further implemented and evaluated to obtain hard evidence whether it does improve reasoning about configurable programs, and that it scales to a large number of variants. One way of evaluating is to run a controlled experiment combined with a survey and follow-up interviews with software developers, and measure time, accuracy, and usefulness of the prototype tool in a variety of debugging, comprehensive, and fixing tasks.

Chapter *8*

Conclusion and Future Work

In this Ph.D. thesis, I have investigated the impact of *variability* on programmers and programs through different perspectives. First, I studied about a hundred variability bugs in highly-configurable systems with the objective of understanding the complexity and nature of such bugs. Second, I designed and ran two controlled experiments with almost one hundred programmers in total to quantify the impact of variability on debugging of preprocessor-based programs. The study of variability debugging using eye tracking is the first study of its kind. I have also presented a variability-aware technique that makes single-program analysis tools able to detect (variability) bugs in configurable software.

In the following, I divide my final conclusions into three parts which correspond to Theses **T1**, **T2**, and **T3** (described in Chapter 2). I discuss the *results*, *implications*, *limitations* and *future work* for each thesis.

THESIS T1

The time needed for finding bugs appears to increase with the number of features, while effectiveness of finding bugs is relatively independent of variability. However, identifying the exact set of erroneous configurations is difficult, already for a low number of features. Variability also increases the number of gaze transitions (eye movements) between definition-usages for variables and call-returns for methods.

I investigated the impact of variability on the time and accuracy of bug finding in highly-configurable systems via two controlled experiments (cf. Chapter 3). The objective of the first experiment was to *quantify* the impact of variability on debugging. I ran the experiment with N=69 programmers while measuring speed and precision for bug finding tasks defined at three different degrees of variability on several subject programs derived from real configurable systems. In the second study, I focused on *qualitatively* understanding *how* actually developers approach and debug programs in presence of variability. I exposed developers to debug programs *with* and *without* variability, while recording their eye movements using an eye tracker.

In **Paper 1A**, I learned that the time needed for finding bugs appear to increase (only) linearly with the number of features, with the time becoming less predictable with more features. I also observed that most developers correctly identify bugs, yet many fail to identify the exact set of affected configurations. This is consistent with earlier hypotheses that programmers introduce errors because it is difficult to reason about all the executions involved via configuration choices. Interestingly, this ability does not seem to improve with increasing level of education, while general, non-variability related, bug-finding skills do seem to improve. I could also see that challenges with reasoning about configurations are already measurable for low degrees of variability.

On top of quantitative results, **Paper 1B** showed that variability increases debugging time for code fragments that contain variability and for neighboring locations. Also, it appears that developers navigate much more between definitions and uses of program objects when interleaved with variability. This is presumably caused by increased complexity of def-use relationships, or by difficulties of maintaining all variants in short-term memory. Additionally, I noticed that variability seems to prolong the "initial scan" of the program that most subjects initiate debugging with and that developers appear to debug programs with variability by using either a consecutive or simultaneous approach.

Even though variability complicates debugging, it is not terribly so (only linearly). If developers reasoned about each of the variants separately we would have observed an exponential, not linear, growth. The practical implication is that it is beneficial to introduce variation points into programs from the debugging perspective. I mean, it is beneficial to pay a linear price for bug finding, if the alternative is to maintain a super-linear set of variants. However, there might be benefits in selecting designs (architectures and algorithms) that require less variability, if possible.

Another point, which seems obvious, is that reasoning about multiple configurations is a challenge. This is consistent with earlier qualitative indications that variability bugs appear when developers unintentionally ignore an execution that is enabled by an unexpected (for them) configuration of features. In fact, my study suggests that, for higher degrees of variability, it is more difficult to correctly identify the set of erroneous configurations than to find the bug in the first place. This is rather unexpected, given the fact that to understand the bug one needs to reason about control flow, a temporal non-local phenomenon that is not obviously simpler than combinatorics. This means that it is beneficial to work on support tools that help developers to navigate the configuration space (on top of flow-oriented bug finders).

Also, knowing that developers perform more navigation in code with variability, I encourage the developers to structure the code in a way that minimizes the distance between uses and definitions of field variable declarations or between methods calling each other, especially for those declarations and uses that involve variability. For the builders of programmer supporting tools, I suggest that they shall consider providing convenient ways to navigate from uses to definitions and back again and along call-returns for method invocations. Emergent interfaces [78] and emergent feature interfaces [67] are examples of tooling that attempt to simplify reasoning about variability. This thesis confirms the need for more research on such tools.

Future work. The way that these two experiments have been conducted was optimized for internal validity and observations in lab conditions. (There is an inherent trade-off between *internal* versus *external* validity in experiment design [82].) It would be interesting to investigate the extent to which the results generalize to more realistic (non-lab) conditions; more programs, larger programs, more realistic programs, higher degrees of variability, in order to confront the findings presented here and to draw new ones. However, each of these generalizations extend the duration of tasks beyond one hour. This makes it significantly harder to attract anywhere near one hundred participants for the experiments. Thus, I believe that qualitative research like surveys, case studies and action researches could gather more evidence. It would be also interesting to know when the linear relationship I observed (increasing of the bug-finding time) breaks down, for some higher degrees of variability. Presumably, at some point, developers will not be able to simply cope with the exponentially many combinations of features. Another possible direction is to exploit the results of these studies—in particular that it is challenging for developers to correctly identify the set of erroneous configurations—by building programmer supporting tools to simplify reasoning about the data and control flows of all configurations.

Thesis T2

Variability is ubiquitous. There appears to be no specific nature of variability bugs that could be exploited. Variability bugs are not confined to any particular type of bug, error-prone feature, or location. Variability also increases the complexity of bugs due to unintended feature interactions, hidden features, combinations of layers (code, mapping, model) involving different languages (e.g., C, cpp, Kconfig for Linux), many function calls, etc.

I studied variability from the program perspective by analyzing about a hundred of bugs and warnings in highly-configurable systems (cf. Chapter 4). The idea behind this study was to understand the complexity and nature of variability bugs and gather concrete examples of such bugs in order to ground research in actual problems and to evaluate tool implementations of variability-sensitive analyses. Additionally, I analyzed one of the largest open source projects, the Linux kernel, with the objective to quantify basic properties of configuration-related warnings.

In **Paper 2A**, I found that variability bugs are not limited to any particular *type of bugs*, error-prone *features*, or specific *locations*. In total, the corpus of 98 variability bugs falls in 25 different types of error categories, involving 155 distinct features, and spread out in over 30 different subsystems in the four highly-configurable systems investigated. I therefore conclude that 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 kinds of variability bug during family-based analysis, they would thus fail to detect large classes of errors (the kinds not focused on). 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. 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 all of this is not so surprising, but now we could *confirm* these folkloric hypotheses with *evidence* in terms of hard evidence.

I also characterize in what ways variability affects bugs. 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.

In **Paper 2B**, I observed 13 different types of warnings appearing in the Linux kernel. The majority of these are regarding *dead code* (e.g., unused variable) and *uninitializations*. I also found that the `drivers/` and `include/` subsystems contain warnings in about half of all configurations, whereas much fewer warnings originate from the *core* subsystems `kernel/` and `security/`. Additionally, I observed that there are generally more warnings in the in-development version of Linux than in the stable version. Finally, I hope that indication of which areas are hot in warnings can be useful for further research on bug finding in Linux, showing which subsystems are good candidates as analysis subjects. Also, this study seems to confirm the, commonly held but rarely followed, recommendation to focus bug-finding studies on unstable trees of Linux, as opposed to stable. Moreover, the study shows that the errors survive all the way to the stable version, so the variability-aware tools could help to substantially decrease the bug density in Linux.

Future work. A natural direction to continue the study of 98 variability bugs in four highly-configurable systems would be to design quantitative studies to confirm the 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 can be a worth exploring direction in the future. In fact, the VBDb database has already influenced a quantitative study on the effectiveness of sampling strategies for configurable systems [63]. Al-Hajjaji et al. [3] also used VBDb to derive a set of mutation operators for software with preprocessor-based variability. I 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, I also hope that the community can contribute to the usefulness 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.

Regarding the quantitative analysis of warnings in Linux, one possible direction would be to evaluate whether the current results hold for another technique that generates random configurations instead of `randconfig` (as the probability distribution of `randconfig` is not thoroughly representative).

Thesis T3

Single-program analysis tools, which do not handle variability, are able to effectively and efficiently find bugs in configurable programs by rewriting variability. Rewriting variability appears to be a cost-efficient solution to *lifting* program analyses.

Knowing that variability is ubiquitous and that program analysis tools aiming to find variability bugs in highly-configurable systems need to be targeted widely, I along with my co-authors proposed a method to *lift* any analysis tools to configurable systems (cf. **Paper 3A**). The method is called *rewriting variability* which consists of a series of variability-related transformations for translating configurable systems into single programs by replacing compile-time variability (`#ifdefs`) with run-time variability (ordinary `ifs`). We evaluated our transformations by analyzing the obtained transformed (single) programs using off-the-shelf single-program analysis tools such as Frama-C, Clang, and LLBMC. In fact, we could detect successfully and efficiently some interesting variability bugs using those tools.

Another way of analyzing configurable software is to use variability-aware analysis tools. However, in comparison with conventional analysis tools, variability-aware analyzers are rare, experimental, and not fast enough to extensively scan the long history of real software systems like Linux. For these reasons, we opted for transforming the source code by rewriting variability. One limitation of this approach is the precision loss, which depends on the analysis tools, since our transformation produces a super single variant that contains all possible paths that may occur in any variant.

Future work. One immediate extension is to consider more files, larger files, and realistic files to gather further evidence on the practicality of rewriting variability and confront our findings. In terms of implementation, the next step is to run our tool on a wide variety of highly-configurable systems (e.g., Linux and BusyBox), which would make it more accessible to the research community. Another direction would be to test the technique against variability-aware tools and assess the trade-offs.

Summary

In this Ph.D. project I have studied software variability from the *problem* and *solution* space in combination with the *program* and *programmer* perspective. Overall, this Ph.D. thesis shows that variability complicates debugging and the complexity of bugs, but not terribly so. This is positive and consistent with the existence of highly-configurable software systems with hundreds, even thousands, of features, testifying that developers in the trenches are able to deal with variability. This thesis also shows that, by rewriting the variability in the source code, off-the-shelf single-program analysis tools are able to detect successfully and efficiently variability bugs.

Bibliography

[1] Typechef analysis engine. `http://ckaestne.github.com/TypeChef/`.

[2] I. Abal, C. Brabrand, and A. Wasowski. 42 Variability Bugs in the Linux Kernel: A Qualitative Analysis. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ASE '14, pages 421–432, New York, NY, USA, 2014. ACM.

[3] M. Al-Hajjaji, F. Benduhn, T. Thüm, T. Leich, and G. Saake. Mutation operators for preprocessor-based variability. In *Proceedings of the 10th International Workshop on Variability Modelling of Software-intensive Systems*, VaMoS '16, pages 81–88, New York, NY, USA, 2016. ACM.

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

[5] S. Apel, C. Kästner, A. Grö*ß*linger, and C. Lengauer. Type safety for feature-oriented product lines. *Automated Software Engineering*, 17(3):251–300, Sept. 2010.

[6] S. Apel, H. Speidel, P. Wendler, A. von Rhein, and D. Beyer. Detection of feature interactions using feature-aware verification. In *Proceedings of the 26th IEEE/ACM International Conference on Automated Software Engineering*, ASE '11, pages 372–375, Washington, DC, USA, 2011. IEEE Computer Society.

[7] S. Apel, A. von Rhein, P. Wendler, A. Größlinger, and D. Beyer. Strategies for product-line verification: case studies and experiments. In *35th Intern. Conference on Software Engineering, ICSE '13*, pages 482–491, 2013.

[8] D. Benavides, S. Segura, and A. R. Cortés. Automated analysis of feature models 20 years later: A literature review. *Information Systems*, 35(6):615–636, 2010.

[9] T. Berger, D. Nair, R. Rublack, J. M. Atlee, K. Czarnecki, and A. Wąsowski. Three cases of feature-based variability modeling in industry. In *International Conference on Model Driven Engineering Languages and Systems*, pages 302–319. Springer, 2014.

[10] T. Berger, R. Rublack, D. Nair, J. M. Atlee, M. Becker, K. Czarnecki, and A. Wąsowski. A survey of variability modeling in industrial practice. In *Proceedings of the Seventh International Workshop on Variability Modelling of Software-intensive Systems*, page 7. ACM, 2013.

[11] T. Berger, S. She, R. Lotufo, A. Wasowski, and K. Czarnecki. A study of variability models and languages in the systems software domain. *Software Engineering, IEEE Transactions on*, 39(12):1611–1640, Dec 2013.

[12] D. Binkley. The application of program slicing to regression testing. *Information and Software Technology*, 40(11–12):583 – 594, 1998.

[13] E. Bodden, T. Tolêdo, M. Ribeiro, C. Brabrand, P. Borba, and M. Mezini. Spllift: Statically analyzing software product lines in minutes instead of years. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI ’13, pages 355–364, New York, NY, USA, 2013. ACM.

[14] C. Brabrand, M. Ribeiro, T. Tolêdo, J. Winther, and P. Borba. Intraprocedural dataflow analysis for software product lines. *Transactions on Aspect-Oriented Software Development X*, pages 73–108, 2013.

[15] L. Braz, R. Gheyi, M. Mongiovi, M. Ribeiro, F. Medeiros, and L. Teixeira. A change-centric approach to compile configurable systems with #ifdefs. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*, GPCE 2016, pages 109–119, New York, NY, USA, 2016. ACM.

[16] K. Burr and W. Young. Combinatorial test techniques: Table-based automation, test generation and code coverage. In *Proceedings of the International Conference on Software Testing Analysis & Review*. San Diego, 1998.

[17] T. Busjahn, R. Bednarik, A. Begel, M. Crosby, J. H. Paterson, C. Schulte, B. Sharif, and S. Tamm. Eye movements in code reading: Relaxing the linear order. In *2015 IEEE 23rd International Conference on Program Comprehension*, pages 255–265, May 2015.

[18] T. Busjahn, C. Schulte, and A. Busjahn. Analysis of code reading to gain more insight in program comprehension. In *Proceedings of the 11th Koli Calling International Conference on Computing Education Research*, Koli Calling '11, pages 1–9, New York, NY, USA, 2011. ACM.

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

[20] S. Chen, M. Erwig, and E. Walkingshaw. An error-tolerant type system for variational lambda calculus. In *ACM SIGPLAN International Conference on Functional Programming, ICFP'12*, pages 29–40, 2012.

[21] Clang. Clang static analyzer. Clang: a C language family frontend for LLVM.

[22] A. Classen, M. Cordy, P. Schobbens, P. Heymans, A. Legay, and J. Raskin. Featured transition systems: Foundations for verifying variability-intensive systems and their application to LTL model checking. *IEEE Transactions on Software Engineering*, 39(8):1069–1089, 2013.

[23] A. Classen, P. Heymans, P. Y. Schobbens, and A. Legay. Symbolic model checking of software product lines. In *Proceedings of the 33rd International Conference on Software Engineering (ICSE)*, pages 321–330, May 2011.

[24] A. Classen, P. Heymans, P.-Y. Schobbens, A. Legay, and J.-F. Raskin. Model checking lots of systems: efficient verification of temporal properties in software product lines. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering (ICSE) - Volume 1*, pages 335–344. ACM, 2010.

[25] P. Clements and L. Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley, 2002.

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

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

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

[29] Y. Deng, S. Kothari, and Y. Namara. Program slice browser. In *Proceedings of the 9th International Workshop on Program Comprehension*, IWPC '01, pages 50–59, Washington, DC, USA, 2001. IEEE Computer Society.

[30] A. S. Dimovski. Symbolic game semantics for model checking program families. In *Model Checking Software - 23nd International Symposium, SPIN 2016, Proceedings*, volume 9641 of *LNCS*, pages 19–37. Springer, 2016.

[31] A. S. Dimovski, A. S. Al-Sibahi, C. Brabrand, and A. Wąsowski. Efficient family-based model checking via variability abstractions. *International Journal on Software Tools for Technology Transfer*, pages 1–19, 2016.

[32] A. S. Dimovski, C. Brabrand, and A. Wasowski. Variability abstractions: Trading precision for speed in family-based analyses. In *29th European Conference on Object-Oriented Programming, ECOOP '15*, volume 37 of *LIPIcs*, pages 247–270. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2015.

[33] A. S. Dimovski, C. Brabrand, and A. Wasowski. Finding suitable variability abstractions for family-based analysis. In *FM 2016: Formal Methods - 21st International Symposium, Proceedings*, volume 9995 of *LNCS*, pages 217–234, 2016.

[34] A. S. Dimovski and A. Wąsowski. Variability-specific abstraction refinement for family-based model checking. In *International Conference on Fundamental Approaches to Software Engineering*, pages 406–423. Springer, Berlin, Heidelberg, 2017.

[35] H. Do, S. Elbaum, and G. Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engineering: An International Journal*, 10:405–435, 2005.

[36] M. D. Ernst, G. J. Badros, and D. Notkin. An empirical analysis of C preprocessor use. *IEEE Transactions on Software Engineering*, 28:1146–1170, 2002.

[37] M. Erwig and E. Walkingshaw. The choice calculus: A representation for software variation. *ACM Transactions on Software Engineering and Methodology*, 21(1):6:1–6:27, Dec. 2011.

[38] J. Feigenspan, C. Kästner, S. Apel, J. Liebig, M. Schulze, R. Dachselt, M. Papendieck, T. Leich, and G. Saake. Do background colors improve program comprehension in the #ifdef hell? *Empirical Software Engineering*, 18(4):699–745, Aug. 2013.

[39] P. Gazzillo and R. Grimm. SuperC: Parsing all of C by taming the preprocessor. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, pages 323–334, New York, NY, USA, 2012. ACM.

[40] M. Goedicke, K. Pohl, and U. Zdun. Domain-specific runtime variability in product line architectures. *Proceedings of Object-Oriented Information Services (OOIS'02).: Lecture Notes in Computer Science*, pages 213–218, 2002.

[41] C. L. Goues, S. Forrest, and W. Weimer. Current challenges in automatic software repair. *Software Quality Journal*, 21(3):421–443, 2013.

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

[43] M. Hansen, R. L. Goldstone, and A. Lumsdaine. What makes code hard to understand? *arXiv preprint arXiv:1304.5257*, 2013.

[44] W. E. Hick. On the rate of gain of information. *Quarterly Journal of Experimental Psychology*, 4(1):11–26, 1952.

[45] G. J. Holzmann. *The SPIN Model Checker - primer and reference manual*. Addison-Wesley, 2004.

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

[47] K.-C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson. Feature-Oriented Domain Analysis (FODA). Feasibility Study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, 1990.

[48] F. Kanning and S. Schulze. Program slicing in the presence of preprocessor variability. In *Proceedings of the 2014 IEEE International Conference on Software Maintenance and Evolution*, ICSME '14, pages 501–505, Washington, DC, USA, 2014. IEEE Computer Society.

[49] C. Kastner and S. Apel. Type-checking software product lines - a formal approach. In *Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering*, ASE '08, pages 258–267, Washington, DC, USA, 2008. IEEE Computer Society.

[50] C. Kästner, S. Apel, and M. Kuhlemann. Granularity in Software Product Lines. In *Proceedings of the 30th International Conference on Software Engineering (ICSE)*, pages 311–320. ACM, 2008.

[51] C. Kästner, S. Apel, T. Thüm, and G. Saake. Type checking annotation-based product lines. *ACM Transactions on Software Engineering and Methodology*, 21(3):14, 2012.

[52] C. Kästner, P. G. Giarrusso, T. Rendel, S. Erdweg, K. Ostermann, and T. Berger. Variability-aware parsing in the presence of lexical macros and conditional compilation. In *Proc. 2011 ACM int. conf. on Object oriented programming systems languages and applications*, OOPSLA, pages 805–824, 2011.

[53] C. Kästner, A. von Rhein, S. Erdweg, J. Pusch, S. Apel, T. Rendel, and K. Ostermann. Toward variability-aware testing. In *FOSD '12*, pages 1–8, 2012.

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

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

[56] F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski. Frama-C: A software analysis perspective. *Formal Aspects of Computing*, 27(3):573–609, 2015.

[57] M. Krone and G. Snelting. On the inference of configuration structures from source code. In *Proceedings of the 16th International Conference on Software Engineering (ICSE)*, pages 49–57. IEEE Computer Society Press, 1994.

[58] K. Lauenroth, K. Pohl, and S. Toehning. Model checking of domain artifacts in product line engineering. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, ASE '09, pages 269–280, Washington, DC, USA, 2009. IEEE Computer Society.

[59] D. Le, E. Walkingshaw, and M. Erwig. #ifdef confirmed harmful: Promoting understandable software variation. In *IEEE International Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 143–150, 2011.

[60] J. Liebig, S. Apel, C. Lengauer, C. Kästner, and M. Schulze. An analysis of the variability in forty preprocessor-based software product lines. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering (ICSE)*, pages 105–114. ACM, 2010.

[61] B. P. Lientz and E. B. Swanson. *Software Maintenance Management*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1980.

[62] R. Malaquias, M. Ribeiro, R. Bonifácio, E. Monteiro, F. Medeiros, A. Garcia, and R. Gheyi. The discipline of preprocessor-based annotations does #ifdef tag n't #endif matter. In *Proceedings of the*

25th International Conference on Program Comprehension, ICPC '17, pages 297–307, Piscataway, NJ, USA, 2017. IEEE Press.

[63] F. Medeiros, C. Kästner, M. Ribeiro, R. Gheyi, and S. Apel. A comparison of 10 sampling algorithms for configurable systems. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 643–654, New York, NY, USA, 2016. ACM.

[64] F. Medeiros, C. Kästner, M. Ribeiro, S. Nadi, and R. Gheyi. The love/hate relationship with the C preprocessor: An interview study. In *29th European Conference on Object-Oriented Programming, ECOOP 2015, July 5-10, 2015, Prague, Czech Republic*, pages 495–518, 2015.

[65] F. Medeiros, M. Ribeiro, and R. Gheyi. Investigating preprocessor-based syntax errors. In *Proceedings of the 12th International Conference on Generative Programming: Concepts & Experiences*, GPCE, pages 75–84. ACM, 2013.

[66] F. Medeiros, I. Rodrigues, M. Ribeiro, L. Teixeira, and R. Gheyi. An empirical study on configuration-related issues: Investigating undeclared and unused identifiers. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*, GPCE 2015, pages 35–44, New York, NY, USA, 2015. ACM.

[67] J. Melo and P. Borba. Improving modular reasoning on preprocessor-based systems. In *Proceedings of the 7th Brazilian Symposium on Software Components, Architectures and Reuse*, SBCARS '13, pages 11–19, Washington, DC, USA, 2013. IEEE Computer Society.

[68] M. Mendonca, A. Wąsowski, and K. Czarnecki. SAT-based analysis of feature models is easy. In *Proceedings of the 13th International Software Product Line Conference*, SPLC '09, pages 231–240, Pittsburgh, PA, USA, 2009. Carnegie Mellon University.

[69] F. Merz, S. Falke, and C. Sinz. LLBMC: bounded model checking of C and C++ programs using a compiler IR. In *Verified Software: Theories, Tools, Experiments - 4th International Conference, VSTTE 2012, Proceedings*, volume 7152 of *LNCS*, pages 146–161. Springer, 2012.

[70] J. Midtgaard, A. S. Dimovski, C. Brabrand, and A. Wąsowski. Systematic derivation of correct variability-aware program analyses. *Sci. Comput. Program.*, 105(C):145–170, July 2015.

[71] S. Nadi, C. Dietrich, R. Tartler, R. C. Holt, and D. Lohmann. Linux variability anomalies: What causes them and how do they get fixed? In *Proceedings of the 10th Working Conference on Mining Software Repositories*, MSR '13, pages 111–120, Piscataway, NJ, USA, 2013. IEEE Press.

[72] P. W. Oman, R. Cook, and M. Nanja. Effects of programming experience in debugging semantic errors. *Journal of Systems and Software*, 9(3):197–207, 1989.

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

[74] N. Palix, G. Thomas, S. Saha, C. Calvès, J. Lawall, and G. Muller. Faults in linux: Ten years later. *SIGARCH Comput. Archit. News*, 39(1):305–318, Mar. 2011.

[75] L. Passos, L. Teixeira, N. Dintzner, S. Apel, A. Wąsowski, K. Czarnecki, P. Borba, and J. Guo. Coevolution of variability models and related software artifacts. *Empirical Software Engineering*, 21(4):1744–1793, 2016.

[76] K. Pohl, G. Bockle, and F. J. van der Linden. *Software Product Line Engineering*. Springer, 2005.

[77] H. Post and C. Sinz. Configuration lifting: Verification meets software configuration. In *Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering*, ASE, pages 347–350, Washington, DC, USA, 2008. IEEE Computer Society.

[78] M. Ribeiro, P. Borba, and C. Kästner. Feature maintenance with emergent interfaces. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 989–1000, New York, NY, USA, 2014. ACM.

[79] P. Rodeghero, C. McMillan, P. W. McBurney, N. Bosch, and S. D'Mello. Improving automated source code summarization via an eye-tracking study of programmers. In *Proceedings of the 36th International Conference on Software Engineering*, pages 390–401. ACM, 2014.

[80] S. Schulze, J. Liebig, J. Siegmund, and S. Apel. Does the discipline of preprocessor annotations matter?: A controlled experiment. In *Proceedings of the 12th International Conference on Generative Programming: Concepts & Experiences*, GPCE '13, pages 65–74, 2013.

[81] J. Siegmund, C. Kästner, S. Apel, C. Parnin, A. Bethmann, T. Leich, G. Saake, and A. Brechmann. Understanding understanding source code with functional magnetic resonance imaging. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 378–389, New York, NY, USA, 2014. ACM.

[82] J. Siegmund, N. Siegmund, and S. Apel. Views on internal and external validity in empirical software engineering. In *Software Engineering (ICSE), 2015 IEEE/ACM 37th IEEE International Conference on*, volume 1, pages 9–19, May 2015.

[83] J. Slaby, J. Strejček, and M. Trtík. ClabureDB: Classified Bug-Reports Database. In R. Giacobazzi, J. Berdine, and I. Mastroeni, editors, *Verification, Model Checking, and Abstract Interpretation*, volume 7737 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2013.

[84] H. Spencer and G. Collyer. #ifdef considered harmful, or portability experience with C news. In *Proceedings of the Usenix Summer Technical Conference*, pages 185–198. Usenix Association, 1992.

[85] R. Tartler. Finding and burying Configuration Defects in Linux with the undertaker. 2011.

[86] R. Tartler, C. Dietrich, J. Sincero, W. Schröder-Preikschat, and D. Lohmann. Static analysis of variability in system software: The 90, 000 #ifdefs issue. In G. Gibson and N. Zeldovich, editors, *2014 USENIX Annual Technical Conference, USENIX ATC '14, Philadelphia, PA, USA, June 19-20, 2014.*, pages 421–432. USENIX Association, 2014.

[87] R. Tartler, D. Lohmann, C. Dietrich, C. Egger, and J. Sincero. Configuration coverage in the analysis of large-scale system software. *Operating Systems Review*, 45(3):10–14, 2011.

[88] T. Thüm, S. Apel, C. Kästner, I. Schaefer, and G. Saake. A classification and survey of analysis strategies for software product lines. *ACM Computing Surveys (CSUR)*, 47(1):6:1–6:45, June 2014.

[89] T. Thüm, I. Schaefer, M. Hentschel, and S. Apel. Family-based deductive verification of software product lines. In *Generative Programming and Component Engineering, GPCE'12*, pages 11–20. ACM, 2012.

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

[91] M. Völter. Handling variability. In *Proceedings of the 14th annual European Conference on Pattern Languages of Programming*, EuroPLoP'09. Kelly & Weiss, 2009.

[92] A. von Rhein, T. Thüm, I. Schaefer, J. Liebig, and S. Apel. Variability encoding: From compile-time to load-time variability. *Journal of Logical and Algebraic Methods in Programming*, 85(1):125–145, 2016.

[93] E. Walkingshaw and K. Ostermann. Projectional editing of variational software. In *Proceedings of the 2014 International Conference on Generative Programming: Concepts and Experiences*, GPCE 2014, pages 29–38, New York, NY, USA, 2014. ACM.

[94] M. Weiser. Program slicing. In *Proceedings of the 5th International Conference on Software Engineering (ICSE)*, pages 439–449. IEEE Press, 1981.

[95] M. Weiser. Programmers use slices when debugging. *Commun. ACM*, 25(7):446–452, July 1982.

[96] Z. Yin, X. Ma, J. Zheng, Y. Zhou, L. N. Bairavasundaram, and S. Pasupathy. An empirical study on configuration errors in commercial and open source systems. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles*, SOSP '11, New York, NY, USA, 2011. ACM.

[97] Z. Yin, D. Yuan, Y. Zhou, S. Pasupathy, and L. Bairavasundaram. How do fixes become bugs? In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering (ESEC/FSE)*, pages 26–36. ACM, 2011.

[98] B. Zhang and M. Becker. Recovar: A solution framework towards reverse engineering variability. In *Proceedings of the 4th International*

Workshop on Product Line Approaches in Software Engineering (PLEASE), pages 45–48, 2013.

How Does the Degree of Variability Affect Bug Finding? (Paper 1A)

2016 IEEE/ACM 38th IEEE International Conference on Software Engineering

How Does the Degree of Variability Affect Bug Finding?

Jean Melo, Claus Brabrand, Andrzej Wąsowski
IT University of Copenhagen, Denmark
{jeanmelo,brabrand,wasowski}@itu.dk

ABSTRACT

Software projects embrace variability to increase adaptability and to lower cost; however, others blame variability for increasing complexity and making reasoning about programs more difficult. We carry out a controlled experiment to quantify the impact of variability on debugging of preprocessor-based programs. We measure speed and precision for bug finding tasks defined at three different degrees of variability on several subject programs derived from real systems.

The results show that the speed of bug finding decreases linearly with the degree of variability, while effectiveness of finding bugs is relatively independent of the degree of variability. Still, identifying the set of configurations in which the bug manifests itself is difficult already for a low degree of variability. Surprisingly, identifying the exact set of affected configurations appears to be harder than finding the bug in the first place. The difficulty in reasoning about several configurations is a likely reason why the variability bugs are actually introduced in configurable programs.

We hope that the detailed findings presented here will inspire the creation of programmer support tools addressing the challenges faced by developers when reasoning about configurations, contributing to more effective debugging and, ultimately, fewer bugs in highly-configurable systems.

Categories and Subject Descriptors

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

Keywords

Variability, Preprocessors, Bug Finding

1. INTRODUCTION

A recent study reports that the global cost of debugging software has risen to 312 billion dollars annually, and that on average, software developers spend half of their programming time finding and fixing bugs [14]. This is particularly worrisome in the context of variability. Software projects embrace variability hoping to increase flexibility at lower cost (to better control system resources, to extend portability across different hardware, to meet requirements of various market segments). However, multiple research indicate that variability might also amplify maintenance problems. Recent literature on variability is riddled with claims to that end. We list a few examples: *"bug-finding is a time-consuming and tedious task in the presence of variability"*[30]; *"managing variability can become complex"* [34]; *"variability specifications and realizations tend to erode in the sense that they become overly complex."* [36]; *"understandability and maintainability may be negatively affected"* [12].

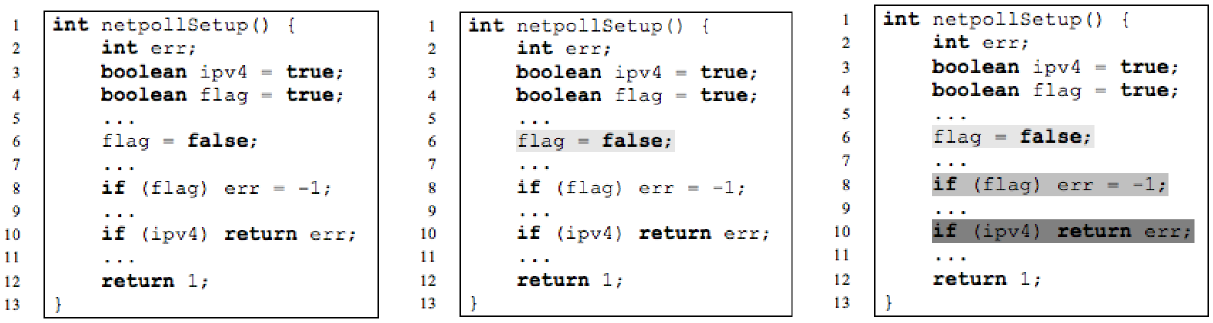
However reasonable, there is little to no hard evidence for these claims. Specifically, how are maintenance tasks (bug finding in particular) affected by variability? How much harder is it to debug a program as variability increases? Does variability affect speed or also quality of debugging? In this paper, we set off to understand such issues using a controlled experiment designed to quantify the impact of the degree of variability in program code on bug finding.

In the experiment we use simplifications of real bugs extracted from Linux, BusyBox, and BestLap. We define three degrees of variability: no, low, and high; corresponding to zero, one, and three features, respectively. Given a program and a degree of variability, we ask the participants to debug the programs. In summary, we learn that:

- The time needed for finding bugs increases only linearly with the amount of features, with the time becoming less predictable with more features. The differences in effectiveness of various programmers are amplified by the increase of variability.
- Most developers correctly identify bugs (partial correctness), yet many fail to identify the set of affected configurations (complete correctness). This is consistent with earlier hypotheses that programmers introduce errors because it is difficult to reason about all the executions involved via configuration choices. Interestingly, this ability does not seem to improve with increasing level of education, while general, non-variability related, bug-finding skills do seem to improve.
- Challenges with reasoning about configurations are already measurable for low degrees of variability.

The intended audience of this work are designers of variability management tools, of bug-finding tools, and of other supporting methods, such as variability-aware software architectures, aiming at improving efficiency and correctness in

ICSE '16, May 14-22, 2016, Austin, TX, USA
© 2016 ACM. ISBN 978-1-4503-3900-1/16/05…$15.00
DOI: http://dx.doi.org/10.1145/2884781.2884831



```
1  int netpollSetup() {
2      int err;
3      boolean ipv4 = true;
4      boolean flag = true;
5      ...
6      flag = false;
7      ...
8      if (flag) err = -1;
9      ...
10     if (ipv4) return err;
11     ...
12     return 1;
13 }
```

(a) No variability (zero features).

```
1  int netpollSetup() {
2      int err;
3      boolean ipv4 = true;
4      boolean flag = true;
5      ...
6      flag = false;
7      ...
8      if (flag) err = -1;
9      ...
10     if (ipv4) return err;
11     ...
12     return 1;
13 }
```

(b) Low variability (one feature).

```
1  int netpollSetup() {
2      int err;
3      boolean ipv4 = true;
4      boolean flag = true;
5      ...
6      flag = false;
7      ...
8      if (flag) err = -1;
9      ...
10     if (ipv4) return err;
11     ...
12     return 1;
13 }
```

(c) High(er) variability (three features).

Figure 1: A program with an *uninitiliazed variable* error with progressively increasing degrees of variability.

development of variability-intensive systems. We hope that the insights provided will influence new tools and methods that can help avoiding bugs like the one presented in the next section, or help debugging programs that contain them. We also hope to inspire software architecture researchers to (further) understand the trade-offs between increasing variability and development cost, by also including potential debugging costs.

2. MOTIVATING SCENARIO

Today, variability-intensive software systems include both large industrial product lines [8, 26, 4, 3] and open-source systems of various sizes, up to the LINUX kernel with more than 13,000 configurable features [5]. A multitude of technologies can be used to implement configurable systems: object-oriented patterns, aspects, domain-specific languages and code generation, plugin mechanisms, and so on. Among these, the C preprocessor (CPP) is one of the *oldest*, the *simplest*, and the *most popular* [9, 19, 17] mechanisms in use, especially in the systems domain. For these reasons, we use the preprocessor in our study. The results do not generalize to other mechanisms, but provide a good indication, given that the other solutions are more complex and require much more additional code to handle variability.

Figure 1 presents an example extracted from the Netpoll module of LINUX kernel, slightly adapted to JAVA syntax using coloured lines instead of preprocessor [17]. Netpoll is an API that provides a means to implement UDP clients and servers in the kernel independently of the main networking stack. These can be used in unusual situations, like failure monitoring, crash recovery, or debugging of the kernel. The original function, called **netpoll_setup** in C, is used to initialize the module. It is about 100 lines long and involves one optional feature. Historic versions of the function, contained an *error*.[1] If the feature is disabled, the function returns the value of an *uninitialized variable*, **err**, intended to hold an error value in case of unexpected situations.

We illustrate how the task of debugging becomes more complex as the numbers of features increase. Figure 1a shows a version of the bug as a conventional program without variability. It is fairly easy to establish that the function returns the value of an uninitialized variable in line 10. (In line 6, the variable **flag** is assigned **false** which means that the conditional statement in line 8 is not executed; hence, the variable **err** which was declared and uninitialized in line 2, is never assigned a value. Now, since the value of the variable **ipv4** is **true** (line 3), the conditional statement in line 10 returns the value of the *uninitialized* variable **err**.)

Figure 1b contains the same program, but now involving *one feature* shown in light gray background color. A feature, such as light gray in this example, can be configured either as *enabled* or *disabled*. Features are used in compile-time conditional directives (**#ifdefs**) to control whether to *include* or *exclude* code fragments in a program. We use the conventions of CIDE [17] which assign colors to features and show conditional statements using background colors rather than **#ifdefs**. We discuss implications of this difference in Section 5. A colored statement is to be *included* in a program if and only if the corresponding feature (color) is *enabled*. Obviously, a feature thus gives rise to *two* possible configurations: a program *with* the light gray statement, and a program *without* it. In general, n features will give rise to 2^n configurations; i.e., 2^n programs.

Now programming errors conditionally depend on configurations. Indeed, the error in Figure 1b occurs only whenever we *enable* the light gray feature. If the light gray statement in line 6 is included, the variable **err** is not initialized in line 8. If we *disable* the light gray feature, the error no longer occurs; **err** would then be initialized in line 8.

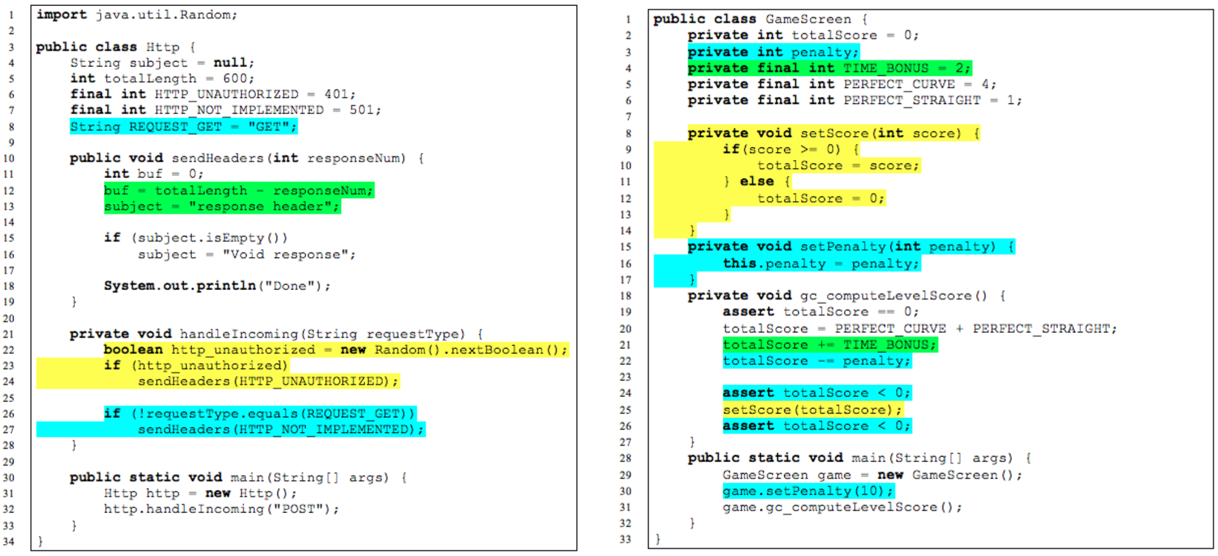
Ultimately, this leads to a notion of *partial correctness* for debugging. A developer can correctly diagnose the error (find the bug), but incorrectly misdiagnose the exact set of configurations (the combinations of features producing the bug). In the case of a single feature, partial correctness (misdiagnosis) means mixing up enabled and disabled.

The problems and difficulties escalate when we consider more features; i.e., higher degrees of variability. Figure 1c shows the same programs as before, but now with *three* features: light gray, gray, and dark gray. The three features yield eight configurations. In debugging the program, the developer must somehow consider all configurations. Determining the erroneous products becomes a non-trivial combinatorial problem which, as we shall see in Section 4, is difficult. Indeed, for our program in Figure 1c, the error now occurs in exactly three (out of eight) configurations.

3. EXPERIMENT

In this section we explain our experimental design and setup.

[1] http://git.kernel.org/cgit/linux/kernel/git/stable/linux-stable.git/commit/?id=e39363a9def53dd4086be107dc8b3ebca09f045d



```java
import java.util.Random;

public class Http {
    String subject = null;
    int totalLength = 600;
    final int HTTP_UNAUTHORIZED = 401;
    final int HTTP_NOT_IMPLEMENTED = 501;
    String REQUEST_GET = "GET";

    public void sendHeaders(int responseNum) {
        int buf = 0;
        buf = totalLength - responseNum;
        subject = "response header";

        if (subject.isEmpty())
            subject = "Void response";

        System.out.println("Done");
    }

    private void handleIncoming(String requestType) {
        boolean http_unauthorized = new Random().nextBoolean();
        if (http_unauthorized)
            sendHeaders(HTTP_UNAUTHORIZED);

        if (!requestType.equals(REQUEST_GET))
            sendHeaders(HTTP_NOT_IMPLEMENTED);
    }

    public static void main(String[] args) {
        Http http = new Http();
        http.handleIncoming("POST");
    }
}
```

(a) P2.

```java
public class GameScreen {
    private int totalScore = 0;
    private int penalty;
    private final int TIME_BONUS = 2;
    private final int PERFECT_CURVE = 4;
    private final int PERFECT_STRAIGHT = 1;

    private void setScore(int score) {
        if(score >= 0) {
            totalScore = score;
        } else {
            totalScore = 0;
        }
    }
    private void setPenalty(int penalty) {
        this.penalty = penalty;
    }
    private void gc_computeLevelScore() {
        assert totalScore == 0;
        totalScore = PERFECT_CURVE + PERFECT_STRAIGHT;
        totalScore += TIME_BONUS;
        totalScore -= penalty;

        assert totalScore < 0;
        setScore(totalScore);
        assert totalScore < 0;
    }
    public static void main(String[] args) {
        GameScreen game = new GameScreen();
        game.setPenalty(10);
        game.gc_computeLevelScore();
    }
}
```

(b) P3.

Figure 2: Programs P2 and P3 with HI variability degree; P1 is a larger version of Fig. 1c.

Prg	Origin	Filename	Bug type	LOC	#mth
P1	Linux	netpoll.c	*Uninitialized variable*	21	2
P2	BusyBox	http.c	*Null-pointer dereference*	29	3
P3	BestLap	GameScreen.java	*Assertion violation*	31	4

Prg	Degree	$\lvert\mathbb{F}\rvert$	Scattering	Tangling	VCC
P1	NO/LO/HI	0/1/3	0/1/3	0/2/6	5/6/7
P2	NO/LO/HI	0/1/3	0/3/4	0/4/8	7/8/9
P3	NO/LO/HI	0/1/3	0/5/10	0/6/15	8/12/14

Figure 3: Characteristics of our three benchmark programs: P1, P2, and P3.

3.1 Objective

Our experiment aims to analyze the impact of variability on bug finding. We want to understand exactly how much harder does the debugging task become as the degree of variability in a program increases? Specifically, we aim to answer the following research questions:

- **RQ1:** How does the degree of variability affect the ***time*** of bug finding?
- **RQ2:** How does the degree of variability affect the ***accuracy*** of bug finding?

To address these research questions, we perform a range of classic *"find the bug"* experiments [25] and measure the *time* and *accuracy* of the bug-finding task. We expose developers to programs with different degrees of variability, while controlling for noise factors such as learning, developer competence and program complexity. We measure accuracy as the number of correct vs incorrect identifications of bugs. We are particularly interested in the time and accuracy of bug finding as functions of the degree of variability.

3.2 Treatments

In order to study debugging as a function of variability, we expose each participant to programs with different *degrees of variability*, so programs using different amounts of conditional compilation blocks. We settled on using three distinct degrees of variability. Let $\mathbb{F}$ denote the set of features (conditional compilation symbols used in a program). First, to establish a baseline, we consider programs with no variability (degree NO, $\mathbb{F} = \emptyset$). Then we consider programs that use one feature (degree LO, $|\mathbb{F}| = 1$) and programs with three features (degree HI, $|\mathbb{F}| = 3$). The number of configurations grows from one for degree NO programs, two for LO programs, to eight for HI. This should make any performance differences manifest themselves clearly. Even though it would be interesting to study higher degrees, the limitation to three features has one important advantage: it leaves us with programs sufficiently small to be used in a time-delimited controlled experiment.

We derive programs of lower degrees by taking an erroneous program with three features (see below) and appropriately fix features as either *enabled* or *disabled* retaining the original error. We thus obtain three versions of each program: "NO", "LO", and "HI" (much like in Fig. 1).

3.3 Subjects

We now turn to the subjects of the experiment which are affected by the treatments; the *participants* and the *programs*.

Participants. We performed the experiment with N=69 participants: 31 M.Sc. students, 32 Ph.D. students, and 6 post-docs. The M.Sc. students came from two courses at the IT University of Copenhagen: *"Interactive Web Services using Java and XML"* and *"System Architecture and Security"*. The Ph.D. students and post-docs came from three Danish universities: IT University of Copenhagen (ITU), University of Copenhagen (KU), and Technical University of Denmark (DTU). We informed all participants that they could stop

128

participating at any time.

All participants had programming experience, especially in Java, and around half of the participants had industrial experience ranging from a few months to several years.

Programs. For the robustness of our experiment—in order to minimize risks of specific effects from particular programs and bug types—we took *three* programs with different kinds of errors. We based our programs on *real* variability errors from three highly-configurable systems: Linux [1], BusyBox[2], and BestLap [27]. These are qualitatively different systems in terms of size, architecture, purpose, variability, and complexity. Linux is an operating system and is likely the largest highly-configurable open-source system with more than 12 MLOC and 13,000 features. BusyBox is an open-source highly-configurable system with 204 KLOC and about 600 features, that provides several essential Unix tools in a single executable file. BestLap is a commercial highly-configurable race game with about 15 KLOC. The kinds of errors we consider are also different: an *uninitialized variable*, a *null-pointer dereference*, and an *assertion violation*. We simplified the error in each system down to an erroneous program that would fit on a screen without scrolling (25-35 lines) yet involve exactly three features.

Figure 2 shows the programs `P2` and `P3` with `HI` variability degree (`P1` is a larger version of Fig. 1c). The `LO` and `NO` variability programs are obtained from the `HI` variants by selecting a feature and configuration that preserves the bug and influences the program size as little as possible, so that it can still be comparable to the `HI` variant. In the following, we describe the bugs used in the experiment of `HI` variability degree, informing the type and erroneous configurations of each bug.

Bug description of `P1`. `P1` has one only method which contains conditional statements and an integer local variable called `err`, as shown in Fig. 1c. The three features yield eight possible configurations. In debugging the program, the developer must somehow consider all configurations. To accomplish the task, the developer needs to identify that the variable `err` is *uninitialized* in exactly three (out of eight) configurations. Indeed, for our program in Figure 1c, the error occurs in the following configurations by *enabling* only: (i) dark gray; (ii) light gray and dark gray; (iii) light gray, gray, and dark gray. In these erroneous configurations, the variable `err` is not initialized in line 8.

Bug description of `P2`. `P2` contains two methods to handle incoming HTTP requests (see Figure 2a). It also has a variable (`subject`) that may be null and is dereferenced for certain configurations. So, to identify the bug, the developer should realize that this happens in three configurations. In fact, the error in Figure 2a occurs only when we disable the green feature. Thus, the erroneous configurations are (when we *enable* only): (i) blue; (ii) yellow; (iii) blue and yellow. If the `sendHeaders` method is called by either the yellow or blue features, the variable `subject` will be null whenever the green feature is disabled, and in line 15 there is an access to `subject` which may cause null pointer exception.

Bug description of `P3`. `P3` has three methods responsible for computing the score, as can be seen in Fig. 2b. According to a user requirement, the game should also compute negative scores. But, the method `setScore` contains a condition prohibiting negative scores. We encode the requirement using assertions. To find the bug, the developer should consider all configurations and somehow see that the variable `totalScore` is always equal to zero in the end of `setScore` computation, when passing negative values to the method, which is revealed through an assertion violation in the code (line 26). For our program in Figure 2b, the presence condition of the error is: *blue* ∧ *yellow*. Thus, the assertion error occurs in exactly two configurations: (i) blue and yellow; (ii) blue, yellow, and green.

Figure 3 lists various characteristics for each of our programs. The left-hand side gives the basic characteristics of the programs; their origins (project and file name), bug type, number of lines of code (excluding whitespace and comments), and number of methods. The right-hand side lists variability metrics for each of the degrees: *feature scattering*, *feature tangling*, and *variational cyclometric complexity* (vcc). We show metrics for each of the degrees using a slash-separated notation (`NO`/`LO`/`HI`). For each feature, *scattering* counts the number of conditional-compilation (colored) blocks (based on the cdc metric [29]). We give accumulated numbers for all features involved. The feature in `P2-LO`, for example, is scattered over three locations in the source code. For each feature, *tangling*, in turn, counts the number of switches between regular code and feature code through the control-flow of the program (based on the cdloc metric [29]). Again, the feature in `P2-LO`, for instance, requires a concern/scope change between the code base and the feature code four times through the program. For vcc, we use the *cyclomatic complexity* metric [21] on the variability programs, treating `#ifdefs` (colored lines) as ordinary `ifs`.

Notice that the programs are all quite different, yet in terms of complexity they are hierarchically ordered from `P1` (simplest) to `P3` (most complex).

3.4 Design

We present first a fully randomized experiment design, point out a problem, and address it using a well-known technique.

In terms of debugging tasks, we have nine, in total: three programs, each at three degrees of variability. However, a developer cannot be assigned to perform *all* nine debugging tasks. Clearly, we can only have a developer find bugs in a given program *once*, otherwise there would be a *learning effect* on subsequent attempts. For similarly obvious reasons, we can only have a developer consider a variability degree *once*. Abiding by these constraints, we can have each developer debug three *different* programs, each at a *different* variability degree. However, there might still be learning effects lurking due to sequencing of assigned tasks. Presumably, debugging `LO` after `HI` is not the same as debugging `HI` after `LO`, even for different programs. Similarly, debugging an "easy" program after a "hard" one is not the same as in the reverse order.

Fully Randomized Design. To counter these effects, we need to randomize the order developers consider the tasks. Aside from learning effect, we also need to control for other subject-related noise factors (confounding factors) such as differences in *developer competence* and *program complexity*. After all, a "competent" developer debugging an "easy" program will obviously not produce the same result as an "incompetent" developer debugging a "hard" program. Again, randomization may be used to control for these effects. For larger samples these effects will diminish.

Thus, one solution would be, for each developer, to assign

[2] http://git.busybox.net/busybox/commit/?id=5cd6461b6fb51e8cf297a49074fce825e1960774

programs

developers	P1	P2	P3
D1	NO	LO	HI
D2	LO	HI	NO
D3	HI	NO	LO

Figure 4: Latin Square (3×3).

programs and degrees completely at random without ever reusing a program or a degree. This does control for confounding factors. However, statistically, we could get vastly different number of data points for the nine debugging tasks. In particular, we could get a low number of data points for certain debugging tasks (e.g., P1-NO). Obviously, this could compromize the quality of subsequent statistical analysis.

Latin Square Design. However, there is a better solution. A $n\times n$ *Latin square* is an $n\times n$ matrix with n distinct values as entries with the property that no row or column contains the same value twice.

Latin squares present a common solution to the above statistical problem in many experimental setups [24, 2, 6]. Figure 4 depicts a 3×3 Latin square applied to our context. The columns are labelled with the three subject *programs* (P1, P2, P3). The rows are labelled with names of three subject developers (D1, D2, D3). The nine squares in the center contain the three treatments (NO, LO, HI). Now with this design, each developer receives all three treatments listed in his row, for all three subject programs listed in the headers of the corresponding columns.

We apply the *Latin square design* to ensure the same number of data points for all debugging tasks, without compromising control over the confounding factors. There are 12 distinct 3×3 Latin squares, modulo swapping rows and swapping columns. For each three developers, we randomly pick one of these 12 Latin squares and randomize also the assignment of developers and programs to rows and columns. The result is the same number of data points for all debugging tasks, without compromising control over the confounding factors such as developer personal competence or program complexity. Still, each combination of treatments and programs is equally probable for each developer (in any order of programs and in any order of treatments). For N=69 participants, each performing three out of nine tasks, we get exactly 23 data points for each of the nine debugging tasks.

Technically, our experiment is an instance of the so-called *within-group design* in which all subjects are exposed to every treatment. We apply our treatments (independent variable: variability with three levels: NO, LO, and HI) to the subjects (programs and developers). In addition, we distinguish the tasks for each program since the programs are not equivalent. We measure our dependent variables; time and number of correct and incorrect answers for bug-finding tasks.

Data Analysis. We used Anova [6] to test significance of differences between different treatments. Anova is heavily used in controlled experiments and useful in comparing three or more means for statistical significance. Anova requires a normal distribution, variance homogeneity, and model additivity of the samples. We check these assumptions using the Box Cox, Bartlett, and Tukey tests, respectively. We conventionally consider a factor as significant when a p-value < 0.05.

3.5 Execution

Before the actual large-scale experiment on 69 subjects, we executed a *pilot* study with a small group of local students to assess our design and tasks. The results of the pilot are not considered in our analysis and the number 69 is excluding the pilot study. Based on the pilot feedback, we changed mainly the presentation of the tasks.

Before the subject developers were confronted with their tasks, we presented a simple tutorial on the basics of variability; in particular, features, configurations, and compile-time conditional statements. Also, we demonstrated how to solve a small *warm-up task* to demonstrate the nature of the tasks and what kind of answers are expected. The warm-up task was inspired by Figure 2 in a paper by Liebig et al. [20].

We randomly generated 23 Latin squares as described above. We then used the Latin squares to compile a *task description sheet* for each participant with their three relevant debugging tasks (e.g., first P2-LO, second P1-HI, third P3-NO). Every participant then performs the debugging task (i.e., find the bug) for the three given programs, in sequence.

All task description sheets contained instructions and a link to an online form for completing the task. We ensured that each program fits in a single screen to avoid participants scrolling up and down to see the entire code. We prepared all tasks using Google forms to avoid heterogeneous environments and installing software on different machines. That is, we provided to the participants only a static window, i.e., no IDEs, no tools, no navigation. We recorded timestamps for each of the participants when they start and finish, allowing us to calculate the duration of their debugging. In addition to time, after the experiment, we calculated the number of correct, incorrect, and partially correct answers.

We eliminated participants who left the experiment early without completing the task, and returned their Latin square assignments to the pool of the rows available for further random allocation. The eliminated (unmotivated) participants are not included in the 69 figure. No other deviations happened during execution.

In addition to the quantitative results, we conducted semi-structured interviews after the experiments. This was to get qualitative feedback on how the participants approached the debugging tasks, particularly the ones involving a HI degree. We asked two questions: (i) How did you go about finding the HI bug? and (ii) What were the difficulties?

4. RESULTS

We now present the results of our experiment and discuss the implications. We make eight observations addressing the research questions—the impact of variability on the *time* and *accuracy* of bug finding. Before proceeding, we stress that the observations should not be generalized far beyond the degree of variability for which we ran the experiment; i.e., $|\mathbb{F}| \leq 3$. We will elaborate on external validity in Sect. 5.4.

All experiment materials are available online at http://itu.dk/people/jeam/variability-experiment/ (including data, programs, task descriptions, and statistical processing scripts).

4.1 How does the degree of variability affect the time of bug-finding? (RQ1)

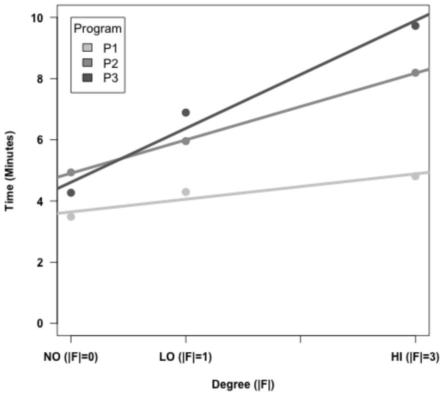


Figure 5: Mean bug-finding time (along the y-axis in minutes) as a function of the degree of variability (x-axis).

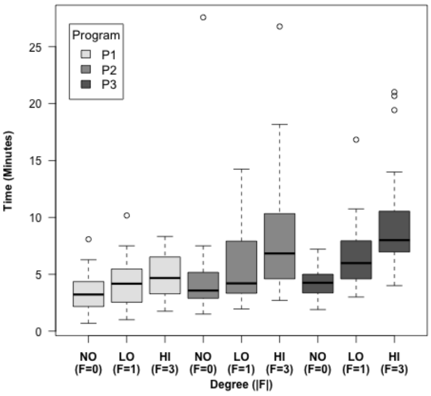


Figure 6: The distribution of bug-finding time.

We consider the first research question now.

Observation 1: *Mean bug-finding time appears to increase linearly with the degree of variability.*

Figure 5 plots the mean bug-finding times (in minutes, along the y-axis) for each of our three benchmark programs. Each dot depicts the mean time to find the bug, for a particular program (`P1`, `P2` and `P3`), for a particular degree of variability, i.e., `NO` ($|\mathbb{F}| = 0$), `LO` ($|\mathbb{F}| = 1$), and `HI` ($|\mathbb{F}| = 3$). For instance, the fastest mean bug-finding time is about $3\frac{1}{2}$ minutes (for program `P1` with `NO` variability), whereas the slowest mean bug-finding time is a bit less than 10 minutes (for program `P3` with a `HI` variability degree of $|\mathbb{F}| = 3$). For each program, we fit a regression line to its respective points. The lines suggests that the mean bug-finding time increases *linearly* with the degree of variability. According to an ANOVA test, the difference between bug-finding times for distinct degrees of variability is statistically significant, with p-value $= 2.0 \times 10^{-8}$. Also bug-finding time is a linear function of programs and degrees, with p-value $= 3.6 \times 10^{-9}$, by F-test for regression.

Recall that the number of variant programs to be considered by a participant grows *exponentially* with the degree of variability (i.e., $|\mathbb{K}| = 2^{|\mathbb{F}|}$, assuming all variants constitute valid programs). Clearly, a developer has to somehow consider each of the $2^{|\mathbb{F}|}$ variants in order to make an accurate diagnosis of the bug. Afterall, each of the variants may or may not harbour a bug. One might then, in fact, suspect that bug-finding time ought to increase *exponentially* with the degree of variability.

The post-treatment interviews provide qualitative insights into how the participants approached the problem and what difficulties they faced in understanding programs with a `HI` variability degree. The participants agreed that finding bugs in the `NO` programs, so without variability, required less effort than in programs with `HI` degree of variability. One participant explains:

> *"I tried to keep all different paths in mind, but it was especially difficult with multiple colors [`HI`]."*

Along the same lines, another participant says:

> *"With more variability [`HI`] you need to build up exponentially more traces in your head."*

The participants analyze programs as one unit despite variability. They do *not* split the task into analysis of exponentially many independent programs, one variant at a time. An unconscious use of brute force would yield a $2^{|\mathbb{F}|}$ factor slow down in overall bug-finding time.

Hick's Law [15] from psychology, based on so-called *choice-reaction-time experiments*, explains that the amount of time for a human response increases logarithmically with the number of possible choices. Compared to a baseline program with `NO` variability, programs with higher degrees of variability involve exponentially more choices to be made. Obviously, composing an exponential function with a logarithmic one yields a linear function. We thus hypothesize that the seemingly linear increase in bug-finding time, in spite of the exponential blow up, can be attributed to Hick's Law.

Presumably, the more complex the variability of a program, the more time it would take to find bugs in that program. Indeed, Fig. 5 is consistent with this expectation: the slopes of the lines are ordered according to the complexity of the respective programs. Recall from Figure 1 that programs `P1`, `P2`, and `P3` were increasingly complex both in terms of variability-unaware and variability-aware characteristics. Also, we remark that, even if we exclude participants that failed to correctly identify the bug, we see a picture similar to that of Fig. 5.[3]

In summary, the first observation indicates that an increase in variability (e.g., by adding features) complicates bug finding, but not dramatically and not prohibitively so. This is a very positive finding, that is consistent with existence of software products with hundreds, even thousands, of features, testifying that developers in the trenches *are* able to deal with variability.

Observation 2: *The variance of bug-finding time appears to be amplified by the degree of variability.*

Figure 6, we plot the distribution of bug-finding times for each program and variability degree. Each box encapsulates the middle 50% data points. The lower and upper limit of the box respectively represent the lower and upper quartiles (the 25% and 75% percentiles). The upper and lower whiskers represent the data above and below the middle half of the data. The horizontal line within the box draws up the *median*

[3] The diagram can be found in the accompanying materials.

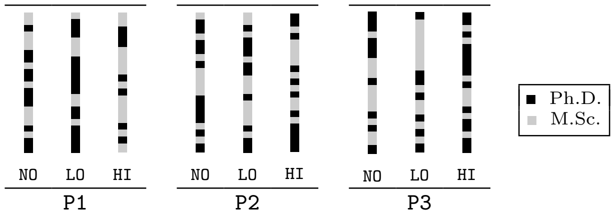


Figure 7: Ranking of *fastest* (at the bottom) to *slowest* (top) participants according to their educational level (i.e., M.Sc. vs. Ph.D. students).

of the data points. Finally, the circles above the boxes visualize outliers. For instance, for program P3 (the three rightmost boxes), the middle half of the participants spent between $3\frac{1}{3}$ and 5 minutes to find the bug with NO variability, whereas, for HI variability, the middle half spent from about 7 to $10\frac{1}{2}$ minutes. Again, considering only participants that found the bug yields a similar diagram, consistent with the above.[4]

Amplification of variance is a predictable consequence of our first finding. For the *variance* of a stochastic variable, X, multiplied by a constant factor, c (depending on the degree of variability), we have that: $\mathrm{Var}(c\,X) = c^2\,\mathrm{Var}(X)$.

In popular terms, this observation means that differences in bug-finding competences are amplified when working with variability. Ultimately, this means that getting talented developers on such projects is important.

> Observation 3: *Ph.D. students appear to not be faster at finding variability bugs than M.Sc. students.*

Figure 7 shows the ranking of bug-finding from the *fastest* participants (towards the bottom) to the *slowest* participants (towards the top), abstracting away the actual time they spent debugging. M.Sc. students are shown in light gray, Ph.D. students in black. There appears to be no pattern of one group of students being faster than the other, even for higher degrees of variability.

In the late 1980es, Oman et al. [25] compared debugging abilities of novice, intermediate, and skilled student programmers using two programs written in Pascal. Among other things, they found that experienced programmers find errors faster than less experienced programmers. We, in turn, do not notice any difference in terms of bug-finding time between Ph.D. and M.Sc. students. This can be explained by the level of subjects in that study. They considered only undergraduate students, separating them according to the amount of computer science courses taken, whereas we test with graduate students, who would likely be considered as skilled programmers in their setup. Furthermore, we study a different phenomenon, which is variability, that might be challenging independently of education level.

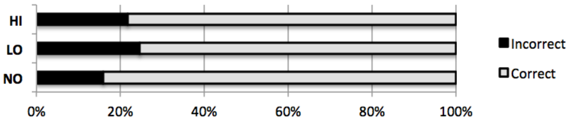


Figure 8: Ratio of *incorrectly* vs. *correctly* identifying a bug.

4.2 How does the degree of variability affect the accuracy of bug-finding? (RQ2)

We now turn to our second research question (RQ2) on the *accuracy* of bug-finding:

> Observation 4: *Most developers correctly identify bugs in programs regardless of the degree of variability.*

Figure 8 shows shows what percentage of developers were able to find the bugs correctly. The *incorrect* answers are black, and the *correct* ones are gray. The data is presented for each degree of variability separately. The frequency of incorrect answers is consistently low, with around a fifth being the incorrect answers. For programs with NO variability, 16% of subjects (11 out of 69) did not find the bug. Even for the HI variability programs, only 22% of the subjects (15 out of 69) answered incorrectly.

Generally, developers seem to be good at finding bugs in programs—and in programs with variability (at least, up to three features). Interestingly, more than half (38 out of 69) of the participants correctly identified the bug in *all* three tasks. On average, if we disregard the variability degrees, 79% of the participants were able to correctly find the bug. All in all, we conclude that finding bugs in programs seems to not be significantly affected by the degree of variability (at least for $|\mathbb{F}| \leq 3$).

> Observation 5: *Many developers fail to exactly identify the set of erroneous configurations, already for a low degree of variability.*

We now look a little closer at accuracy and split the *correct* answers in two sets. If the participant got the set of erroneous configurations exactly right, we classify her answer as *fully correct*. Similarly, we classify answers as *partially correct*, if the developer has correctly identified the bug, but failed to correctly specify the set of configurations in which the error occurred (missing some configurations or listing too many). We ignore incorrectly identified bugs for this part of the analysis, as it is hard to interpret the identification of configurations for them. For instance, program P3 with HI variability, contains an *assertion error* that occurs in two (out of eight) configurations. For this task, some participants found only one of the erroneous configurations and others listed extra configurations for which the error does not occur. Figure 9, presents the numbers of fully and partially correct answers at different levels of variability.

Obviously, partial correctness does not make sense for programs without variability (for NO we have only one possible configuration). Already for LO variability (one feature), we see that the number of partially correct answers quickly rises to 17% (9 out of 52). For HI variability, this number escalates to almost 40% (20 out of 54).

Identifying the exact set of erroneous configurations seems to become difficult already for $|\mathbb{F}| = 3$ (HI variability). Doing this requires understanding the combinations of features that enable the incriminated execution paths—a form combinatorial reasoning, which apparently becomes difficult fast. Such problems are notoriously hard for humans. For realistic systems, where a feature model additionally shapes the set of legal configurations, this task would presumably be *even harder* (as one needs to reason about feature model

[4] See the accompanying website for more information.

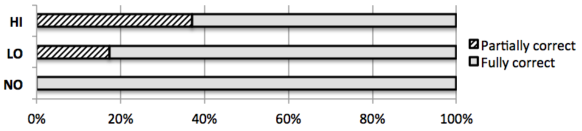


Figure 9: Ratio of *partially correctly* versus *fully correctly* identifying a bug.

constraints, in addition).

From a prior qualitative study [1], we know that programming errors related to variability appear due to inability of programmers to correctly reason about all variations of the program that they are modifying. Those findings are consistent with the above: it is plausible that developers mis-identify the sets of configurations during programming tasks and during debugging tasks for the same reasons. To the best of our knowledge, this study presents the first quantitative confirmation that indeed reasoning about multiple configurations is a challenge, even for relatively small sets.

> Observation 6: *For higher degrees of variability, it appears to be more difficult to correctly identify the set of erroneous configurations than to find the bug in the first place.*

For HI variability, we saw that 22% (15 out of 69) did not find the bug (see Figure 8). Among the ones that did, a staggering 37% (20 out of 54) erred on set of erroneous configurations (cf. Figure 9).

Although the participants were only asked to *find* the bugs, not (also) *fix* them, we find that our results are consistent with studies of creating and fixing bugs. Yin and coauthors report that in general bug fixers "*may forget to fix all the buggy regions with the same root cause.*" [35]. Our earlier study [1] also reports that bugs are introduced because the programmers do not realize the complexity of all the configurations in which their code will run.

> Observation 7: *Ph.D. students appear to be more accurate at finding variability bugs than M.Sc. students.*

Figure 10a compares the ratio of correct-to-incorrect answers according to educational level, separating M.Sc. students and Ph.D. students. We see that the number of *incorrect* answers for Ph.D. students are consistently low. In fact, even for HI variability, only 10% of the Ph.D. students answered incorrectly. For M.Sc. students, the numbers are consistently higher. For HI variability, the number of incorrect answers are more than three times higher at 35%. On average, the Ph.D. students found bugs three times more accurately than M.Sc. students for all degrees of variability. Presumably, Ph.D. students, having more education, are more careful and meticulous when debugging than M.Sc. students.

Interestingly, Ph.D. students prevail only as far as identifying the actual bug is concerned, but they are not better in identifying the relevant set of configurations. For both M.Sc. and Ph.D. students the percentage of fully correct answers seems to not significantly be impacted by variability:

> Observation 8: *Identifying the exact set of erroneous configurations is hard regardless of education (both for M.Sc. and Ph.D. students).*

Figure 10b shows the frequency of *partially correct* answers for M.Sc. versus Ph.D. students. For HI variability, for instance, we see that 40% of M.Sc. students answered *partially incorrect* versus 35% for that of Ph.D. students.

The numbers testify that the combinatorial task of identifying the exact combination of features provoking an error is difficult, regardless of educational level. We see the frequency of *partially correct* answers double from LO to HI degree in both groups of students.

5. THREATS TO VALIDITY

5.1 Internal validity

Choice of variability degrees? We chose *zero*, *one*, and *three* features for pragmatic reasons. If we instead had chosen, for instance, *two*, *four*, and *six* features, the experiments would have required much longer time, discouraging and tiring participants. In fact, the mean time for program P3 with three features is almost ten minutes. Also, five participants spent more than twenty minutes to find the bug in programs involving three features.

Choice of language? In this experiment, we adopted Java as our programming language. This is because we want to run the experiment with as many students as possible, and Java is well-known among students in Denmark. We only admitted students who had experience with Java.

Use of background color? Researchers have shown that background colors may improve program comprehension and subjects favor background colors [11]. Additionally, the benefits of colors compared to text-based annotations is that one can clearly distinguish background colors (feature code) from base code and humans are able to recognize colors faster than text [13]. Thus, we decide to use background colors instead of preprocessor directives.

Choice of colors? Before the experiment, we check for color blindness among the participants. We also take care of choosing colors that are clearly distinguishable (blue, green, and yellow). Aside from this, we do not believe that the exact choice of colors matter so much,[5] for our experiment.

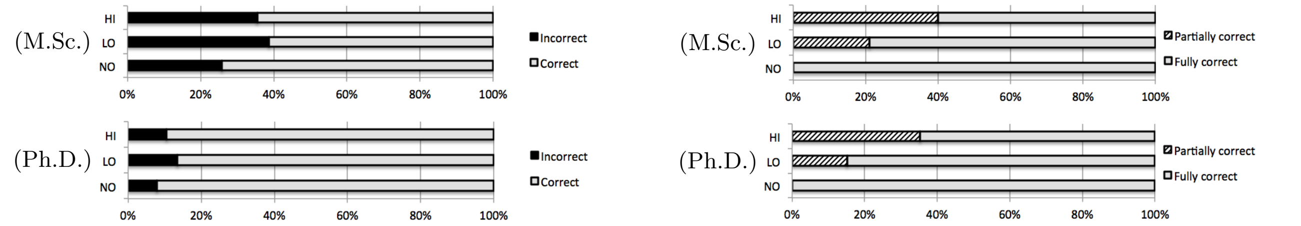
Selection bias? To minimize selection bias, we randomly assign participants, degrees, and programs into the Latin squares. So, we control our confounding factors via Latin square design and randomization. Every participant takes all treatments, including all the three programs. We did have few (unmotivated) students who left the experiment early without completing the tasks. As mentioned earlier, we eliminated them from the study and returned their Latin square assignments to the pool of the rows available for further random allocation.

Participation incentives? We offered a chocolate bar as a participation incentive. Aside from that, only pride prohibits participants from deliberately performing poorly. We found no indications of participants giving deliberately silly answers.

5.2 Conclusion validity

Statistical tests? In this experiment, we use Anova (including verification of its assumptions) to verify whether or not the means of our test groups are equal by analyzing variance. In fact, Anova is heavily used in controlled experiments and useful in comparing three or more means for statistical significance. Besides that, we are comparing

[5] The fashion industry may disagree.



(a) Ratios of *incorrect* vs *correct* answers for M.Sc. students (above) and Ph.D. students (below).

(b) Ratios of *partially correct* vs *fully correct* answers for M.Sc. students (above) and Ph.D. students (below).

Figure 10: Accuracy of bug-finding according to educational level (M.Sc. students vs Ph.D. students).

group means against each other, not specific subjects, which decreases the impact of developer competence.

Time measured? The time measured to complete a test involves both thinking as well as writing down the answer. We asked the participants to write down the bug kind, line, and a list of erroneous configurations (a configuration is written as a list of enabled features). Obviously, this might interfere in the measured time due to variations in writing speed. However, we observed that the task of writing usually took tens of seconds whereas thinking took on the order of minutes. Hence, the speed of the task of writing is dwarfed by the thinking. Note that we exclude the time of reading the tasks, as we only start the timer once the participants have read each task description (i.e., when they see the programs).

5.3 Construct validity

Do participants know what to do? Before exposure to the programs with variability, we explained the basics of variability (including features, compile-time conditional statements, and configurations). In addition, we performed a warm-up task with different degrees of variability with them in order to demonstrate what they need to do and what they need to answer (including the format of answers). In summary, we essentially taught them how to accomplish the tasks and fill in the forms.

Disregarding incorrect answers? When analyzing response times, we decided not to exclude wrong answers because we wanted to measure the time it takes to debug a program, whether correct or not. Note, however, that we do check that our observations still hold (are stable) when disregarding incorrect answers.[6] Recall that the number of incorrect answers were consistently low, regardless of the degree of variability (cf. OBSERVATION 4, Figure 8 and 10a).

5.4 External validity

Beyond CIDE and preprocessors? Our experiment and entire study is dedicated and tailored to a particular technique for dealing with variability: preprocessor. Our observations generalize to `#ifdef`s instead of background colors because of their close relationship (after all, CIDE is based on `#ifdef`s) [16] and known results shows that a judicious use of colours instead of `#ifdefs` can only simplify the task [11]. Generalization to other variability techniques is not intended.

Beyond university students? The main question is whether our study is also relevant to the industry? Our N=69 participants were predominantly M.Sc. and Ph.D. students from three different Danish universities. All had JAVA programming experience and around half of them had industrial experience (few months to several years). Note that we had participants from Africa, Asia (incl., The Middle East), Europe, North, and South America. In addition, previous research has established that graduate students make good proxies for industry developers [7]. All of this contributes to representativity and generalization to "real-world" industrial developers.

Beyond our buggy programs? We based our programs (`P1`, `P1`, and `P3`) on real variability bugs from real highly-configurable systems (LINUX [1], BUSYBOX,[7] and BESTLAP [28], respectively) precisely to minimize the risks of introducing and studying artificial problems. Also, the programs were qualitatively different (cf. Figure 3). Further, our bug-finding tasks present three qualitatively different types of bugs: *uninitialized variable*, *null-pointer dereference*, and *assertion violation*. In fact, these are common variability bugs in bug reports [1]. For these reasons, we expect the results should transfer to other (smaller) programs. Of course, there may be additional effects, unaccounted for, when debugging programs beyond 35 lines.

Beyond lab settings? More programs, larger programs, higher degrees, realistic programs and tools, all extend the task duration beyond one hour and make it significantly harder to attract anywhere near 69 participants. For these reasons, we optimized for internal validity and quantitative observations in lab conditions, recognizing that there is an inherent tradeoff between internal and external validity in experiment design [32]. We ensured that each program fits in a single screen to avoid participants scrolling up and down to see the entire code. Additionally, we prepared all tasks using Google forms to avoid heterogeneous environments and installing software on different machines. That is, we provided to the participants only a static window, i.e., no IDEs, no tools, no navigation.

Beyond three features? It is entirely likely that the linear relationship we observed (in OBSERVATION 1) breaks down, at some point, for some higher degrees of variability. Presumably at some point, developers will be unable to simply cope with the exponentially many combinations of features. However, this is beyond the scope of our study.

6. RELATED WORK

Variability Bugs. Previous studies have shown negative aspects of preprocessor usage such as code pollution, no separation of concerns, and error-proneness [33, 18, 10, 9, 17,

[6] See the accompanying website for more information.

[7] http://git.busybox.net/busybox/commit/?id=5cd6461b6fb51e8cf297a49074fce825e1960774

19]. These studies are predominantly artifact-based (so based on studying programs), not investigating human abilities to work with the code.

Recently, Medeiros and co-authors interviewed 40 developers to study their perceptions of the C preprocessor [22]. The developers assess that preprocessor-related bugs are easier to introduce, harder to fix, and more critical than other bugs. Many admit that they check only a few configurations of the source code in practice when testing their implementations. Our experiment confirms these qualitative insights and complements them with quantitative data.

Medeiros et al. [23] investigated syntactic errors in preprocessor-based systems. They noticed that developers introduce syntax errors when changing existing code and adding preprocessor directives, and that some of the relevant errors survive in the life cycle all the way to the release stage. In this paper, we work with human developers (not artifacts), which allows us to quantify the effort of bug finding. Also, we are concerned not with syntactic but with semantic errors.

Ribeiro et al. [27] conducted a controlled experiment to evaluate whether emergent interfaces reduce effort and number of errors during code-change tasks involving feature code dependencies. In general, they found a decrease in code-change effort and number of errors when using their tool support. Emergent interfaces are an example of tooling that attempts to simplify reasoning about variability. Our experiment confirms the need for more research on such tools.

Bug Finding. Oman et al. [25] compared debugging abilities of novice, intermediate, and skilled student programmers using two Pascal programs. Among other things, they found that programmers' ability to find errors increases with general programming experience; they become faster and make fewer mistakes. Comparing to our study, we did not design the experiment to directly compare novices versus experts even though we discussed some indications. Our observations suggest that Ph.D. students are not faster at finding variability bugs than M.Sc. students. But, the former appear to be more careful and meticulous when debugging than the latter. Furthermore, we studied a different phenomenon, which is variability, trying to measure the impact of the degree of variability on debugging.

Program comprehension. Feigenspan et al. [11] in a series of controlled experiments show that use of distinct background colors improves comprehension of `#ifdefs`. This is one important reason why are we using colours in the experiment, instead of preprocessor directives. In accordance with their work, the bug finding with actual `#ifdef` directives should likely be slower than with colours.

Another controlled experiment applied *functional magnetic resonance imaging* (fMRI) to measure program comprehension [31]. They found that five different brain regions associated with working memory, attention, and language processing become activated for comprehending source code. However, variability was not in their focus. We, in turn, focused on quantifying the effect of the degree of variability on debugging.

Schulze et al. [30] studied the influence of the discipline of preprocessor annotations on program comprehension. They found that the discipline of annotations has no influence at all. Our observations agree in that finding bugs with variability is time-consuming and difficult. We are however able to quantify the increase of difficulties when the degree of variability grows.

7. CONCLUSION

We have presented a controlled experiment quantifying the impact of variability on the *time* and *accuracy* of bug finding in highly-configurable systems. We observe that bug-finding *time* appears to increase linearly with the degree of variability. This conclusion is both positive and negative. An increase in variability complicates bug finding (negative), but not dramatically so (positive)—if developers reasoned about each of the variants separately we would have observed an exponential, not linear, growth. The practical implication is that it is beneficial to introduce variation points into programs from the debugging perspective: It is beneficial to pay a linear price for bug finding, if the alternative is to maintain a super-linear set of variants (at least up to three variations in a file). However, there might be benefits in selecting designs (architectures and algorithms) that require less variability, if possible.

Somewhat expectedly, the variance in bug-finding time is amplified by variability. In other words, differences in bug-finding competences of developers appear to be amplified when working on software projects with variability. Getting talented developers for such projects might be important.

We also find that most participants correctly identify bugs in programs with accuracy, that is independent of the degree of variability. However, developers often fail to exactly identify the set of erroneous configurations, and this happens already for a rather low number of features, and gets worse with the degree of variability increasing. Clearly, reasoning about multiple configurations is a challenge. This is consistent with earlier qualitative indications that variability bugs appear, when developers unintentionally ignore an execution that is enabled by an unexpected (for them) configuration of features.

In fact, our study suggests that, for higher degrees of variability, it is more difficult to correctly identify the set of erroneous configurations than to find the bug in the first place. This is rather unexpected, given that to understand the bug one needs to reason about control flow, a temporal non-local phenomenon that is not obviously simpler than combinatorics. This means that it is beneficial to work on support tools that help developers to navigate the configuration space (on top of flow-oriented bug finders).

The future follow up on this work, is expected to design tools exploiting the results of the study, in particular indicating the sets of configurations impacted by a program change, in order to simplify reasoning about all flows that a change participates in (cf. OBSERVATIONS 5 and 8). Additionally, further research like replicating this experiment —with more programs, larger programs, more subjects, higher degrees, realistic programs and tools— is required to confront our observations and to draw new ones. It would be also interesting to replicate our study using fMRI or eye-tracking to better explain the impact of variability on debugging. With this, it would be possible to actually see how developers approach programs with different degrees of variability.

Acknowledgements. We thank all the students for devoting their time. Fritz Henglein, Christian Probst, Søren Debois helped to attract the participants. We thank participants of the FOSD'15 Meeting for valuable feedback. Melo is funded by Brazilian Science without Borders Programme, CNPq grant no. 249020/2013-0. Wąsowski is funded by The Danish Council for Independent Research, grant no. 0602-02327B.

8. REFERENCES

[1] I. Abal, C. Brabrand, and A. Wasowski. 42 Variability Bugs in the Linux Kernel: A Qualitative Analysis. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ASE '14, pages 421–432, New York, NY, USA, 2014. ACM.

[2] R. A. Bailey. *Design of comparative experiment.* Cambridge University Press, 2008.

[3] T. Berger, D. Nair, R. Rublack, J. M. Atlee, K. Czarnecki, and A. Wasowski. Three cases of feature-based variability modeling in industry. In *ACM/IEEE 17th International Conference on Model Driven Engineering Languages and Systems (MODELS)*, 2014.

[4] T. Berger, R. Rublack, D. Nair, J. M. Atlee, M. Becker, K. Czarnecki, and A. Wąsowski. A survey of variability modeling in industrial practice. In *Proceedings of the Seventh International Workshop on Variability Modelling of Software-intensive Systems*, VaMoS '13, pages 7:1–7:8, New York, NY, USA, 2013. ACM.

[5] T. Berger, S. She, R. Lotufo, A. Wasowski, and K. Czarnecki. A study of variability models and languages in the systems software domain. *Software Engineering, IEEE Transactions on*, 39(12):1611–1640, Dec 2013.

[6] G. E. P. Box, J. S. Hunter, and W. G. Hunter. *Statistics for Experimenters: design, innovation, and discovery.* Wiley-Interscience, 2005.

[7] R. P. Buse, C. Sadowski, and W. Weimer. Benefits and barriers of user evaluation in software engineering research. *ACM SIGPLAN Notices*, 46(10):643–656, October 2011.

[8] P. Clements and L. Northrop. *Software Product Lines: Practices and Patterns.* Addison-Wesley, 2002.

[9] M. D. Ernst, G. J. Badros, and D. Notkin. An empirical analysis of C preprocessor use. *IEEE Transactions on Software Engineering*, 28:1146–1170, 2002.

[10] J. M. Favre. Understanding-in-the-large. In *Proceedings of the 5th International Workshop on Program Comprehension (WPC)*, pages 29–38. IEEE Computer Society, 1997.

[11] J. Feigenspan, C. Kästner, S. Apel, J. Liebig, M. Schulze, R. Dachselt, M. Papendieck, T. Leich, and G. Saake. Do background colors improve program comprehension in the #ifdef hell? *Empirical Softw. Engg.*, 18(4):699–745, Aug. 2013.

[12] M. Goedicke, K. Pohl, and U. Zdun. Domain-specific runtime variability in product line architecture. *Proceedings of Object-Oriented Information Services (OOIS'02).: Lecture Notes in Computer Science*, 2002.

[13] E. Goldstein. *Sensation and Perception.* Cengage Learning Services, 2002.

[14] P. Goodliffe. *Becoming a Better Programmer.* O'Reilly Media, Inc., 2014.

[15] W. E. Hick. On the rate of gain of information. *Quarterly Journal of Experimental Psychology*, 4(1):11–26, 1952.

[16] C. Kästner. Virtual separation of concerns: Toward preprocessors 2.0, 5 2010. Logos Verlag Berlin, isbn 978-3-8325-2527-9.

[17] C. Kästner, S. Apel, and M. Kuhlemann. Granularity in Software Product Lines. In *Proceedings of the 30th International Conference on Software Engineering (ICSE)*, pages 311–320. ACM, 2008.

[18] M. Krone and G. Snelting. On the inference of configuration structures from source code. In *Proceedings of the 16th International Conference on Software Engineering (ICSE)*, pages 49–57. IEEE Computer Society Press, 1994.

[19] J. Liebig, S. Apel, C. Lengauer, C. Kästner, and M. Schulze. An analysis of the variability in forty preprocessor-based software product lines. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering (ICSE)*, pages 105–114. ACM, 2010.

[20] J. Liebig, A. von Rhein, C. Kästner, S. Apel, J. Dörre, and C. Lengauer. Scalable analysis of variable software. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 81–91, New York, NY, USA, 2013. ACM.

[21] T. J. McCabe. A complexity measure. *IEEE Trans. Softw. Eng.*, 2(4):308–320, July 1976.

[22] F. Medeiros, C. Kästner, M. Ribeiro, S. Nadi, and R. Gheyi. The love/hate relationship with the C preprocessor: An interview study. In *Proceedings of the 29th European Conference on Object-Oriented Programming (ECOOP)*, Lecture Notes in Computer Science, Berlin/Heidelberg, 2015. Springer-Verlag.

[23] F. Medeiros, M. Ribeiro, and R. Gheyi. Investigating preprocessor-based syntax errors. In *Proceedings of the 12th International Conference on Generative Programming: Concepts & Experiences*, GPCE, pages 75–84. ACM, 2013.

[24] D. C. Montgomery. *Design and Analysis of Experiments.* John Wiley & Sons, 2006.

[25] P. W. Oman, C. R. Cook, and M. Nanja. Effects of programming experience in debugging semantic errors. *J. Syst. Softw.*, 9(3):197–207, Mar. 1989.

[26] K. Pohl, G. Bockle, and F. J. van der Linden. *Software Product Line Engineering.* Springer, 2005.

[27] M. Ribeiro, P. Borba, and C. Kästner. Feature maintenance with emergent interfaces. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 989–1000, New York, NY, USA, 2014. ACM.

[28] M. Ribeiro, F. Queiroz, P. Borba, T. Tolêdo, C. Brabrand, and S. Soares. On the impact of feature dependencies when maintaining preprocessor-based software product lines. In *Proceedings of the 10th ACM International Conference on Generative Programming and Component Engineering (GPCE)*, pages 23–32. ACM, 2011.

[29] C. Sant'anna, A. Garcia, C. Chavez, C. Lucena, and A. v. von Staa. On the reuse and maintenance of aspect-oriented software: An assessment framework. In *Proceedings XVII Brazilian Symposium on Software Engineering*, 2003.

[30] S. Schulze, J. Liebig, J. Siegmund, and S. Apel. Does the discipline of preprocessor annotations matter?: A controlled experiment. In *Proceedings of the 12th International Conference on Generative Programming: Concepts & Experiences*, GPCE '13, pages 65–74, New York, NY, USA, 2013. ACM.

[31] J. Siegmund, C. Kästner, S. Apel, C. Parnin,

A. Bethmann, T. Leich, G. Saake, and A. Brechmann. Understanding understanding source code with functional magnetic resonance imaging. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 378–389, New York, NY, USA, 2014. ACM.

[32] J. Siegmund, N. Siegmund, and S. Apel. Views on internal and external validity in empirical software engineering. In *Software Engineering (ICSE), 2015 IEEE/ACM 37th IEEE International Conference on*, volume 1, pages 9–19, May 2015.

[33] H. Spencer and G. Collyer. #ifdef considered harmful, or portability experience with C news. In *Proceedings of the Usenix Summer Technical Conference*, pages 185–198. Usenix Association, 1992.

[34] M. Völter. Handling variability. In *Proceedings of the 14th annual European Conference on Pattern Languages of Programming*, EuroPLoP'09. Kelly & Weiss, 2009.

[35] Z. Yin, D. Yuan, Y. Zhou, S. Pasupathy, and L. Bairavasundaram. How do fixes become bugs? In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering (ESEC/FSE)*, pages 26–36. ACM, 2011.

[36] B. Zhang and M. Becker. Recovar: A solution framework towards reverse engineering variability. In *Product Line Approaches in Software Engineering (PLEASE), 2013 4th International Workshop on*, pages 45–48, May 2013.

Variability through the Eyes of the Programmer (Paper 1B)

Variability through the Eyes of the Programmer

Jean Melo, Fabricio Batista Narcizo, Dan Witzner Hansen,
Claus Brabrand, Andrzej Wasowski

IT University of Copenhagen, Denmark
Email: {jeanmelo, fabn, witzner, brabrand, wasowski}@itu.dk

***Abstract*—Preprocessor directives (`#ifdefs`) are often used to implement compile-time variability, despite the critique that they increase complexity, hamper maintainability, and impair code comprehensibility. Previous studies have shown that the time of bug finding increases linearly with variability. However, little is known about the cognitive process of debugging programs with variability. We carry out an experiment to understand *how* developers debug programs with variability. We ask developers to debug programs *with* and *without* variability, while recording their eye movements using an eye tracker.**

The results indicate that debugging time increases for code fragments containing variability. Interestingly, debugging time also seems to increase for code fragments without variability in the proximity of fragments that do contain variability. The presence of variability correlates with increase in the number of gaze transitions between definitions and usages for fields and methods. Variability also appears to prolong the "initial scan" of the entire program that most developers initiate debugging with.

***Keywords*-Variability, Preprocessors, Debugging, Eye Tracking, Highly-Configurable Systems**

I. Introduction

Many modern software systems are highly configurable. They embrace variability to increase adaptability, to extend portability across different hardware, and to lower cost. Highly-configurable systems include both large industrial product lines [1], [2], [3] and open-source systems. In some cases, such as the LINUX kernel, thousands of configuration options (*features*) controling the compilation process are used [4].

Although bringing important benefits, variability also comes at a cost. It makes reasoning about programs more difficult [5]. As a consequence configuration-dependent (*variability*) bugs appear [6], [7]. Previous studies [5], [8] have shown that debugging is hard and time consuming in the presence of variability. Specifically, our prior study [5] revealed that the time it takes developers to find a bug increases linearly with the number of features, while the ability to actually find the bug is relatively independent of variability. Also, identifying the exact set of configurations in which a bug manifests itself appears to be difficult already for a low number of features.

These prior studies focus on *quantitative* questions only, analyzing debugging time and correctness. There is little evidence in the literature on *how* developers debug programs with variability. In this paper, we describe an eye-tracking experiment with follow-up interviews to study more precisely *how* developers approach and debug programs with variability. We use simplified versions of real buggy programs taken from two highly-configurable systems: BUSYBOX and BESTLAP. While the developers debug program versions *with* and *without* variability, we record their eye movements using an eye-tracking device.

Eye movement data constitute a continuous, non-interruptive process measure that can proxy for ongoing mental processes and human activities. Eye movements are intimately linked to the allocation of attention and can be guided by low-level and high-level factors (e.g., [9]). The most commonly studied aspects of eye movement behavior are saccades and fixations, but several additional specialized eye movements that provide different information also exist. Which features attract attention is subject to continued debate and research [9]. Eye tracking has a wide variety of applications outside code comprehension studies. It has been used for medical diagnosis (e.g., Alzheimer's), communication tool for people with severe disability and measures of workload, fatigue and stress levels [10], [9].

Source code comprehension is a multi-level process that involves visual processing, as well as mental encoding and representing the program's source code [11], [12], [13]. Programming involves a series of tasks but with two processes common to all: reading the code (chunking) and searching through the code (tracing). In practice, programmers rarely chunk every statement in a program but the programmer searches for task specific relevant code [11], [13], [14], [15]. Analyzing what the programmer looks at while programming can be monitored through an eye tracker. Eye tracking is used to monitor the eye movements and estimate where the subject is looking (e.g., on a screen) and is typically based on one or more cameras observing the users' eyes [10].

In our study we find that:

- Variability increases debugging time of code fragments that contain variability.
- Debugging time also seems to increase for code fragments without variability in the proximity of fragments that do contain variability.
- Variability makes the number of gaze transitions (also known as *saccades*) grow between definition-usages for fields and call-returns for methods.
- Most developers initiate debugging with an "initial scan" of the program from the first line down to the last, independent of variability. However, variability seems to prolong this "initial scan" of the program disproportionately.
- Developers appear to debug programs with variabil-

978-1-5386-0535-6/17 $31.00 © 2017 IEEE
DOI 10.1109/ICPC.2017.34

```
 1  import java.util.Random;
 2
 3  public class Http {
 4      String subject = null;
 5      int totalLength = 600;
 6      final int HTTP_UNAUTHORIZED = 401;
 7      final int HTTP_NOT_IMPLEMENTED = 501;
 8      boolean LARGE_FORMAT = false;
 9      String REQUEST_GET = "GET";
10
11
12      public void sendHeaders(int responseNum) {
13          if (LARGE_FORMAT) {
14              int buf = 0;
15              buf = totalLength - responseNum;
16              subject = "response header";
17          }
18          if (subject.isEmpty())
19              subject = "Void response";
20          System.out.println("done");
21      }
22
23      private void handleIncoming(String requestType) {
24
25          boolean http_unauthorized = new Random().nextBoolean();
26          if (http_unauthorized)
27              sendHeaders(HTTP_UNAUTHORIZED);
28
29
30          if (!requestType.equals(REQUEST_GET))
31              sendHeaders(HTTP_NOT_IMPLEMENTED);
32
33      }
34
35      public static void main(String[] args) {
36          Http http = new Http();
37          http.handleIncoming("POST");
38      }
39  }
```

(a) *Without* variability.

```
 1  import java.util.Random;
 2
 3  public class Http {
 4      String subject = null;
 5      int totalLength = 600;
 6      final int HTTP_UNAUTHORIZED = 401;
 7      final int HTTP_NOT_IMPLEMENTED = 501;
 8      #ifdef CONFIG_FEATURE_HTTPD_CGI
 9      String REQUEST_GET = "GET";
10      #endif
11
12      public void sendHeaders(int responseNum) {
13          #ifdef CONFIG_LFS
14          int buf = 0;
15          buf = totalLength - responseNum;
16          subject = "response header";
17          #endif
18          if (subject.isEmpty())
19              subject = "Void response";
20          System.out.println("done");
21      }
22
23      private void handleIncoming(String requestType) {
24          #ifdef CONFIG_FEATURE_HTTPD_BASIC_AUTH
25          boolean http_unauthorized = new Random().nextBoolean();
26          if (http_unauthorized)
27              sendHeaders(HTTP_UNAUTHORIZED);
28          #endif
29          #ifdef CONFIG_FEATURE_HTTPD_CGI
30          if (!requestType.equals(REQUEST_GET))
31              sendHeaders(HTTP_NOT_IMPLEMENTED);
32          #endif
33      }
34
35      public static void main(String[] args) {
36          Http http = new Http();
37          http.handleIncoming("POST");
38      }
39  }
```

(b) *With* variability.

Fig. 1: Program `P` *without* and *with* variability.

ity by considering either one configuration at a time (consecutively) or all configurations at the same time (simultaneously).

We hope that our findings will help designers of debugging and developer support tools, and inspire more research to further investigate the interplay between debugging and variability.

II. Motivating Example

Developers of highly-configurable systems often use the C preprocessor (Cpp) to implement compile-time variability using conditional compilation directives (`#ifdefs`) [16], [17]. For this reason, we examine the impact of variability on debugging in the context of Cpp.

Configurable software systems are challenging for developers because code fragments may be conditionally *included* or *excluded* depending on whether particular features are *enabled* or *disabled*. This means that developers need to reason about several different configurations (versions of the program), each with different data- and control-flow in order to understand a program with variability. This impacts debugging. In programs with variability, some errors occur conditionally, only in certain *erroneous configurations* (i.e., when certain combinations of features are enabled/disabled).

Previous studies have demonstrated that debugging is overall difficult and time consuming in the presence of variability [5], [8]. In this paper, we use eye tracking to study more precisely *how* developers debug programs with variability. We ran a classic "find the bug" experiment using programs containing exactly one error and then compare how the developers look at a program *with* variability against a version of it *without* variability (as a baseline).

Figure 1 presents a code scenario extracted from BusyBox which is an open-source highly-configurable system with about 600 features that provides several essential Unix tools in a single executable file. We have adapted the extracted example from C to Java to widen the audience of potential participants for our experiment.

Figure 1a shows the version of this program *without* variability derived from the original version with variability shown in Fig. 1b. The program contains an error in line 18 where evaluation of the expression `subject.isEmpty()` causes a *null-pointer exception* because `subject` has the value `null`. The entry point `main` calls `handleIncoming` in line 37 which, in turn, calls `sendHeaders` in line 27. This method then skips past the statements in lines 14–16 because the variable `LARGE_FORMAT` has the value `false` (line 8). Hence, when we reach line 18, the variable `subject` has never been assigned a proper value aside from its initialization to `null` in line 4.

Figure 1b depicts the original version of the program *with* variability. Notice that the program now contains three so-called *features*: `LFS`, `AUTH`, and `CGI` (names abbreviated). Each of these three features can be designated as either *enabled* or *disabled*. Features are used in conditional compilation directives (`#ifdefs`), which control whether to *include* or *exclude* code before compilation, depending on whether features are *enabled* or *disabled*. For instance, the fragment in lines 14–16 (wrapped in an `#ifdef` and `#endif` in lines 13 and 17) is to be *included* in the code if `LFS` is enabled; and *excluded* if `LFS` is disabled. Since n features yield 2^n distinct configurations, our variability program with three features then comes in eight (2^3) distinct configurations, each corresponding to a different version of the program.

The *null-pointer exception* from before now only appears in specific configurations: whenever we *disable* the feature `LFS` as well as *enable* either `AUTH` or `CGI`. The exception thus occurs in exactly three (out of eight) configurations. The error no longer occurs if we, for instance, *enable* `LFS`; then `subject` is indeed assigned a non-null value in line 16. Also, if we do not *enable* either `AUTH` or `CGI`, `sendHeaders` is no longer invoked in line 27 or 31. The developer must thus somehow consider *all* configurations when debugging a variability program. Further, for a program with variability it is not enough to simply find an error in some configuration. In order to *fix* a bug, a developer must thus not only identify the error, but also correctly identify the exact set of erroneous configurations (combinations of feature enablings/disablings). If the developer gets the configurations wrong, the bug may only be partially fixed. Clearly, this is a difficult task. Combinatorial problems are notoriously difficult.

For these reasons, a developer has to be highly alert and conscious of the features and `#ifdefs` in the code. Previous work has demonstrated to what extent variability complicates debugging. In this paper, we consider *how* variability impacts debugging.

III. Experiment

We have designed a controlled experiment based on eye tracking to investigate and compare how developers debug programs *with* versus *without* variability. We will now explain our experimental design and setup.

A. *Objective*

The experiment aims to investigate the effect of variability on debugging. In other words, we want to understand *how* developers debug programs with variability in comparison with ordinary programs. Specifically, we aim to answer the following research question:

> **RQ:** How do developers debug programs with variability?

To respond to this question, we perform several so-called *"find the bug"* experiments [18] with an eye tracker and analyze how developers go about finding the bug. We are particularly interested in the impact of variability on bug finding from the developer's perspective.

B. *Treatments*

We expose each participant to programs *with* and *without* variability, while controlling for noise factors such as learning effect, developer competence, and program complexity. First, we establish programs *without* variability (i.e., simple programs) as a baseline. Then, we consider programs *with* variability containing three features (eight configurations).

C. *Participants*

We performed the experiment with N=20 participants: seven undergraduate students, one M.Sc. student, seven Ph.D. students, and five post-docs. All participants had programming experience, especially in Java, and around half of the participants had industrial experience ranging from a few months to several years. All subjects were informed that they were free to stop participating at any time, but no one elected to do so.

D. *Programs*

For the robustness of our experiment—in order to minimize risks of specific effects from a particular program and bug type—we took *two* programs with different kinds of errors. We based our programs on *real* variability errors from two highly-configurable systems: BusyBox and BestLap, taken from previous research [6], [19]. These are qualitatively different systems in terms of size, architecture, purpose, variability, and complexity. BusyBox is an open-source highly-configurable system with 204 KLOC and about 600 features, that provides several essential Unix tools in a single executable file. BestLap is a commercial highly-configurable race game with about 15 KLOC. The kinds of errors we consider are also different: a *null-pointer dereference* and an *assertion violation*. We simplified the error in each system down to an erroneous program that would fit on a screen without scrolling (about 40 lines) yet involve exactly three features.

Bug description of `P`*:* The program has two methods to handle incoming HTTP requests and to send headers (see Fig. 1). As previously explained, the program provokes a null-pointer exception in certain configurations. In debugging this program, the developer needs to identify that the variable `subject` is dereferenced with value null, in exactly three (out of eight) configurations. The method `sendHeaders` is invoked in line 27 (if `AUTH` is *enabled*) or in line 31 (if `CGI` is *enabled*); the variable `subject` will be null whenever the feature `CONFIG_LFS` is *disabled*.

Bug description of `Q`*:* The program originates from a commercial race game. It has one main method responsible for computing a score, as can be seen in the online appendix.[1] The car racing game calculates lap times and assesses qualification for so-called pole position. According to a user requirement, the game should add a penalty when the car crashes. This means that the score can also be negative. However, the method `setScore()` contains a condition prohibiting negative scores. We encode the requirement using assertions. To identify the bug, the developer should somehow see that the variable `totalScore` is always equal to zero after `setScore()` computation, when passing negative values to the method. This error is revealed through an assertion violation in the code (line 31) whenever the features `ARENA` and `NEGATIVE_SCORE` are both enabled, which occurs in exactly two configurations.

Figure 2 lists several characteristics of our benchmark programs. Figure 2a depicts the basic characteristics of the

[1] http://itu.dk/people/jeam/code-gaze-experiment/

Prg	Origin	Filename	Bug type	LOC	#mth
P	BusyBox	`http.c`	*Null-pointer dereference*	39	3
Q	BestLap	`GameScreen.java`	*Assertion violation*	41	4

(a) Basic characteristics.

Prg	#features	Scattering	Tangling	VCC
P	3	4	8	9
Q	3	9	15	14

(b) Variability characteristics.

Fig. 2: Characteristics of our benchmark programs: `P` and `Q`.

	Program 1	Program 2
Developer 1	*without* variability	***with*** **variability**
Developer 2	***with*** **variability**	*without* variability

Fig. 3: Latin Square (2×2).

programs: project, filename, bug type, number of lines of code, and number of methods (including the `main`). Figure 2b lists the variability characteristics of the programs *with* `#ifdefs`, such as: number of features, feature scattering, feature tangling, and variational cyclometric complexity (VCC). Feature scattering is the number of `#ifdef` blocks throughout the entire program. We put accumulated numbers for all features involved. For example, `P` contains four `#ifdef` blocks. Feature tangling, in turn, counts the number of switches between regular code and feature code or between different features. Vcc consists of the *cyclomatic complexity* metric [20] (and counting two flows for `#ifdef` statements).

The programs are similar in terms of size and the number of features. The programs differ in terms of bug type and their variability characteristics, but we control for this in the experiment design, as described below.

E. Design

The two subject programs give rise to four debugging tasks: one for each program *with* and *without* variability. However, in this setup, we need to deal with two main constraints: First, every developer must get each program *once*, otherwise there would be a *learning effect* on subsequent attempts. Second, every developer must get each treatment (with or without variability) *once*, for a similar reason.

Abiding by these constraints, we use a standard *Latin Square design*, which is a common solution for this kind of experiment [21], [22], [23]. A *Latin square* ensures that no row or column contains the same treatment twice. Figure 3 depicts a 2×2 Latin square applied to our context. The columns are labelled with two programs (Program 1 and Program 2). The rows are labelled with two developers (Developer 1 and Developer 2). The four squares in the center contain the two treatments (*with* and *without* variability). Therefore, each developer debugs two *different* programs, each at a *different* variability degree, according to her row.

Finally, we randomly assign participants, treatments, and programs into the Latin squares. The result is the same number of data points for all debugging tasks, without compromising control over the confounding factors such as developer competence or program complexity. For N=20 participants, each performing two out of four debugging tasks, we get exactly 10 data points for each of the four tasks. Technically, our experiment is a *within-group design* in which all participants are exposed to every treatment.

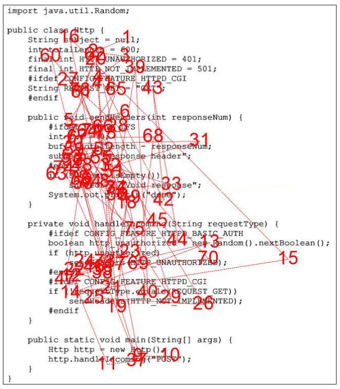
F. Procedure

Before the actual experiment, we executed a *pilot* study with a few local students to test our experiment design and the eye tracking setup. We do not consider the results of the pilot study in our analysis. Based on the pilot study, we optimized mostly the alignment of the programs on the screen for the eye tracker.

The entire experiment consists of five phases: (1) tutorial, (2) warm-up, (3) questionnaire, (4) debugging, and (5) interview. First, when a subject enters the room, we present a tutorial on variability explaining the concepts of features, configurations, and variability. Second, we demonstrate the nature of the tasks and questions through a small warm-up task. Third, we ask the participant to fill in a self-assessment questionnaire about her programming background and experience with Java and `#ifdefs`. Fourth, we run the actual debugging experiment using an eye tracker. We use the randomly generated Latin squares to create a *task description sheet* detailing the order of the tasks for a participant. We also run the personal calibration procedure right before the tasks. Then, the participant performs the “find the bug” debugging task for each treatment, in order. Fifth, once a developer finishes the tasks, we conduct a semi-structured interview to get qualitative feedback on how the participant approached the debugging tasks, especially the program with variability. We ask three questions: (i) How did you go about finding the bug? (ii) What were the difficulties? and (iii) How could you fix the bug?

All task description sheets contain instructions and questions that every participant should answer (see the online appendix for an example). We ensure that each program fits onto a single screen to avoid participants scrolling up and down, which would significantly complicate getting the eye tracking data. In other words, we provided the participants with only a static screen (i.e., no IDEs, no tools, no navigation) and the task description sheets on paper. For each participant, we recorded timestamps, the duration of each debugging task, as well as x and y coordinates (fixations) via the eye tracker.

To avoid unintended effects from different software and hardware environments, we executed all experiments on a 64-bit



```
import java.util.Random;

public class Http {
    String subject = null;
    int totalLength = 600;
    final int HTTP_UNAUTHORIZED = 401;
    final int HTTP_NOT_IMPLEMENTED = 501;
    #ifdef CONFIG_FEATURE_HTTPD_CGI
    String REQUEST_GET = "GET";
    #endif

    public void sendHeaders(int responseNum) {
        #ifdef CONFIG_LFS
        int buf = 0;
        buf = totalLength - responseNum;
        subject = "response header";
        #endif
        if (subject.isEmpty())
            subject = "Void response";
        System.out.println("done");
    }

    private void handleIncoming(String requestType) {
        #ifdef CONFIG_FEATURE_HTTPD_BASIC_AUTH
        boolean http_unauthorized = new Random().nextBoolean();
        if (http_unauthorized)
            sendHeaders(HTTP_UNAUTHORIZED);
        #endif
        #ifdef CONFIG_FEATURE_HTTPD_CGI
        if (!requestType.equals(REQUEST_GET))
            sendHeaders(HTTP_NOT_IMPLEMENTED);
        #endif
    }

    public static void main(String[] args) {
        Http http = new Http();
        http.handleIncoming("POST");
    }
}
```

Fig. 4: Fixation sequence: sequence of all fixations from one developer as a "connect-the-dots" visualization.

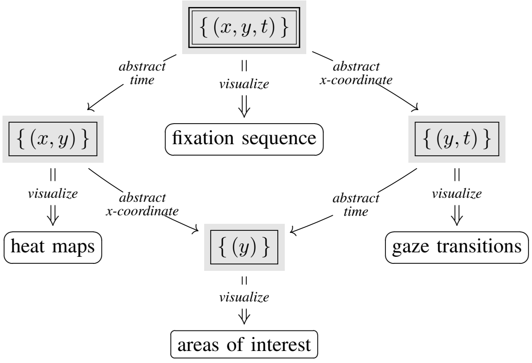


Fig. 5: Overview of abstractions for data analysis.

Windows®8.1 machine, Intel®Core^{TM} i5 CPU running at 2.5 GHz with 8 GB memory. Also, all experiments were conducted in the same room (quiet location). We recorded all of the eye tracking data using the open-source tool OGAMA.[2] In the experiment, we used the Tobii EyeX Controller integrated into the EyeInfo Framework.[3] Together they had a radial accuracy error of < 0.4 degrees which translates to a mere 1.5 lines of inaccuracy on the screen. We discuss implications of this inaccuracy in Sect. V.

We eliminated the data from two subjects because of the eye tracker malfunction occurring during the experiment. The N=20 subjects include only the valid data points. We discuss this issue in Section V. No other deviations happened.

IV. Results & Discussion

We now present the results of the experiment and discuss the implications. We make five observations on how developers debug programs with variability. We begin with presenting our framework of abstractions to simplify data analysis. Then, we address the research question and discuss the findings. All materials of the experiment are available at:

http://itu.dk/people/jeam/code-gaze-experiment/

A. Abstractions for data analysis

Figure 4 displays a scan path registered for a single developer as a sequence of all gaze fixations during a debugging task. For each participant, the eye tracker records a set of triples: x and y coordinates over time t. The raw data enables us to draw the fixation sequence of Fig. 4, a "connect-the-dots" visualization, not particularly helpful for a big set of data of more than one subject. Since the diagram is difficult to understand, we use a range of abstractions (Fig. 5) to ease the data interpretation. We use additional three abstractions besides the fixation sequence, obtained by simplifying away selected dimensions: *heat maps*, *gaze transitions*, and *areas of interest* (AOI).

Heat Map: First, we marginalize the data over time which gives us a multi-set of timeless (x, y) pairs. We convert this to a histogram, telling us for each screen location, how much fixation time it attracted. The two-dimensional histogram can be visualized as a so-called *heat map*–using colors to represent the value of time for each location on the screen. An example is shown in Fig. 6. The lowest value in the heat map (lowest fixation time) is shown using the purple color and the highest value is red (long fixation time), with a smooth transition between these extremes.

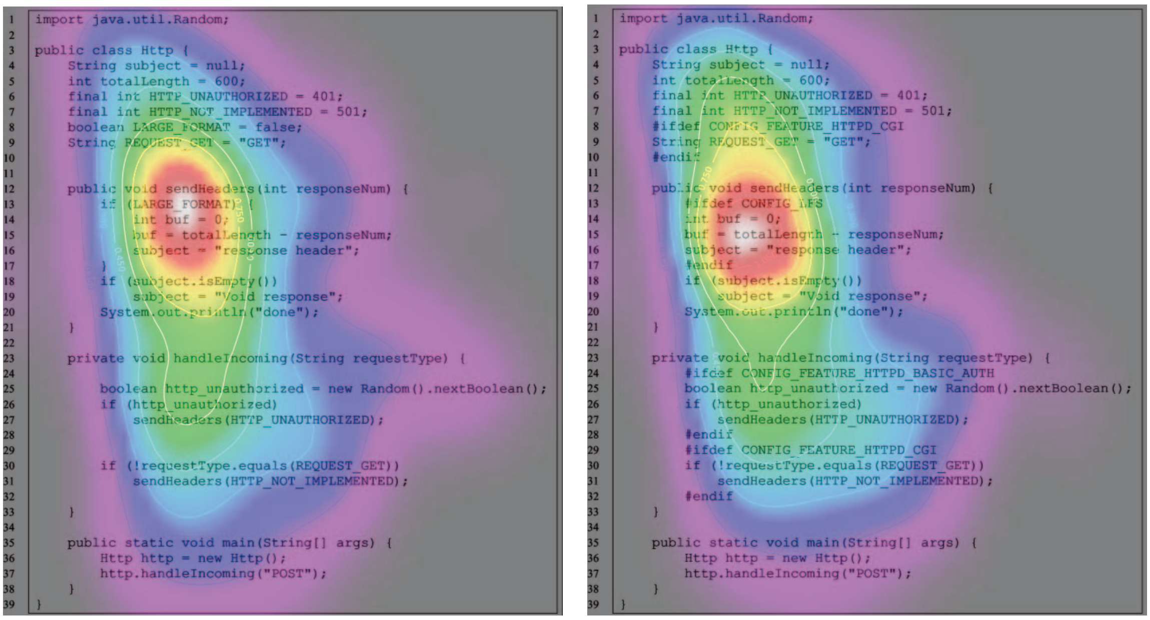
Gaze Transition (a.k.a. Saccade): We can abstract the x-coordinate away, obtaining a set of (*y, t*) pairs, from which, we are able to generate a timeline of how the participants read programs vertically—the gaze transitions graphs. In our gaze transition graph (cf. Fig. 11) the y-coordinate is translated to line numbers in code and plotted as a function of time in seconds. The gaze transition graph shows where exactly in the program the participant's gaze is at in the corresponding time.

Areas of Interest: The last abstraction is a combination of the previous two—we abstract away both the horizontal coordinate and time. We end up with a histogram over y-coordinates (see Fig. 8 for an example). This abstraction is primarily useful for displaying the number of fixations related to given *areas of interest* (AOI). An AOI is a region of interest in a study. In our study, we define the areas of interest with the AOI editor in OGAMA following the guidelines provided by Holmqvist et al. [24]. Consequently, once we have the set of *y*-coordinates and the AOIs defined, we are able to visualize the percentage/amount of fixations from each participant via a table or a bar chart.

An observant reader will notice that we ignored another natural abstraction—marginalizing over the *y*-coordinate. Abstracting the *y*-coordinate is not relevant for this study, as we are not interested in how subjects approach lines horizontally (how they read within a line).

We now return to discussion our research question.

[2] http://www.ogama.net/
[3] http://eyeinfo.itu.dk/



```
1  import java.util.Random;
2
3  public class Http {
4      String subject = null;
5      int totalLength = 600;
6      final int HTTP_UNAUTHORIZED = 401;
7      final int HTTP_NOT_IMPLEMENTED = 501;
8      boolean LARGE_FORMAT = false;
9      String REQUEST_GET = "GET";
10
11
12     public void sendHeaders(int responseNum) {
13         if (LARGE_FORMAT) {
14             int buf = 0;
15             buf = totalLength - responseNum;
16             subject = "response header";
17         }
18         if (subject.isEmpty())
19             subject = "Void response";
20         System.out.println("done");
21     }
22
23     private void handleIncoming(String requestType) {
24
25         boolean http_unauthorized = new Random().nextBoolean();
26         if (http_unauthorized)
27             sendHeaders(HTTP_UNAUTHORIZED);
28
29
30         if (!requestType.equals(REQUEST_GET))
31             sendHeaders(HTTP_NOT_IMPLEMENTED);
32
33     }
34
35     public static void main(String[] args) {
36         Http http = new Http();
37         http.handleIncoming("POST");
38     }
39 }
```

(a) *Without* variability.

```
1  import java.util.Random;
2
3  public class Http {
4      String subject = null;
5      int totalLength = 600;
6      final int HTTP_UNAUTHORIZED = 401;
7      final int HTTP_NOT_IMPLEMENTED = 501;
8      #ifdef CONFIG_FEATURE_HTTPD_CGI
9      String REQUEST_GET = "GET";
10     #endif
11
12     public void sendHeaders(int responseNum) {
13         #ifdef CONFIG_LFS
14         int buf = 0;
15         buf = totalLength - responseNum;
16         subject = "response header";
17         #endif
18         if (subject.isEmpty())
19             subject = "Void response";
20         System.out.println("done");
21     }
22
23     private void handleIncoming(String requestType) {
24         #ifdef CONFIG_FEATURE_HTTPD_BASIC_AUTH
25         boolean http_unauthorized = new Random().nextBoolean();
26         if (http_unauthorized)
27             sendHeaders(HTTP_UNAUTHORIZED);
28         #endif
29         #ifdef CONFIG_FEATURE_HTTPD_CGI
30         if (!requestType.equals(REQUEST_GET))
31             sendHeaders(HTTP_NOT_IMPLEMENTED);
32         #endif
33     }
34
35     public static void main(String[] args) {
36         Http http = new Http();
37         http.handleIncoming("POST");
38     }
39 }
```

(b) *With* variability.

Fig. 6: Aggregated heat maps for the program P.

	Program P	Program Q
without variability	$5\frac{3}{4}$ min	5 min
with **variability**	$10\frac{1}{2}$ min	$10\frac{1}{2}$ min

Fig. 7: Average total debugging times in our experiment.

B. How do developers debug programs with variability?

Previous work has demonstrated that debugging time increases with variability; in fact, the increase appears to be linear in the number of features [5]. Figure 7 shows the average total debugging time for each of the two programs P and Q, *without* variability (zero features) vs. *with* variability (three features). For both programs, the average total debugging time goes up from roughly *five* to *ten* minutes when the programs involve variability; i.e., the debugging time is *doubled*.

Using the eye tracking data we can investigate deeper *where* developers are spending all this extra debugging time. Based on our eye-tracking experiment, we made five observations:

> OBSERVATION 1: *Variability appears to increase debugging time of the areas of the program that contain variability.*

Figure 6 shows the aggregated heat maps for the program P *without* variability (to the left) versus *with* variability (to the right). Aggregated heat maps are produced by first normalizing (with respect to time) and then superimposing all individual heat maps such that contributions from each developer will be accounted for equally. (Since we have N=20 participants, each aggregated heat map is derived from ten individual heat maps.) Aggregated heat maps give an overall picture of the focus of the developers; i.e., how much they were looking at each part of the program, on average. Importantly, in contrast to Figure 7 that considers *absolute* time, Figure 6 considers *relative* time: how attention is distributed among the program parts.

The *hot spots* (red regions) indicate areas where most of the attention was directed. Not surprisingly, most attention was awarded to the method containing the bug, `sendHeaders` (specifically, lines 12 to 18). Recall that the bug was in line 18 where the condition `subject.isEmpty()` produces a null-pointer exception since the variable `subject` has the value `null`. (In the case *with* variability, this happens in certain configurations.[4]) Overall, the red regions appear quite similar. Without variability, developers dedicate 12% of all fixations to this area (752 out of 6,355). With variability, the dedication to this area is comparable in relative terms with 15% fixations (although using more fixations in absolute terms: 1,249 out of 8,339). The Kullback-Leibler Divergence test confirms that the similarity between the two hot spots is highly significant (divergence value = 0.05, in a scale [0,1]). We observe the same phenomenon for the hot spots in the other program Q (divergence value = 0.07).

Figure 8 details the total time spent looking at each of the four designated *areas of interest* of the program: the field

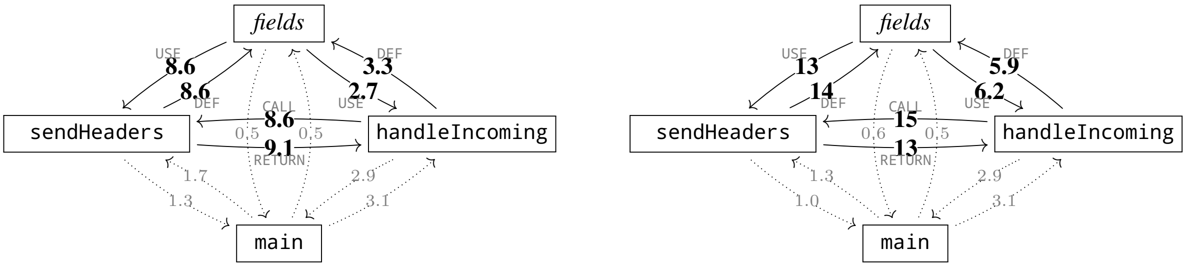
[4] The bug occurs when LFS is *disabled* and either AUTH or CGI is *enabled*; i.e., $\neg$LFS $\wedge$ (AUTH $\vee$ CGI).

area of interest		variability		increase
lines	area	*without*	*with*	factor
`4-9`	*fields*	26 s	58 s	2.2 x
`12-21`	`sendHeaders`	63 s	120 s	1.9 x
`23-33`	`handleIncoming`	56 s	98 s	1.8 x
`35-38`	`main`	8.2 s	5.3 s	0.7 x
Σ	*all four areas*	153 s	281 s	1.8 x

Fig. 8: Average debugging time for four *areas of interest* of the program `P` *without* vs. *with* variability.

sub-area of interest		variability		increase
lines	sub-area	*without*	*with*	factor
`P:12-17`	- *with* variability	38 s	77 s	2.0 x
`P:18-21`	- ***without*** **variability**	**25 s**	**43 s**	**1.7 x**
Σ	`sendHeaders`	63 s	120 s	1.9 x
`Q:18-20`	- ***without*** **variability**	**24 s**	**45 s**	**1.9 x**
`Q:21-33`	- *with* variability	48 s	130 s	2.7 x
Σ	`gc_computeLevelScore`	72 s	175 s	2.4 x

Fig. 9: Average debugging time for fragments without variability in proximity of fragments with variability.



(a) *Without* variability.

(b) *With* variability.

Fig. 10: Average number of gaze transitions (eye switches) between the differents elements of program `P`.

declarations (lines 4–9); the method `sendHeaders` (lines 12–21); `handleIncoming` (lines 23–33); and `main` (lines 35–38). For instance, the attention devoted to the method `sendHeaders` goes up from about a minute (63 seconds) to two minutes (120 seconds) in the presence of variability; i.e., an increase factor of 1.9 (almost twice as much attention). Overall, it appears that the extra (roughly double) debugging time is spent on all areas of the program that involve variability: the field declarations and the two methods `sendHeaders` and `handleIncoming` all double debugging time. In contrast, no extra time is spent on `main` that does not involve variability. In fact, attention to this area appears to drop slightly in the presence of variability.

Please note that the attention awarded to the four *areas of interest* (last line in Figure 8) does not add up to the total debugging time of Figure 7. This is because the four elements do not cover everything (e.g., imports, blank lines, class definitions, and even areas beyond the screen), gaze transitions (rapid eye movements) are not accounted for in Figure 8, and the total debugging time also involves answering questions about the bugs on a sheet of paper (i.e., not looking at the screen).

> <u>Observation 2</u>: *Debugging time also increases for code fragments without variability in proximity of code fragments that do contain variability.*

Consider the body of the `sendHeaders` method in program `P` with variability (cf. Fig. 6b). We see that it consists of a code fragment *with* variability (lines 13–17) followed by a fragment *without* variability (lines 18–20). A similar phenomenon occurs in program `Q` in the function `gc_computeLevelScore`, where the top part (lines 18–20) does not contain variability followed by a fragment (lines 21–33) with variability.

Designating these as our *sub-areas of interest*, we can thus zoom in and study the impact of code fragments *with* variability on code fragments *without* variability within the same method.

Figure 9 splits these two methods into their sub-areas of interest with vs. without variability. The sub-areas without variability "in proximity" of variability are shown in bold face. Variability appears to be "contagious" along the flow of control, within a method. Even though lines (18–21) in `P` do not have variability, they go from 25 seconds to 43 seconds to debug in the presence of variability (i.e., debugging takes 1.7 times longer). Similarly, for lines 18–20 in `Q`; they go from 24 seconds to 45 seconds (i.e., debugging takes 1.9 times longer).

We hypothesize that this is because the developers are considering different configurations while debugging (more on this in Observation 5 later).

> <u>Observation 3</u>: *Variability appears to increase the number of gaze transitions between definition-usages for fields and call-returns for methods.*

Figure 10 depicts the average number of *gaze transitions* between the four previously introduced areas of interest. Without variability there are, for instance, on average 8.6 navigations from `handleIncoming` to `sendHeaders` and 9.1 back again (see Fig. 10a). Navigations between two methods are annotated with *call* and *return* according to invocations in the program (e.g., `sendHeaders` is called from `handleIncoming` in line 27 and 31). The gaze transition diagrams confirm that the eye movements proceed along method invocations. Similarly,

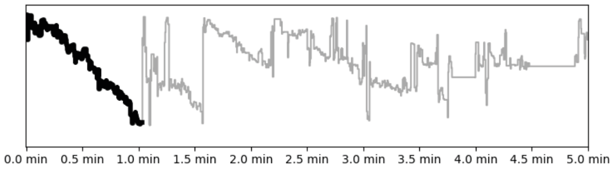


Fig. 11: Gaze transition diagram with *initial scan*.

method-to-field navigations are annotated with *def* and *use* as developers navigate from a field variable usage to its *definition* and back again to the *use*. For instance, we see on average 8.6 navigations from `sendHeaders` to the *fields* area of interest (def) and exactly the same number going back again to the usage within the method (use).

With variability, all gaze transitions out of methods containing variability increase significantly (cf. Figure 10b compared to Figure 10a). The *method-to-method* navigation along call-return from `handleIncoming` to `sendHeaders` goes up to 15 and 13 (from 8.6 and 9.1). For *method-to-field*, the (def-use) navigations out of `sendHeaders`, for instance, goes up to 13 and 14 (from 8.6 and 8.6). For navigations out of the method `main` that does not contain any variability (shown as dotted gray edges), we see little change.

Thus, the participants made significantly more gaze transitions in the presence of variability. Again, we hypothesize that developers are exploring and re-exploring different configurations while debugging (cf. OBSERVATION 5).

> OBSERVATION 4: *Variability appears to prolong the "initial scan" of the program (first line to last line) that most developers initiate debugging with.*

Previous work has reported that when performing static code review of program (without variability), reviewers initially perform a preliminary reading of the code, known as a *scan*, whereby a reviewer will *"read the entire code before investigating the details [..]"* [25]. Similarly, when debugging programs (without variability), developers perform a *"first scan followed by several rounds of navigation"* [26].

Not surprisingly, for debugging programs *with* variability, we also see this *initial scan*. Figure 11 illustrates a *gaze transition* diagram of the first five minutes of a participant debugging `P` *with* variability. The diagram shows the y-coordinate the developer is looking at as a function of time. The top of the diagram corresponds to the first line (line 1) and the bottom to the last line of the program (line 39). We have highlighted the initial scan which, in this case, lasted for one minute.

This is supported by the post-experiment interviews: *"I took a global look first from top to bottom and then I started from the main"* (one developer); *"First, I started reading from the top and double checking the fields and methods"* (another one).

Without variability, 7 out of 10 developers performed an initial scan within half a minute (32 seconds) on average. With variability, 8 out of 10 developers scanned initially and it took an average of 51 seconds. Obviously, scanning a larger program will take longer time. Because of the conditional compilation directives (i.e., the `#ifdef` and `#endif` directives), the programs with variability are slightly larger. In fact, they contain 14% more characters. However, this does not account for the 65% increase in debugging time (from 32 to 51 seconds).

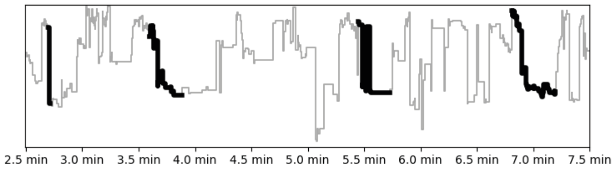


Fig. 12: Gaze transition diagram for a developer using a consecutive strategy and repeatedly considering a method (highlighted in black).

> OBSERVATION 5: *Developers appear to debug programs with variability by considering either one configuration at a time (consecutively) or all configurations at the same time (simultaneously).*

The interviews give some qualitative insights into how the subjects debug programs with variability. Most participants complained that they had trouble finding the bug in the presence of variability. One subject explains that he is using a *consecutive strategy* by considering one configuration at a time: *"I began with all features enabled, then I removed one-by-one."* Along the same lines, another explains: *"After I get a good understanding of the code, I started to enable/disable features one at a time to see if the bug appears."* This approach manifests itself on his gaze transition diagram which contains repetitions corresponding to the method `sendHeaders` with variability (cf. Figure 12).

Another subject claims to adopt a *simultaneous strategy* by considering all configurations at the same time: *"I tried to keep track of everything by compiling every combination in mind."* The two strategies are also well-known in *automated* program analysis of programs with variability [27].

Independent of strategies, all developers agreed that debugging programs *without* variability required much less effort. This finding aligns with the study of Medeiros et al. [28] in which they observed that bugs involving variability are easier to introduce and harder to debug and fix than ordinary bugs.

C. Discussion: Implications of our Results

Our results *confirm* previous hypotheses [5], regarding the *accuracy* of debugging programs with variability:

> CONFIRMATION: *Most developers correctly identify bugs in programs with variability; however, many developers fail to identify exactly the set of erroneous configurations (already for 3 features).*

This is also consistent with previous research reporting that developers admit that when fixing programs with variability, they "check only a few configurations of the source code" [28].

Observation 3 (above) stated that developers perform more navigation in the presence of variability. Knowing that, we encourage the programmers using variability to structure the code in a way that minimize the distance between uses and definitions of field variable declarations or between methods calling each other, especially for those declarations and uses that involve `#ifdefs`. At the same time, the builders of development environments shall consider providing convenient ways to navigate from uses to definitions and back again and along call-returns for method invocations. An IDE equipped with continual eye tracking could even automatically "pop up" relevant definitions next to uses as they are being considered by the developer. Clearly, as shown in our data, these pop-ups might be more useful, in areas of code that involve variability (so intensive variability could activate them).

Observations 1–3 indicate that it is worth to contain variability in as few methods as possible to keep other methods variability free. Observation 2 hints that it is advantageous to hoist code fragments without variability "in proximity" of variability out of the method. For instance, in program `P` with variability, lines 18–20 could be moved into a fresh method.

All observations 1–5 may indicate that there are potential gains from *projectional editing* of program with variability. Developers could work separately on particular configurations (programs without variability) which would then be automatically synchonized with the entire variability program (spanning all configurations) [29]. This could be activated/suggested automatically for programmers who work following the consecutive (brute-force) process, as this process can be presumably detected automatically as multiple scans in the eye-tracking data. Of course, we do not know to what extent, or whether at all, these suggestions improve debugging programs with variability. However, our findings do provide indications, that these are the directions that might be worth exploring.

V. Threats to Validity

A. Internal Validity

Selection bias: To minimize selection bias, we randomly assigned our subjects into the Latin squares. Additionally, every participant took all treatments (with and without variability) for all subject programs. Therefore, we controlled the confounding factors via Latin square design and randomization.

All participants voluntarily accepted to take part in the experiment. We found no indications of participants performing deliberately bad or exchanging information. However, we eliminated the data points from two participants because the eye tracker stopped tracking their eye movements during the experiment. For this reason, we did not include them in our data analysis. Thus, the N=20 participants represent only valid data points.

Choice of lab setting: We executed the experiment N=20 times (i.e., one for each participant). All experiments were done in the same room and with our supervision, avoiding extra confounding factors. Since this is the first study of variability debugging using eye-tracking, we opted for a controlled experiment to understand how developers debug programs with variability in lab conditions. We thus optimized for internal validity rather than external validity and a real development environment.

Choice of eye tracker: We used the Tobii EyeX Controller integrated into the EyeInfo Framework for two reasons. First, they are portable and easy to set up and install. Second and, more importantly, they have a good accuracy. In fact, they had a radial accuracy error of < 0.4 degrees only. This is excellent since an accurate and reliable calibration is crucial for eye-tracking studies and conclusions. This translates to an inaccuracy of approximately 1.5 lines on the screen. For this reason, we never considered areas of interest with less than three lines of code.

Choice of language: In this experiment, we used the Java programming language because we wanted to run the experiment with several participants and Java is well-known among students at our universities. All participants had experience with Java, ranging from months to years, including professionally. We chose bugs that are relatively independent of programming language, i.e., they occur in programs written in subset of C that is essentially shared with Java.

Choice of the number of features: We studied variability up to *three* features in a program mainly because of timing. Otherwise, the experiment would require much longer time, discouraging and tiring participants. In fact, the participants spent around 15 minutes to debug only the two programs (with and without variability), on average. Additionally, there are very few examples of bugs with higher number of features than three in the literature [6], [30]. Thus, our study focused on the range of variability that seems most relevant.

Program vs. variability complexity: Note that from this experiment and its results, we do not know the *origins* of the extra debugging effort entailed by variability, since we did not focus on the difference between complex programs vs variability programs. This would require another experiment setup and, therefore, it is out of the scope of our study.

B. External Validity

Beyond preprocessors: Our experiment applies to a particular technique for implementing variability: preprocessor (`#ifdefs`). However, among a multitude of technologies that can be used to implement configurable systems, the C preprocessor is one of the oldest, simplest, and most popular mechanisms in use. Generalization to other variability techniques is not intended, even though it might provide hints. Presumably, our results do not translate to CIDE [16], which uses colors to visualize `#ifdef` blocks, since the human visual system is highly sensitive to colors [31].

Beyond university students: The experiment were done predominantly with graduate students. All had Java programming experience and several of them had industrial experience. In addition, research has demonstrated that graduate students make good proxies for industry developers [32]. This contributes to representativity and generalization to "real-world" industrial developers. We acknowledge though that more studies are needed to further understand and confirm the presented observations.

Beyond simple programs: The programs used in our experiment are based on real variability bugs from real highly-configurable systems (BusyBox and BestLap) and previous research [6], [19], which minimize the risks of studying artificial problems. Additionally, the programs were qualitatively different (cf. Figure 2), and comprising of different kinds of bugs. We thus believe that the results may transfer to other smaller programs.

Beyond simple debugging tasks: We purposely designed our debugging tasks to not require a long time and, consequently, discouraging participants. In fact, the participants spent around 15 minutes to debug the two programs, on average. So, there may be additional effects, unaccounted for, when scaling to longer debugging tasks.

Beyond lab settings: We made sure that each program fits onto a single screen to avoid participants scrolling up and down. We prepared all debugging tasks in slideshow manner using Ogama, a framework to analyze eye movements. We also did not bold or highlight any code constructs in the programs in order not to attract special attention and, consequently, favor any particular code elements. In other words, no IDEs, no tools, no navigation were provided to the participants; they only had a static screen with the programs and task sheets (on paper). Anything beyond that it is out of the scope for this paper.

Beyond three features: There are very few examples of bugs with higher number of variability than 3 in the literature. Medeiros et al. [30] found that 95% of undeclared/unused bugs involve 0-3 features. Other research also found that variability bugs predominantly involve 0-3 features [6]. For this reason, we focused on this range that seems most relevant. However, it would be interesting to investigate programs with higher number of features.

VI. Related Work

In a previous study, we have investigated the impact of variability on bug finding in terms of time and accuracy [5]. However, our previous study focused only on *quantitative* aspects of debugging, and not on *how* developers debug programs with variability. Thus, to better account for the effect of variability on debugging, we carried out this eye-tracking experiment to actually "see" how developers approach programs with variability, as shown in Section IV. To the best of our knowledge, this is the first study of variability debugging using eye tracking. A few other studies have used eye tracking to study debugging and program comprehension in ordinary programs (i.e., without variability).

Hansen et al. [33] used eye-tracking to investigate factors that impact code comprehension. They found that even subtle notation changes can have impact on performance, and that notation can also make a simple program more difficult to read.

Busjahn et al. [34] conducted eye-tracking studies on small programs to investigate how programmers read code. They found that the fixation durations increased when reading source code in comparison with natural language text. Busjahn et al. [35] also studied linearity and whether or not the linearity effect in reading natural languages transfers to reading of source code. They observed that expert programmers read code less linearly than novices which, in turn, read code less linearly than natural language text.

Rodeghero et al. [36] conducted an eye-tracking study of ten Java programmers. They noticed that the programmers looked more at a method's signature than its body in order to summarize it in plain English.

Siegmund et al. [37] conducted a controlled experiment with 17 programmers by applying *functional magnetic resonance imaging* (fMRI) to measure program comprehension. They found a distinct pattern active in five brain regions that are thus deemed necessary for source code comprehension.

None of the these studies investigated debugging in the presence of variability using an eye tracker. In other words, variability was not in their focus. We, in turn, focused on the interplay between debugging and variability from the programmers' perspective. We could draw a number of qualitative conclusions (cf. Section IV). However, we believe that further research using eye tracking on variability debugging is important and required to confront our findings, and to draw new ones.

VII. Conclusion

We have presented an experiment aimed at understanding *how* developers debug code with variability implemented using preprocessor directives. We observed that variability increases debugging time for code fragments that contain variability and for neighboring locations. Also, it appears that developers navigate much more between definitions and uses of program objects when interleaved with variability. This is presumably caused by increased complexity of def-use relationships, or by difficulties of maintaining all variants in short-term memory. Variability prolongs the "initial scan" of the program that most subjects initiate debugging with. We notice that developers appear to debug programs with variability by using either a consecutive or simultaneous approach.

Our results are consistent with those of prior studies to the extent that they overlap. The new findings provide some indications how code should be organized to minimize the number of gaze transitions, and on what kind of tools could aid debugging. Automatic tools showing definitions at usage locations, could consider intensive variability as an indicator that the definition is a more sought for information at a given context. Also, possibly, projectional editing techniques can be used to reduce the cognitive overload of variability, especially for subjects using the consecutive approach.

Acknowledgements. We thank Thao for carrying out a pilot study in connection with her Master's Thesis. This work is supported by Brazilian Science without Borders Programme, CNPq grants no. 249020/2013-0 and no. 229760/2013-9. Wasowski is partially funded by The Danish Council for Independent Research, Sapere Aude grant no. 0602-02327B, VARIETE project.

References

[1] P. Clements and L. Northrop, *Software Product Lines: Practices and Patterns*. Addison-Wesley, 2002.

[2] K. Pohl, G. Bockle, and F. J. van der Linden, *Software Product Line Engineering*. Springer, 2005.

[3] T. Berger, D. Nair, R. Rublack, J. M. Atlee, K. Czarnecki, and A. Wasowski, "Three cases of feature-based variability modeling in industry," in *ACM/IEEE 17th International Conference on Model Driven Engineering Languages and Systems (MODELS)*, 2014. [Online]. Available: http://gsd.uwaterloo.ca/sites/default/files/2014-models-vmstudy.pdf

[4] T. Berger, S. She, R. Lotufo, A. Wasowski, and K. Czarnecki, "A study of variability models and languages in the systems software domain," *Software Engineering, IEEE Transactions on*, vol. 39, no. 12, pp. 1611–1640, Dec 2013.

[5] J. Melo, C. Brabrand, and A. Wasowski, "How does the degree of variability affect bug finding?" in *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*, 2016, pp. 679–690. [Online]. Available: http://doi.acm.org/10.1145/2884781.2884831

[6] I. Abal, C. Brabrand, and A. Wasowski, "42 Variability Bugs in the Linux Kernel: A Qualitative Analysis," in *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ser. ASE '14. New York, NY, USA: ACM, 2014, pp. 421–432. [Online]. Available: http://doi.acm.org/10.1145/2642937.2642990

[7] J. Melo, E. Flesborg, C. Brabrand, and A. Wasowski, "A quantitative analysis of variability warnings in linux," in *Proceedings of the Tenth International Workshop on Variability Modelling of Software-intensive Systems*, ser. VaMoS '16. New York, NY, USA: ACM, 2016, pp. 3–8. [Online]. Available: http://doi.acm.org/10.1145/2866614.2866615

[8] S. Schulze, J. Liebig, J. Siegmund, and S. Apel, "Does the discipline of preprocessor annotations matter?: A controlled experiment," in *Proceedings of the 12th International Conference on Generative Programming: Concepts & Experiences*, ser. GPCE '13. New York, NY, USA: ACM, 2013, pp. 65–74. [Online]. Available: http://doi.acm.org/10.1145/2517208.2517215

[9] M. Land and B. Tatler, "Looking and acting: eye movements in everyday life," 2009.

[10] D. W. Hansen and Q. Ji, "In the eye of the beholder: A survey of models for eyes and gaze," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 32, no. 3, pp. 478–500, March 2010.

[11] M. E. Crosby and J. Stelovsky, "How do we read algorithms? a case study," *Computer*, vol. 23, no. 1, pp. 25–35, 1990.

[12] Y.-G. Guéhéneuc and P. Team, "A theory of program comprehension," 2005.

[13] R. Bednarik and J. Randolph, "Studying cognitive processes in computer program comprehension," in *Passive Eye Monitoring*. Springer, 2008, pp. 373–386.

[14] C. Parnin, "A cognitive neuroscience perspective on memory for programming tasks," *Programming Interest Group*, p. 27, 2010.

[15] M. E. Hansen, A. Lumsdaine, and R. L. Goldstone, "Cognitive architectures: A way forward for the psychology of programming," in *Proceedings of the ACM international symposium on New ideas, new paradigms, and reflections on programming and software*. ACM, 2012, pp. 27–38.

[16] C. Kästner, S. Apel, and M. Kuhlemann, "Granularity in Software Product Lines," in *Proceedings of the 30th International Conference on Software Engineering (ICSE)*. ACM, 2008, pp. 311–320.

[17] J. Liebig, S. Apel, C. Lengauer, C. Kästner, and M. Schulze, "An analysis of the variability in forty preprocessor-based software product lines," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering (ICSE)*. ACM, 2010, pp. 105–114.

[18] P. W. Oman, C. R. Cook, and M. Nanja, "Effects of programming experience in debugging semantic errors," *J. Syst. Softw.*, vol. 9, no. 3, pp. 197–207, Mar. 1989. [Online]. Available: http://dx.doi.org/10.1016/0164-1212(89)90040-X

[19] M. Ribeiro, P. Borba, and C. Kästner, "Feature maintenance with emergent interfaces," in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE 2014. New York, NY, USA: ACM, 2014, pp. 989–1000. [Online]. Available: http://doi.acm.org/10.1145/2568225.2568289

[20] T. J. McCabe, "A complexity measure," *IEEE Trans. Softw. Eng.*, vol. 2, no. 4, pp. 308–320, Jul. 1976. [Online]. Available: http://dx.doi.org/10.1109/TSE.1976.233837

[21] R. A. Bailey, *Design of comparative experiment*. Cambridge University Press, 2008.

[22] G. E. P. Box, J. S. Hunter, and W. G. Hunter, *Statistics for Experimenters: design, innovation, and discovery*. Wiley-Interscience, 2005.

[23] D. C. Montgomery, *Design and Analysis of Experiments*. John Wiley & Sons, 2006.

[24] K. Holmqvist, M. Nystrom, R. Andersson, R. Dewhurst, H. Jarodzka, and J. van de Weijer, *Eye Tracking. A comprehensive guide to methods and measures*. Oxford University Press, 2011. [Online]. Available: http://www.oup.com/us/catalog/general/subject/Psychology/CognitivePsychology/CognitivePsychology/?view=usa&#38;ci=9780199697083

[25] H. Uwano, M. Nakamura, A. Monden, and K. ichi Matsumoto, "Analyzing individual performance of source code review using reviewers' eye movement," in *in Proceedings of 2006 symposium on Eye tracking research & applications (ETRA)*, 2006, pp. 133–140.

[26] X. Xie, Z. Liu, S. Song, Z. Chen, J. Xuan, and B. Xu, "Revisit of automatic debugging via human focus-tracking analysis," in *Proceedings of the 38th International Conference on Software Engineering*, ser. ICSE '16. New York, NY, USA: ACM, 2016, pp. 808–819. [Online]. Available: http://doi.acm.org/10.1145/2884781.2884834

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

[28] F. Medeiros, C. Kästner, M. Ribeiro, S. Nadi, and R. Gheyi, "The love/hate relationship with the C preprocessor: An interview study," in *Proceedings of the 29th European Conference on Object-Oriented Programming (ECOOP)*, ser. Lecture Notes in Computer Science. Berlin/Heidelberg: Springer-Verlag, 2015.

[29] E. Walkingshaw and K. Ostermann, "Projectional editing of variational software," in *Proceedings of the 2014 International Conference on Generative Programming: Concepts and Experiences*, ser. GPCE 2014. New York, NY, USA: ACM, 2014, pp. 29–38. [Online]. Available: http://doi.acm.org/10.1145/2658761.2658766

[30] F. Medeiros, I. Rodrigues, M. Ribeiro, L. Teixeira, and R. Gheyi, "An empirical study on configuration-related issues: Investigating undeclared and unused identifiers," in *Proceedings of the 2015 ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*, ser. GPCE 2015. New York, NY, USA: ACM, 2015, pp. 35–44. [Online]. Available: http://doi.acm.org/10.1145/2814204.2814206

[31] D. Moody, "The "physics" of notations: Toward a scientific basis for constructing visual notations in software engineering," *IEEE Trans. Softw. Eng.*, vol. 35, no. 6, pp. 756–779, Nov. 2009. [Online]. Available: http://dx.doi.org/10.1109/TSE.2009.67

[32] R. P. Buse, C. Sadowski, and W. Weimer, "Benefits and barriers of user evaluation in software engineering research," *ACM SIGPLAN Notices*, vol. 46, no. 10, pp. 643–656, October 2011.

[33] M. Hansen, R. L. Goldstone, and A. Lumsdaine, "What makes code hard to understand?" *arXiv preprint arXiv:1304.5257*, 2013.

[34] T. Busjahn, C. Schulte, and A. Busjahn, "Analysis of code reading to gain more insight in program comprehension," in *Proceedings of the 11th Koli Calling International Conference on Computing Education Research*, ser. Koli Calling '11. New York, NY, USA: ACM, 2011, pp. 1–9. [Online]. Available: http://doi.acm.org/10.1145/2094131.2094133

[35] T. Busjahn, R. Bednarik, A. Begel, M. Crosby, J. H. Paterson, C. Schulte, B. Sharif, and S. Tamm, "Eye movements in code reading: Relaxing the linear order," in *2015 IEEE 23rd International Conference on Program Comprehension*, May 2015, pp. 255–265.

[36] P. Rodeghero, C. McMillan, P. W. McBurney, N. Bosch, and S. D'Mello, "Improving automated source code summarization via an eye-tracking study of programmers," in *Proceedings of the 36th International Conference on Software Engineering*. ACM, 2014, pp. 390–401.

[37] J. Siegmund, C. Kästner, S. Apel, C. Parnin, A. Bethmann, T. Leich, G. Saake, and A. Brechmann, "Understanding understanding source code with functional magnetic resonance imaging," in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE 2014. New York, NY, USA: ACM, 2014, pp. 378–389. [Online]. Available: http://doi.acm.org/10.1145/2568225.2568252

Variability Bugs in Highly-Configurable Systems: A Qualitative Analysis (Paper 2A)

Variability Bugs in Highly-Configurable Systems: A Qualitative Analysis

Iago Abal
iago@itu.dk

Jean Melo
jeanmelo@itu.dk

Ștefan Stănciulescu
scas@itu.dk

Claus Brabrand
brabrand@itu.dk

Márcio Ribeiro
marcio@ic.ufal.br

Andrzej Wąsowski
wasowski@itu.dk

ABSTRACT

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**

Keywords

Bugs; Feature Interactions; Linux; Software Variability.

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 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, 36, 26, 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

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

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.
- *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).

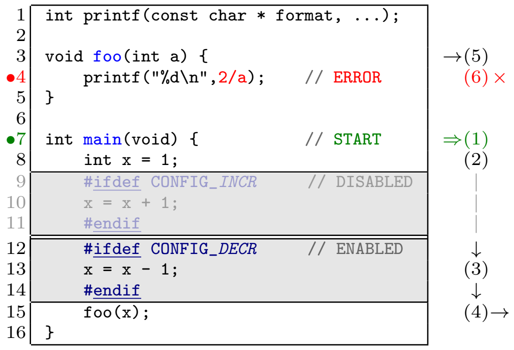
2. BACKGROUND

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 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

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



```
 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  }
```

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

unrestricted feature model ($\psi_{\text{FM}} =$ `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 $\kappa_0 = \neg\texttt{INCR} \land \texttt{DECR}$ and $\kappa_1 = \texttt{INCR} \land \texttt{DECR}$.

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

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 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

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

[2] We merge observations 7 and 8 from [1] due to overlap.

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, 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.[6] 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 [7], Apache [8], BusyBox [9], and Marlin [10] 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)

[3] https://bugzilla.kernel.org/

[4] https://lkml.org/

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

[6] 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.

[7] http://git.kernel.org/

[8] http://git.apache.org/httpd.git

[9] http://git.busybox.net/busybox/

[10] http://github.com/MarlinFirmware/MarlinDev

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

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.

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

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.

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

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[12] 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

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

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

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 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)[13]—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`[14] 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 background. Obtaining the description is often nontrivial. 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.

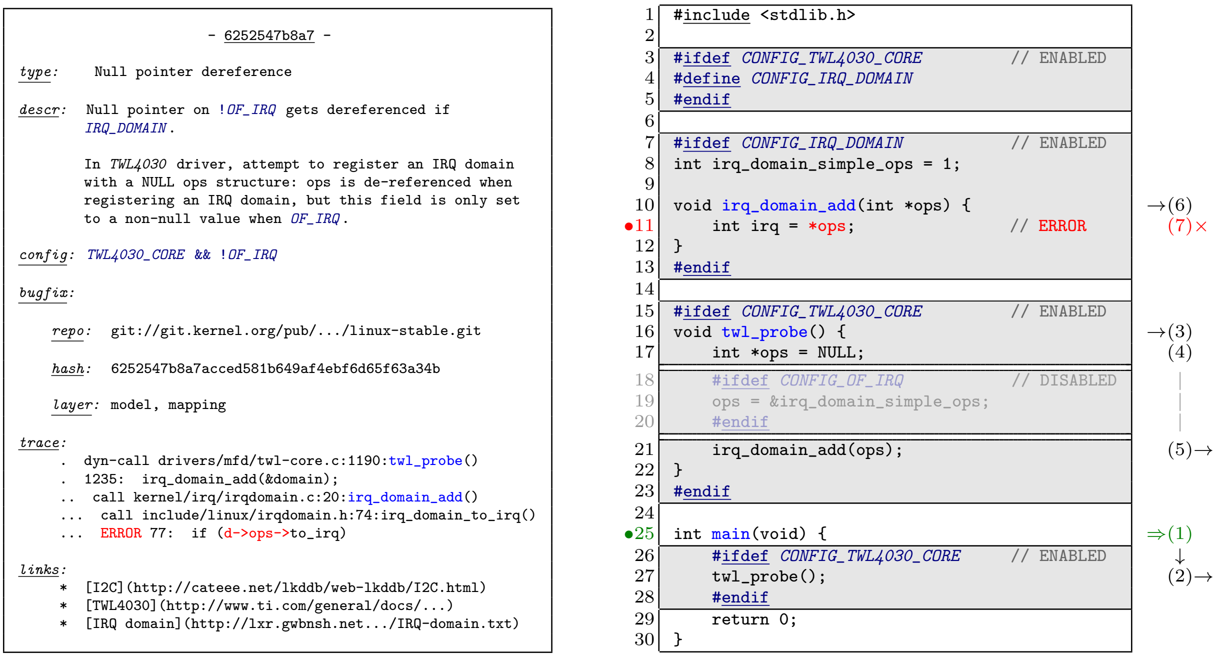
We add a one-line header to the description, here shown in bold, to help identification and listing of bugs.

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)

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

[14] 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.

```
#include <stdlib.h>

#ifdef CONFIG_TWL4030_CORE           // ENABLED
#define CONFIG_IRQ_DOMAIN
#endif

#ifdef CONFIG_IRQ_DOMAIN             // ENABLED
int irq_domain_simple_ops = 1;

void irq_domain_add(int *ops) {
    int irq = *ops;                  // ERROR
}
#endif

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

int main(void) {
    #ifdef CONFIG_TWL4030_CORE       // ENABLED
    twl_probe();
    #endif
    return 0;
}
```

(b) Simplified version.

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

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/i2c-core.c, the generic loading function for *I2C*[15] 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 state-

[15] 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
```

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

ments and control-flow constructs, mapping information and function calls. We keep the original identifiers for features, functions and variables. However, we abstract away dynamic dispatching via function pointers, 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 `TWL4030_CORE` ∧ ¬`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`.

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`.

```
#include <stdlib.h>

#ifdef CONFIG_TWL4030_CORE         // ENABLED
#define CONFIG_IRQ_DOMAIN
#endif

#ifdef CONFIG_IRQ_DOMAIN           // ENABLED
int irq_domain_simple_ops = 1;
#endif

int main(void) {                                ⇒(1)
    #ifdef CONFIG_TWL4030_CORE     // ENABLED     ↓
    int *ops = NULL;                             (2)
    #ifdef CONFIG_OF_IRQ           // DISABLED    |
    ops = &irq_domain_simple_ops;                 |
    #endif                                        |
    int irq = *ops;                // ERROR      (3)×
    #endif
    return 0;
}
```

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

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).

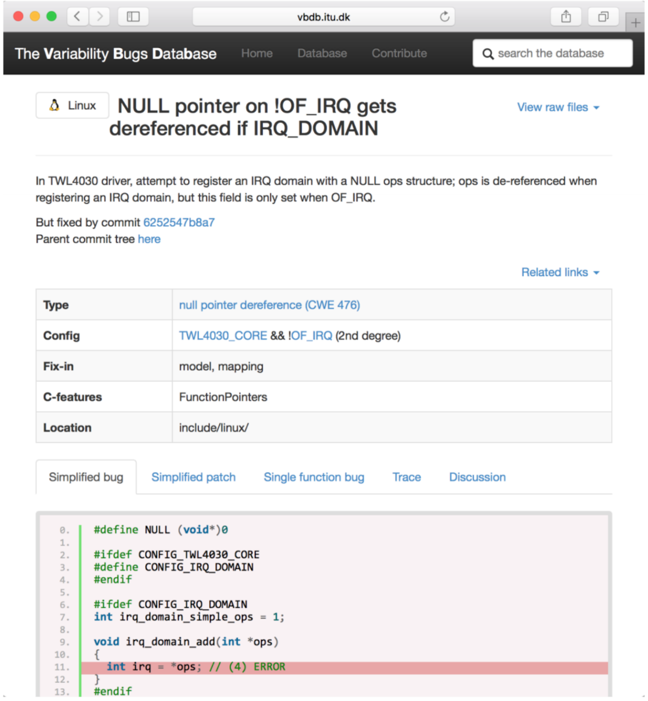
Traceability Information. We store the URL of the repository, in which the bug fix is applied, the commit *hash*, and links to relevant context information about the bug, in order to support independent verification of our analysis.

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.[16] Figure 7 shows a screenshot of our Web UI database for our sample bug `6252547b8a7` from Fig. 4.

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.

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



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 when OF_IRQ.

But fixed by commit 6252547b8a7
Parent commit tree here

Type	null pointer dereference (CWE 476)
Config	TWL4030_CORE && !OF_IRQ (2nd degree)
Fix-in	model, mapping
C-features	FunctionPointers
Location	include/linux/

```c
#define NULL (void*)0

#ifdef CONFIG_TWL4030_CORE
#define CONFIG_IRQ_DOMAIN
#endif

#ifdef CONFIG_IRQ_DOMAIN
int irq_domain_simple_ops = 1;

void irq_domain_add(int *ops)
{
  int irq = *ops; // (4) ERROR
}
#endif
```

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

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.

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.

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**

Figure 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.)

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**

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

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*[17] 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.

Conclusion for RQ1

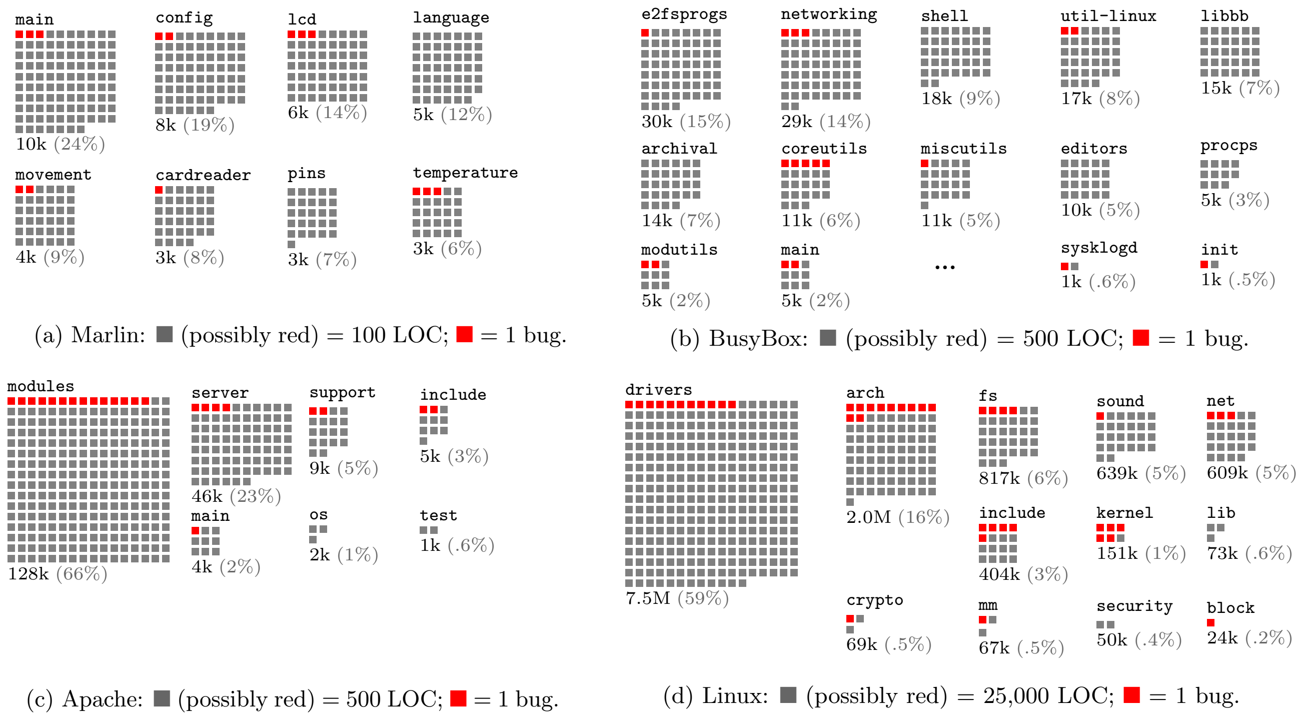
We are now ready to answer RQ1. Based on analyzing four highly-configurable systems, we conclude that:

> **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). Conse-

[17] 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.

Figure 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.

quently, 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 [39, 29, 2]. (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.

> CONFIRMED OBSERVATION 5: The use of a function, variable, macro, or type may involve *implicit*

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

Figure 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).

> *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 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*[18] 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.

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,

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

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**

Figure 12: Bug-fixing layers.

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**

Figure 13: Variability degrees.

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 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.

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 "`Accept-Encoding:  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 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.)

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**

Figure 14: Presence conditions under which the bugs occur.

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)

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

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 [18, 20, 13]), as well as other heuristic strategies such as *all-enabled*, *all-disabled*, *code-coverage* [53, 54, 52] 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[19] 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.

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

[19] 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].

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.[20] 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 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 43$^{\text{rd}}$ 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

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

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.[21] 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 [39, 29, 2].

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 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

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

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.

10. REFERENCES

[1] I. Abal, C. Brabrand, and A. Wasowski. 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, 2014.

[2] M. Al-Hajjaji, F. Benduhn, T. Thüm, T. Leich, and G. Saake. Mutation operators for preprocessor-based variability. In *Proceedings of the Tenth International Workshop on Variability Modelling of Software-intensive Systems*, VaMoS '16, pages 81–88, New York, NY, USA, 2016. ACM.

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

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

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

[6] D. Benavides, S. Segura, and A. R. Cortés. Automated analysis of feature models 20 years later: A literature review. *Inf. Syst.*, 35(6):615–636, 2010.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[23] M. D. Ernst, G. J. Badros, and D. Notkin. An empirical analysis of c preprocessor use. *IEEE TRANSACTIONS ON SOFTWARE ENGINEERING*, 28(12):2002, 2002.

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

[25] C. L. Goues, S. Forrest, and W. Weimer. Current challenges in automatic software repair. *Software Quality Journal*, 21(3):421–443, 2013.

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

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

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

[29] A. F. Iosif-Lazar, J. Melo, A. S. Dimovski, C. Brabrand, and A. Wasowski. Effective analysis of C programs by rewriting variability. *CoRR*, abs/1701.08114, 2017.

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

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

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

[33] C. Kästner, S. Apel, and M. Kuhlemann. Granularity in software product lines. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, pages 311–320, New York, NY, USA, 2008. ACM.

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

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

[36] K. Lauenroth, K. Pohl, and S. Toehning. Model checking of domain artifacts in product line engineering. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, ASE '09, pages 269–280, Washington, DC, USA, 2009. IEEE Computer Society.

[37] J. Liebig, S. Apel, C. Lengauer, C. Kästner, and M. Schulze. 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, pages 105–114, New York, NY, USA, 2010. ACM.

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

[39] F. Medeiros, C. Kästner, M. Ribeiro, R. Gheyi, and S. Apel. A comparison of 10 sampling algorithms for configurable systems. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 643–654, New York, NY, USA, 2016. ACM.

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

[41] J. Melo, C. Brabrand, and A. Wąsowski. How does the degree of variability affect bug finding? In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 679–690, New York, NY, USA, 2016. ACM.

[42] J. Melo, F. B. Narcizo, D. W. Hansen, C. Brabrand, and A. Wasowski. Variability through the eyes of the programmer. In *Proceedings of the 25th International Conference on Program Comprehension*, ICPC '17, pages 34–44, Piscataway, NJ, USA, 2017. IEEE Press.

[43] M. Mendonça, A. Wasowski, and K. Czarnecki. Sat-based analysis of feature models is easy. In D. Muthig and J. D. McGregor, editors, *Software Product Lines, 13th International Conference, SPLC 2009, San Francisco, California, USA, August 24-28, 2009, Proceedings*, volume 446 of *ACM International Conference Proceeding Series*, pages 231–240. ACM, 2009.

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

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

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

[47] N. Palix, G. Thomas, S. Saha, C. Calvès, J. Lawall, and G. Muller. Faults in linux: Ten years later. *SIGARCH Comput. Archit. News*, 39(1):305–318, Mar. 2011.

[48] L. Passos, L. Teixeira, D. Nicolas, S. Apel, A. Wasowski, K. Czarnecki, P. Borba, and J. Guo. 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.

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

[50] J. Slaby, J. Strejček, and M. Trtík. ClabureDB: Classified Bug-Reports Database. In R. Giacobazzi, J. Berdine, and I. Mastroeni, editors, *Verification, Model Checking, and Abstract Interpretation*, volume 7737 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2013.

[51] Ş. Stănciulescu, S. Schulze, and A. Wąsowski. Forked and Integrated Variants in an Open-Source Firmware Project. In *31st International Conference on Software Maintenance and Evolution*, ICSME'15, 2015.

[52] R. Tartler. Finding and burying Configuration Defects in Linux with the undertaker. 2011.

[53] R. Tartler, C. Dietrich, J. Sincero, W. Schröder-Preikschat, and D. Lohmann. Static analysis of variability in system software: The 90, 000 #ifdefs issue. In G. Gibson and N. Zeldovich, editors, *2014 USENIX Annual Technical Conference, USENIX ATC '14, Philadelphia, PA, USA, June 19-20, 2014.*, pages 421–432. USENIX Association, 2014.

[54] R. Tartler, D. Lohmann, C. Dietrich, C. Egger, and J. Sincero. Configuration coverage in the analysis of large-scale system software. *Operating Systems Review*, 45(3):10–14, 2011.

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

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

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

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

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

A Quantitative Analysis of Variability Warnings in Linux (Paper 2B)

A Quantitative Analysis of Variability Warnings in Linux

Jean Melo, Elvis Flesborg, Claus Brabrand, Andrzej Wąsowski
IT University of Copenhagen, Denmark
{jeanmelo,efle,brabrand,wasowski}@itu.dk

ABSTRACT

In order to get insight into challenges with quality in highly-configurable software, we analyze one of the largest open source projects, the LINUX kernel, and quantify basic properties of configuration-related warnings. We automatically analyze more than 20 thousand valid and distinct random configurations, in a computation that lasted more than a month. We count and classify a total of 400,000 warnings to get an insight in the distribution of warning types, and the location of the warnings. We run both on a *stable* and *unstable* version of the LINUX kernel. The results show that LINUX contains a significant amount of configuration-dependent warnings, including many that appear harmful. In fact, it appears that there are no configuration-independent warnings in the kernel at all, adding to our knowledge about relevance of family-based analyses.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors

General Terms

Experimentation

Keywords

Variability Warnings, Linux, Quantitative Analysis, Highly-Configurable Systems, Preprocessors

1. INTRODUCTION

Modern highly-configurable systems include both large industrial product lines [6, 16, 3, 2] and open-source systems of various sizes, up to the LINUX kernel with more than twelve million lines of code and 15 thousand configuration options (*features*) [4]. Features are used to tailor functional and non-functional requirements of a system to the needs of a particular customer. Features combined with a suitable reuse-oriented architecture and software development processes, allow to increase adaptability while maintaining low per-variant cost.

A multitude of technologies can be used to implement configurable systems: object-oriented patterns, aspects, domain-specific languages and code generation, plugin mechanisms, and so on. Among these, the conditional compilation directives of the C preprocessor (CPP) are one of the *oldest*, the *simplest*, and the *most popular* [7, 11, 8] mechanisms in use, especially in the systems domain. Preprocessor directives, like `#ifdef` and `#endif`, enclose the variable code that can be *included* or *excluded* for different selections of features (configurations).

It is commonly assumed that developing highly-configurable systems—being a form of meta-programming—is more difficult than developing single-variant software. A clear challenge is that highly-configurable software can only be handled one variant at a time by conventional software development tools (static analyzers, compilers, testing tools, etc). Using preprocessor just adds to this difficulty. Despite widespread adoption, preprocessors obfuscate the source code, reduce comprehensibility and increase error-proneness [17, 9].

In this short paper, we report on a simple, but extensive experiment that investigates main properties of compilation warnings appearing in different configurations (and two different trees) of the LINUX kernel project. We use sampling [18] across many configurations to investigate what kind of compilation warnings are most common, and configuration-dependent, to see in which subsystems they appear, and whether they are more likely to appear in unstable source trees, than in stable releases.

We use warnings as a proxy for unintended quality issues. Warnings are undesirable in mature code; it is a common practice to disallow code with compilation errors from being committed. Maintainers see warnings as a heuristic indication of low-quality code. Unfortunately, eliminating warnings from highly-configurable code is difficult, because compilers only report them for one configuration at a time. Also, warnings are often produced using the same, or very similar, static analysis techniques as used for detecting errors. Quantitative characteristics of warnings are thus interesting for tool builders, who work on family-based static analysis tools. First, they show that evaluating analysis tools on systems of the size of the LINUX kernel using sampling is feasible. Second, they show what kind of warnings appear frequently, which allows to scope and prioritize work on lifted analyses for these warnings (so that sampling is not necessary, and warnings can be produced with higher reliability).

VaMoS '16, January 27-29, 2016, Salvador, Brazil
© 2016 ACM. ISBN 978-1-4503-4019-9/16/01…$15.00
DOI: http://dx.doi.org/10.1145/2866614.2866615

Warning statistics can be efficiently collected for a large number of configurations. The very small number of compilation errors, especially in stable releases, makes designing quantitative studies difficult. Frequency of warnings is higher than of errors. Moreover, compilation errors are often caused not by mistakes, but by deficiencies of the build environment (for instance absence of configuration-dependent dependencies). Since classifying the cause of error cannot be done automatically, collecting error distribution data automatically is difficult, while it is entirely feasible for warnings.

Our study reveals that the most common warnings involve *dead code* (warnings: `unused-function` and `unused-variable`) and *uninitializations*. Interestingly, it appears that *no warnings* are configuration independent in Linux. All warnings that survive the development process, and are committed to the repositories, even in stable releases, are configuration dependent. We also find that the `drivers/` and `include/` subsystems contain most warnings, whereas there is much less warnings in core subsystems `kernel/` and `security/` of Linux. Additionally, we observe that the unstable version contains more warnings than the stable version, indicating that the Linux process for preparing stable releases does help to reduce configuration-dependent warnings.

We hope that indication of which areas are hot in warnings can be useful for researchers working on bug finding in Linux, showing which subsystems are good candidates as analysis subjects. Also the study seems to confirm the, commonly held but rarely followed, recommendation to focus bug-finding studies (especially, studies on building bug-finders) on unstable trees of Linux, as opposed to stable. Moreover, our study shows that the errors survive all the way to the stable version, so the variability-aware tools could help to substantially decrease the bug density in Linux.

The paper is structured as follows. Section 2 describes the study goal and methodology. Section 3 presents the data and findings, followed by discussion of threats to validity in Section 4. We summarize the related work in Section 5 and conclude in Section 6.

2. OBJECTIVE AND METHOD

Our objective is to quantitatively analyze configuration-dependent warnings in the Linux kernel by checking a large number of randomly generated configurations. This includes addressing the following research questions:

RQ1: Are *configuration-dependent warnings* frequent in the Linux kernel? What are the *most common* configuration-dependent warnings in Linux?

RQ2: In which *sub-systems* of the Linux kernel do most configuration-dependent warnings occur?

RQ3: What are the differences regarding configuration-dependent warnings between an *in-development* version and a *stable* version of Linux?

RQ1 allows us to assess quantitatively at a large scale whether configuration-dependent problems are rare or common in the Linux kernel project. RQ2 allows to see whether configuration-dependent problems are more common in some areas than others. RQ3 sheds light on choosing subjects for empirical studies of bugs, bug finding and evaluation of analysis tools for the Linux kernel. In this perspective this study is exploratory: it informs selection of subject data for future research on bug finding in the kernel and similar projects.

We follow a three-step method: First, we generate random configurations. Second, we collect the warning messages returned by a compilation. Third, we reflect on the aggregated data to answer our research questions. We discuss details of these steps below.

Generation of Random Configurations. We generate random configurations using `randconfig`, a built-in facility of the Linux kernel build system. It produces a file with configuration (so called `.config` file), with values for all features decided. This file is an input for the build system.

To the best of our knowledge, generating uniformly distributed solutions to a constraint system over 15 thousand variables is not feasible with state of the art PSAT tools, which leaves `randconfig` as a viable approximation. While (possibly) not uniformly distributed, `randconfig` is a reproducible mechanism. The `randconfig` mechanism guarantees generating valid configurations, and thus so very efficiently (within seconds).

Compiling and Data Collection. We compile each generated configuration using the `make all` command with the gcc compiler, activating all warnings (the `-Wall` option), which is a common and recommended practice in open source development in C. We collect the warning messages. A warning output contains a *bug type*, a *filename*, *line number*, and a *message* describing the warning.

We repeat the experiment for two different versions of the Linux kernel: a latest stable, v4.1.1, and a two months old in-development from the linux-next tree.[1] The random configurations are selected from the `x86` architecture of the Linux kernel, which decreases the amount of technical problems with building them on a single machine.

Data Analysis. After collecting all the data, we analyzed the warning messages, classifying them by type and location (sub-system).

Operation. The experiment has been carried out on two machines, a 32 core, 2.8MHz, 128GB RAM server (average time to generate and compile one configuration of around 1 minute and 35 seconds) and a conventional laptop with a 4 core, 2.5 MHz CPU and 4 GB of RAM. We ran the experiment for a month on these two machines which allowed to compile 42,060 kernels of Linux. Half of the compilations were carried out on a *stable* version of Linux, the other half (in the same configurations) on an *in-development* version of Linux. In other words, we analyzed 21,030 valid and distinct configurations in two different versions of the Linux kernel: a *stable* version and an *unstable* one, resulting in 42,060 compilations in total.

3. DATA ANALYSIS AND RESULTS

We now present the results of compiling 42,060 kernels (21K in a *stable* version and 21K in an *unstable* version) of Linux using gcc with all warnings enabled. All compilations produced a total of 400,000 warnings (i.e., an average of about ten warnings per compilation). The highest number of warnings produced by a single compilation was 111 (warnings) and 226 configurations compiled without warnings. Obviously, many of the same warnings are found over and over because the same code base in which they occur is included in many different configurations.

[1] next-20150402-22
https://git.kernel.org/cgit/linux/kernel/git/next/linux-next.git

#	Warning	Percent
1	unused-function	59 %
2	maybe-uninitialized	45 %
3	unused-variable	29 %
4	uninitialized	19 %
5	pointer-to-int-cast	17 %
6	frame-larger-than=	14 %
7	array-bounds	11 %
8	return-type	8 %
9	int-to-pointer-cast	8 %
10	overflow	7 %
11	implicit-function-decl	6 %
12	unused-label	5 %
13	deprecated-declarations	5 %

Figure 1: Most common warnings in the *stable* Linux Kernel (according to how many percent of configurations produce the given kinds of warnings, when compiled).

#	Subsystem Directory	Absolute Size	Relative Size	Warning Percentage
1	drivers/	7,713	59 %	64 %
2	include/	423	3 %	40 %
3	crypto/	69	1 %	17 %
4	fs/	831	6 %	14 %
5	net/	631	5 %	10 %
6	arch/x86/	235	2 %	9 %
7	lib/	74	1 %	9 %
8	mm/	68	1 %	8 %
9	kernel/	155	1 %	6 %
10	sound/	659	5 %	4 %
11	block/	24	0 %	1 %
12	security/	50	0 %	0 %

Figure 2: Rank of subsystems in the *stable* Linux Kernel (according to how many percent of configurations produce warnings, when compiled). Size is given in KLOC.

The experiment materials are available online at https://github.com/elvios/quantify_linux_errors (including scripts, notes, and reports).

RQ1: Most Common Warnings

Figure 1 shows the most common warnings occuring in the *stable* version of Linux. A warning is said to *occur* in a configuration of Linux, if a corresponding warning message appears as a result of compiling the configuration of Linux. We see that the most abundant warning is `unused-functions`; functions that are declared, but not used. Such functions are technically *dead code*, but do occupy memory. Gcc does, in fact, *not* remove dead code such as *unused functions* and *unused variables*.[2] To remove *dead code*, the GCC compiler needs to be invoked with options `-fdata-sections` and `-ffunction-sections` in order to keep data and functions in separate sections; subsequently, the *linker* needs the `-gc-sections` flag to be able to finally remove unused sections. Interestingly, the removal of such dead code is *not* available among the optimizations commonly employed in the Linux kernel; it is included in *neither* `-O0` (optimization level *zero*), `-O1` (*one*), *nor* `-O2` (*two*). Note that Linux ubiquitously runs on many small embedded devices such as TiVo and similar DVR devices, network routers, and smartwatches —even credit-card sized single-computer boards such as Raspberry Pi—where memory is a limited resource.

The second most common warning, `maybe-uninitialized`, occurs in 45% of configurations; it occurs whenever GCC determines the existence of *an* execution path from a variable declaration to a usage without prior initialization. If the uninitialization occurs along *all* paths, the warning is strengthened to a (definitely) `uninitialized` which happens in 19% of configurations. Such warnings are quite serious. The third most common warning is `unused-variable` (occuring in 29% of configurations); variables that are declared, but not used. Such variables constitute *dead code* and will take up space on the heap or stack and may thus translate to wasted memory. In total, Figure 1 details the frequencies of thirteen types of warnings commonly occuring in Linux.

We are now ready to answer our first research question with the observation that:

> Observation 1a: The most common warnings in the *stable* Linux kernel involve *dead code* (warnings: `unused-function` and `unused-variable`) and *uninitializations* (warnings: `uninitialized` and `maybe uninitialized`).

In terms of variability, it is interesting to note that none of the warnings are at 100% which would be the case for configuration-independent warnings. We notice from Figure 1 that all warnings in the stable Linux seem dependent on configurations (choice of enabled/disabled features). Hence:

> Observation 1b: All warnings in Linux appear to be *configuration-dependent* (i.e., warnings that occur in some configurations and *not* in others).

Finally, we observe that:

> Observation 1c: Most configurations appear to contain warnings.

In fact, among our 21,030 configurations compiled in the *stable* version, only 226 did not produce warnings. Even though the Linux kernel developers try to improve code quality[3] —making sure that the code follows the coding style, and eliminating the static code checker errors and warnings—, it seems that dealing with variability (i.e., maintaining thousands of features and their interactions) is complicated.

RQ2: Subsystems with Most Warnings

Figure 2 shows the frequency of warnings in the *stable* version of Linux, according to the *subsystems* in which they occur. (Note that we use directories as proxies for *subsystems*.) We present both absolute and relative size for each subsystem, in which we normalize the subsystem size by dividing it by the total size (= 13 MLOC). A warning is said to *occur* in a subsystem, if a corresponding warning message appears designating a location within the given subsystem. In the following, we disregard *smaller* subsystems below ten thousand lines of code: `virt/` (6.8 KLOC), `ipc/` (6.4k), `init/` (2.0k), and `usr/` (0.6k). We also disregard Linux *infrastructure* such as `tools/` (102k), `scripts/` (44k), and `samples/` (2.1k).

The subsystem most frequently producing warnings is also the largest subsystem of Linux with 7 MLOC: `drivers/`. This subsystem produces warnings in more than half (64%) of configurations. The subsystem `include/` (a directory with header files) causes warnings in almost half (40%) of configurations.

[2] https://gcc.gnu.org/ml/gcc-help/2003-08/msg00128.html

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

Warning	Stable	In-Dev.
unused-variable	29 %	51 %
int-to-pointer-cast	8 %	25 %
implicit-function-decl	6 %	23 %
frame-larger-than=	14 %	8 %

(a) Kinds of warnings.

Subsystem	Size	Stable	In-Dev.
arch/x86/	235	9 %	14 %
mm/	68	8 %	13 %
kernel/	155	6 %	3 %
sound/	659	4 %	2 %

(b) Subsystems with most warnings.

Figure 3: Significant differences in warnings in the *stable* vs. *in-development* version of Linux.

Based on this data, we formulate our second observation, related to our second research question:

> Observation 2: The **drivers/** subsystem and **include/** header files produce warnings in around *half of all configurations*; whereas *core* subsystems such as **kernel/** and **security/** rarely produce warnings.

In particular, this means that sampling using **randconfig** is likely to hit warnings in **drivers/** and **include/**, but unlikely to hit warnings in core subsystems such as **kernel/** and **security/**. In case of using another sampling technique, the result is still unclear (as the probability distribution of **randconfig** is hard to describe).

RQ3: Stable vs. In-Development Version

Figure 3 shows a comparison of differences in warnings in the *stable* versus the *in-development* version of Linux.

Figure 3a charters significant differences in frequencies of warning kinds in the two version. We see that variables declared, but not used (**unused-variables**) proliferate in the *in-development* version, occurring in twice as many configurations as that of the *stable* version. Presumably, the lack of rigorous testing that the *stable* version undergoes before "release" does not reveal such superfluous declarations. Also, (integer to pointer) *type casts* and *implicit function declarations* seems to occur more frequently in configurations in the (*un-stable*) development version. Only one kind of warning, **frame-larger-than=**N, occurs less frequently in the *in-development* version. This kind of warning is reported when a function seems to require allocation of more memory on the stack than a compile-time given constant, N. The remaining kinds of warnings do not spawn significant differences between the two versions.

Figure 3b shows differences in the locations of warning among the two versions of Linux. The subsystems that has a percentage point diffrence lower than 2% are not shown. We see warnings occur more frequently in the **arch/x86/** and **mm/** (memory management) subsystems. For subsystems, **kernel/** and **sound/**, we see a decline in the frequency of warning-configurations.

Related to our third and final research question, we observe, not surprisingly, that:

> Observation 3: There appear to be more warnings in the *in-development* version than the *stable* version of Linux (especially, *unused variables*, *type casts*, and *implicit function declarations*).

This is interesting for two reasons. First, it shows that developers do fix many problems before the code becomes stable, and if they had the right tools, this process could possibly be speeded up. In fact, official Linux kernel patch submission guidelines encourage removing warnings to avoid clutter of messages from the compiler.[4] The Linux foundation also urges the developers to heed the warnings produced by the compiler:[5]

> *"Contemporary versions of gcc can detect (and warn about) a large number of potential errors. Quite often, these warnings point to real problems. Code submitted for review should, as a rule, not produce any compiler warnings. When silencing warnings, take care to understand the real cause and try to avoid "fixes" which make the warning go away without addressing its cause."*

Second, it also shows that the errors survive all the way to the stable version, so the variability-aware tools that would be more precise or more accurate than the developers, could help to substantially decrease the bug density in Linux.

Overall, it seems that Linux developers are good at fixing issues in in-development versions for stable version releases.

4. THREATS TO VALIDITY

We now discuss *internal* and *external* threats to validity of our experiment.

Internal Validity. *Randomization?* The built-in random configuration generator that comes with Linux, **randconfig**, does not provide a perfect *uniform distribution* over the entire space of *valid* configurations. Sequentially, for each feature (according to the order in the KConfig files), a coin toss will decide whether a feature should be *enabled* or *disabled* (provided choices for previous features have not already determined a unique choice). Hence, it is *biased* towards features higher up in the feature model tree. We use **randconfig** because it is the official randomized configuration strategy Linux developers use **randconfig**, in practice, to generate valid random configurations.

Feature differences? In the *in-development* Linux version, there are 20 features more than in the stable version. For this reason, we create configurations for the *in-development* version, and then re-use those for the *stable* version. This might result in some unknown feature being set on the stable version, but there will be no code corresponding to the features, so no harm is done.

Error filtering? Finally, we filtered out erroneous configurations (17% of all generated configurations). As previously mentioned, errors were dominated by build errors caused by our hardware and installation environment. Hence, we discard errors and focus on warnings as indicators of quality. In particular, when building certain firmware drivers in Linux, external proprietary drivers are needed before they can be built. This firmware is not in the kernel code, but must be downloaded from the hardware vendors homepages. In this

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

[5] https://www.linuxfoundation.org/content/how-participate-linux-community-0

study, we did not include these firmware drivers, since they are in a sense not a part of the open source Linux kernel and hence beyond the scope of this paper.

External Validity. *Only x86 Architecture?* We ran the experiment only on the `x86` architecture. In total, there are more than 20 different architectures supported. The size of `arch/x86/` is 12% of that of `arch/` (235 out of 2,025 thousand lines of code). To run on all of the other architectures, we would need to rig the build system with a cross-compiler for every architecture and hardware, which is cumbersome and error-prone. Obviously, this limits the generalizability of our experiment with respect to architectures. Still, there are more than 10 thousand features in the `x86` architecture.

Other versions of Linux? In 2008, based on `randconfig`, Linux developers started systematic sampling-based testing by compiling (and booting) thousands of random configurations of Linux (on different hardware), every day [5]. From this, bug reports are automatically compiled and sent to relevant developers (code authors). We thus expect significant differences in occurences of warnings (and errors) before that (i.e., before version 2.6.24)

Other systems? We decided to focus exclusively on Linux, so our findings do not automatically generalize to other highly-configurable systems. The significance of the Linux kernel project itself justifies investigation, even at the cost of limiting generalizability.

Warnings vs. real bugs? We do not know whether all warnings constitute real bugs. For instance, the warning `maybe-uninitialized` may be caused by an infeasible path that the analyzer of the gcc compiler considers due to over approximation. Similarly, a `pointer-to-int-cast` might be legal, as encoding pointers as integers (or as void pointers) is a pattern to implement generic data structures in Linux. Some other problems, like unused variables, might not be immediate problems at all (unless memory consumption is an issue in a particular piece of code). However, warnings such as `uninitialized`, `array-bounds` and `overflow` may represent real bugs. For example, an `uninitialized` warning occurs when the analyzer of the gcc compiler finds an uninitialization that occurs along all paths, leading to a real bug. Such warnings are quite serious. We found that 19% of configurations contain an `uninitialized` warning.

5. RELATED WORK

Many studies have investigated *errors* in highly-configurable systems, including Linux [17, 10, 14, 15, 12, 1].

Abal et al. [1] have qualitatively identified 42 variability bugs in the Linux kernel. They observe that variability bugs are not confined to any particular type of bug, (error-prone) feature, or source code location. Our experiment confirms these qualitative insights and complements them with quantitative data.

Medeiros et al. [12] investigated syntactic errors in preprocessor-based systems. They observed that developers introduce syntax errors when changing existing code and adding preprocessor directives. In this paper, rather than syntactic errors, we study a variety of 33 warnings enabled by the gcc `-Wall` flag, including undeclared and unused identifiers.

Medeiros et al. [13] have also conducted an empirical study on configuration-related issues, specifically, investigating undeclared and unused identifiers. They found 39 configuration-related issues, including 14 (36%) undeclared functions, 2 (5%) undeclared variables, 7 (18%) unused functions, and 23 (41%) unused variables. In other words, they noticed that 59% of the issues are related to unused functions and variables. We complement this finding by generating randomly thousands of configurations from the Linux kernel, in which we observed that the most abundant warning is unused function and the third most is unused variable. Besides that, we ranked both warnings and subsystems, e.g., the `drivers/` subsystem and `include/` header files produced warnings in around half of all configurations we compiled; whereas core subsystems such as `kernel/` and `security/` rarely produced warnings.

Nadi et al. [15] mined the Linux repository to study variability anomalies. 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. We, in turn, focus on variability warnings in the Linux kernel from a quantitative perspective.

6. CONCLUSION

We have presented an experiment quantifying basic properties of configuration-related warnings in the Linux kernel. We have automatically analyzed more than 20 thousand valid and distinct random configurations.

We observe 13 different types of warnings appearing in the Linux kernel. The majority of these are regarding *dead code* (e.g., unused variable) and *uninitializations*. We also find that the `drivers/` and `include/` subsystems contain warnings in about half of all configurations, whereas much fewer warnings originate from the *core* subsystems `kernel/` and `security/`. Additionally, we observe that there are generally more warnings in the in-development version of Linux than in the stable version.

Finally, we hope that indication of which areas are hot in warnings can be useful for further research on bug finding in Linux, showing which subsystems are good candidates as analysis subjects. Also, this study seems to confirm the, commonly held but rarely followed, recommendation to focus bug-finding studies on unstable trees of Linux, as opposed to stable. Moreover, our study shows that the errors survive all the way to the stable version, so the variability-aware tools could help to substantially decrease the bug density in Linux.

As future work, we aim to evaluate whether the current results hold for another technique that generates random configurations instead of `randconfig` (as the probability distribution of `randconfig` is not thoroughly representative).

Acknowledgments. We thank Rene Rydhof Hansen for feedback and encouraging us to pursue publication of these results. Melo is funded by Brazilian Science without Borders Programme, CNPq grant no. 249020/2013-0.

7. REFERENCES

[1] I. Abal, C. Brabrand, and A. Wasowski. 42 Variability Bugs in the Linux Kernel: A Qualitative Analysis. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ASE '14, pages 421–432, New York, NY, USA, 2014. ACM.

[2] T. Berger, D. Nair, R. Rublack, J. M. Atlee, K. Czarnecki, and A. Wasowski. Three cases of feature-based variability modeling in industry. In

ACM/IEEE 17th International Conference on Model Driven Engineering Languages and Systems (MODELS), 2014.

[3] T. Berger, R. Rublack, D. Nair, J. M. Atlee, M. Becker, K. Czarnecki, and A. Wąsowski. A survey of variability modeling in industrial practice. In *Proceedings of the Seventh International Workshop on Variability Modelling of Software-intensive Systems*, VaMoS '13, pages 7:1–7:8, New York, NY, USA, 2013. ACM.

[4] T. Berger, S. She, R. Lotufo, A. Wasowski, and K. Czarnecki. A study of variability models and languages in the systems software domain. *Software Engineering, IEEE Transactions on*, 39(12):1611–1640, Dec 2013.

[5] Y. Chen, F. Wu, K. Yu, L. Zhang, Y. Chen, Y. Yang, and J. Mao. Instant bug testing service for linux kernel. In *High Performance Computing and Communications 2013 IEEE International Conference on Embedded and Ubiquitous Computing (HPCC_EUC), 2013 IEEE 10th International Conference on*, pages 1860–1865, Nov 2013.

[6] P. Clements and L. Northrop. *Software Product Lines: Practices and Patterns.* Addison-Wesley, 2002.

[7] M. D. Ernst, G. J. Badros, and D. Notkin. An empirical analysis of C preprocessor use. *IEEE Transactions on Software Engineering*, 28:1146–1170, 2002.

[8] C. Kästner, S. Apel, and M. Kuhlemann. Granularity in Software Product Lines. In *Proceedings of the 30th International Conference on Software Engineering (ICSE)*, pages 311–320. ACM, 2008.

[9] M. Krone and G. Snelting. On the inference of configuration structures from source code. In *Proceedings of the 16th International Conference on Software Engineering (ICSE)*, pages 49–57. IEEE Computer Society Press, 1994.

[10] D. Le, E. Walkingshaw, and M. Erwig. #ifdef confirmed harmful: Promoting understandable software variation. In *IEEE International Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 143–150, 2011.

[11] J. Liebig, S. Apel, C. Lengauer, C. Kästner, and M. Schulze. An analysis of the variability in forty preprocessor-based software product lines. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering (ICSE)*, pages 105–114. ACM, 2010.

[12] F. Medeiros, M. Ribeiro, and R. Gheyi. Investigating preprocessor-based syntax errors. In *Proceedings of the 12th International Conference on Generative Programming: Concepts & Experiences*, GPCE, pages 75–84. ACM, 2013.

[13] F. Medeiros, I. Rodrigues, M. Ribeiro, L. Teixeira, and R. Gheyi. An empirical study on configuration-related issues: Investigating undeclared and unused identifiers. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*, GPCE 2015, pages 35–44, New York, NY, USA, 2015. ACM.

[14] J. Melo and P. Borba. Improving modular reasoning on preprocessor-based systems. In *Software Components Architectures and Reuse (SBCARS), Seventh Brazilian Symposium on*, pages 11–19, 2013.

[15] S. Nadi, C. Dietrich, R. Tartler, R. Holt, and D. Lohmann. Linux variability anomalies: What causes them and how do they get fixed? In *Mining Software Repositories (MSR), 2013 10th IEEE Working Conference on*, pages 111–120, May 2013.

[16] K. Pohl, G. Bockle, and F. J. van der Linden. *Software Product Line Engineering.* Springer, 2005.

[17] H. Spencer and G. Collyer. #ifdef considered harmful, or portability experience with C news. In *Proceedings of the Usenix Summer Technical Conference*, pages 185–198. Usenix Association, 1992.

[18] T. Thüm, S. Apel, C. Kästner, I. Schaefer, and G. Saake. A classification and survey of analysis strategies for software product lines. *ACM Comput. Surv.*, 47(1):6:1–6:45, June 2014.

Effective Analysis of C Programs by Rewriting Variability (Paper 3A)

Effective Analysis of C Programs by Rewriting Variability

Alexandru F. Iosif-Lazar[a], Jean Melo[a], Aleksandar S. Dimovski[a], Claus Brabrand[a], and Andrzej Wąsowski[a]

a IT University of Copenhagen, Denmark

Abstract Context. Variability-intensive programs (program families) appear in many application areas and for many reasons today. Different family members, called variants, are derived by switching statically configurable options (features) on and off, while reuse of the common code is maximized.

Inquiry. Verification of program families is challenging since the number of variants is exponential in the number of features. Existing single-program analysis and verification tools cannot be applied directly to program families, and designing and implementing the corresponding variability-aware versions is tedious and laborious.

Approach. In this work, we propose a range of variability-related transformations for translating program families into single programs by replacing compile-time variability with run-time variability (non-determinism). The obtained transformed programs can be subsequently analyzed using the conventional off-the-shelf single-program analysis tools such as type checkers, symbolic executors, model checkers, and static analyzers.

Knowledge. Our variability-related transformations are *outcome-preserving*, which means that the relation between the outcomes in the transformed single program and the union of outcomes of all variants derived from the original program family is *equality*.

Grounding. We present our transformation rules and their correctness with respect to a minimal core imperative language IMP. Then, we discuss our experience of implementing and using the transformations for efficient and effective analysis and verification of real-world C program families.

Importance. We report some interesting variability-related bugs that we discovered using various state-of-the-art single-program C verification tools, such as Frama-C, Clang, LLBMC.

ACM CCS 2012

- *Software and its engineering → Software creation and management Software verification and validation; Software notations and tools Formal language definitions;*

Keywords Program Families, Variability-related Transformations, Verification, Static Analysis

The Art, Science, and Engineering of Programming

Submitted September 1, 2016

Published January 27, 2017

DOI 10.22152/programming-journal.org/2017/1/1

In *The Art, Science, and Engineering of Programming*, vol. 1, no. 1, 2017, article 1; 25 pages.

1 Introduction

Many software systems today are variability intensive. They permit users to derive a custom variant by choosing suitable configuration options (features) depending on their requirements. There are different strategies for implementing variational systems (program families) [11]. Still, many popular industrial program families from system software (e.g. Linux kernel) and embedded software (e.g. cars, phones, avionics) domains are implemented using annotative approaches such as conditional compilation. For example, #ifdef annotations from the C-preprocessor are used to specify under which conditions, parts of the code should be included or excluded from a variant.

Due to the increasing popularity of program families, formal verification techniques for proving their correctness are widely studied (see [35] for a survey). Analyzing program families is challenging [29]. From only a few compile-time configuration options, exponentially many variants can be derived. Thus, for large variability-intensive software systems, any brute-force approach that derives and analyzes all variants individually one by one using existing single-program analysis tools is very inefficient or even infeasible. Recently, many dedicated family-based (variability-aware) analysis tools have been developed, which operate directly on program families. They produce results for all variants at once in a single run by exploiting the similarities between the variants. Examples of successful family-based analysis tools are applied to syntax checking [25, 20], type checking [24, 8], static analysis [7, 6], model checking [10, 14], etc. Although they are more efficient than the brute-force approach, still their design and implementation for each particular analysis and language is tedious and error prone. Often, these family-based tools are research prototypes implemented from scratch. So it is very difficult to re-implement all optimization algorithms in them that already exist for their single-program industrial-strength counterparts, which have been under development for several decades.

Another approach for efficient variability-aware verification would be to replace compile-time variability with run-time variability (or non-determinism) [37]. In particular, in this work we consider a class of variability-related transformations that transform a program family into a single program, whose outcomes are equal to the union of all outcomes of individual variants. We call the corresponding transformations outcome-preserving. Subsequently, existing single-program analysis tools (verification oracles) that can handle non-determinism (run-time variability) can be used to analyze the generated single program. Finally, the obtained results are interpreted back on the individual variants. The overview of this approach is given in Figure 1. Instead of using specialized variability-aware tools to analyze program families (which would be tedious and labor intensive), our transformation-based approach allows us to use the standard off-the-shelf single-program analysis tools to achieve the same goal. Nevertheless, the limitation of our approach is that we may not obtain the most precise conclusive results for all individual variants. Of course, this depends on the particular analysis and tool that we use.

To demonstrate correctness of our transformation-based approach, we define the transformations formally using IMP, a small imperative language. To model compile-

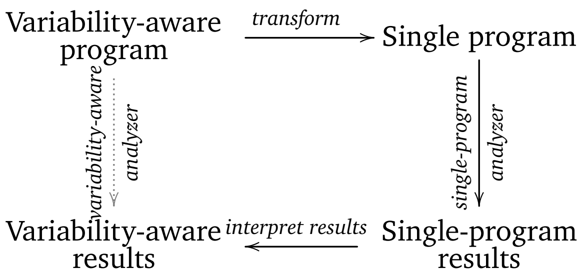


Figure 1 The overview of our transformation-based approach for verification of program families. The single-program analyzer can be any verification oracle for single programs, such as: symbolic executor, type checker, static analyzer, model checker.

time variability, we extend IMP with an "#ifdef" construct for encoding multiple variants, which we call $\overline{\text{IMP}}$ language. To encode run-time variability, we extend IMP with an "or" construct for encoding non-determinism, which we call IMPor language. We define transformations that translate any given $\overline{\text{IMP}}$ program into a corresponding IMPor program. Furthermore, for each transformation we show the relation between the semantics of the input and output programs.

Finally, we report on our experience with implementing and applying our transformations for a full-fledged language, C. The tool, called C Reconfigurator, uses variability-aware parser SuperC [20] for parsing C code with preprocessor annotations, then applies our variability rewrites thus producing a single C program as output. We evaluate our approach on real-world variability intensive C programs with real bugs. We show how some known off-the-shelf single-program analysis tools can be used for efficient and effective verification of such programs.

In summary, this work makes the following contributions:

- A stand-alone variability-related transformation, which transforms a program family into a single program by replacing compile-time variability with non-determinism.
- Correctness of the proposed transformation, which shows that the set of outcomes of the transformed program is equal to the union of sets of outcomes of variants from the input family.
- A prototype tool, C Reconfigurator, which implements the above variability-related transformation for the C language.
- An evaluation of the effectiveness of our transformation-based approach for finding real variability bugs in large variability intensive C software systems.

2 Motivating Example

We begin by showing how our variability transformations work on C program families. Consider a preprocessor-based family of C programs shown in Figure 2 (left column), which uses two (Boolean) features A and B. Our two features give rise to a family of four variants defined by the set of configurations $\mathbb{K} = \{A \wedge B, A \wedge \neg B, \neg A \wedge B, \neg A \wedge \neg B\}$.

```
int foo() {
   int x:= 1;
   #if (A) x:= x+1 #endif;
   #if (B) x:= x-1 #endif;
   return 2/x;
}
```

```
int A:= rand()%2;
int B:= rand()%2;
int foo() {
   int x := 1;
   if (A) x:= x+1;
   if (B) x:= x-1;
   return 2/x; }
```

Figure 2 Before (left column) and after (right column) our transformations

For each configuration a different variant (single program) can be generated by appropriately resolving #if statements. For example, the variant for $A \land B$ will have both features A and B enabled (set to true), thus yielding the following body of $foo()$: int x := 1; x := x+1; x := x-1; return $2/x$. The variant for $\neg A \land \neg B$ is: int x := 1; return $2/x$. In such program families, errors (also known as *variability bugs* [1]) can occur in some variants (configurations) but not in others. In our example program family in Figure 2, the variant $\neg A \land B$ will crash at the return statement when we attempt to divide by zero. On the other hand, the other variants do not contain the division-by-zero error since the value of x at the return statement is: 1 for variants $A \land B$ and $\neg A \land \neg B$, and 2 for $A \land \neg B$.

In Figure 2, we show a single program (right column) obtained by applying our variability-related transformation on the family shown in the left column. All features are first declared as ordinary global variables and non-deterministically initialized to 0 or 1, then all #if statements are transformed into ordinary if-s with the same conditions. Thus, the division-by-zero error is present in this single program and corresponds to a trace when A is initialized to 0 and B to 1. The set of outcomes of the transformed program (Figure 2, right column) is equal to the union of outcomes of all individual variants from the family (Figure 2, left column). Therefore, the division-by-zero error is present in the transformed program.

In general, the transformed program that we obtain from the original program family can be analyzed by various single-program verification tools, in order to find variability errors or to confirm the absence of errors in the given program family.

3 A Formal Model for Transformations

We now introduce the IMP language that we use to demonstrate our transformations and their proofs of correctness. We describe two extensions of IMP: IMPor used to represent run-time variability (non-determinism), and $\overline{\text{IMP}}$ used to represent compile-time variability.

3.1 IMP

We use a simple imperative language, called IMP [32, 34], which represents a regular general-purpose programming language, aimed at the development of single programs. IMP is a well established minimal language, which is used only for presentational purposes here.

Syntax. IMP is an imperative language with two syntactic categories: expressions and statements. Expressions include integer constants, variables, and binary operations. Statements include a "do-nothing" statement skip, assignments, statement sequences, conditional statements, while loops, and local variable declarations. Its abstract syntax is summarized using the following grammar:

$$\begin{array}{lll} e & ::= & n \mid \mathsf{x} \mid e_0 \oplus e_1 \\ s & ::= & \mathsf{skip} \mid \mathsf{x} := e \mid s_0 \mathbin{;} s_1 \mid \mathsf{if}\ e\ \mathsf{then}\ s_0\ \mathsf{else}\ s_1 \mid \mathsf{while}\ e\ \mathsf{do}\ s \mid \mathsf{var}\ \mathsf{x}{:=}e\ \mathsf{in}\ s \end{array}$$

In the above, n stands for an integer constant, x stands for a variable name, and $\oplus$ stands for any binary arithmetic operator. We denote by *Stm* and *Exp* the set of all statements, s, and expressions, e, generated by the above grammar.

Semantics. A state of a program is a *store* mapping variables to values (integer numbers), $\mathit{Val} = \mathbb{Z}$. We write $\mathit{Store} = \mathit{Var} \to \mathit{Val}$ to denote the set of all possible stores. Expressions are computed in a given store, denoted by σ. A function $\mathscr{E} : \mathit{Exp} \times \mathit{Store} \to \mathit{Val}$ defined below by induction on e, maps an expression and a store to a single value, thereby formalizing evaluation of expressions.

$$\mathscr{E}(n,\sigma) = n, \qquad \mathscr{E}(\mathsf{x},\sigma) = \sigma(\mathsf{x}), \qquad \mathscr{E}(e_0 \oplus e_1,\sigma) = \mathscr{E}(e_0,\sigma) \oplus \mathscr{E}(e_1,\sigma)$$

Figure 3 presents the inference rules for a small-step operational semantics for IMP [32, 34]. The notation $\sigma[\mathsf{x} \mapsto n]$ denotes the state that maps x into n and all other variables y into $\sigma(\mathsf{y})$. Following the convention popularized by C, we model Boolean values as integers, with zero interpreted as false and everything else as true (see rules If2 and Wh2, respectively, If1 and Wh1). Note that for variable declarations (see rules Var1 or Var2) we need to restore the declared variable, x, to its earlier global value assigned to x before the declaration, when the scope of declaration has completed. That is why the statement s' in intermediate configurations (rule Var1) is prefixed with variable declarations whose initializations store the local values of x. We can use the inference rules in Figure 3 to define the transition relation: $\langle s,\sigma\rangle \to \gamma$, where γ is either of the form $\langle s',\sigma'\rangle$ or of the form σ'. If γ is of the form $\langle s',\sigma'\rangle$ then the execution of s is not completed and the complex statement s is rewritten into simpler one s', possibly updating the store σ into σ' (for instance, Seq1 or Seq2). If γ is of the form σ' then the execution of s from σ has terminated and the final state is σ' (for instance, Skip or Wh2).

A *derivation sequence* of s starting in store σ can be either a finite sequence $\langle s,\sigma\rangle \to \langle s_1,\sigma_1\rangle \to \ldots \to \sigma'$ (means: s is run in σ and terminates successfully transforming σ to σ' in the process), or an infinite sequence $\langle s,\sigma\rangle \to \langle s_1,\sigma_1\rangle \to \ldots$ (means: s diverges

$$\textsc{Skip}\ \frac{}{\langle \mathsf{skip},\sigma\rangle \to \sigma} \qquad \textsc{Asgn}\ \frac{n=\mathscr{E}(e,\sigma)}{\langle \mathsf{x} := e,\sigma\rangle \to \sigma[\mathsf{x}\mapsto n]} \qquad \textsc{Sq1}\ \frac{\langle s_0,\sigma\rangle \to \langle s_0',\sigma'\rangle}{\langle s_0 ;s_1,\sigma\rangle \to \langle s_0' ;s_1,\sigma'\rangle}$$

$$\textsc{Sq2}\ \frac{\langle s_0,\sigma\rangle \to \sigma'}{\langle s_0 ;s_1,\sigma\rangle \to \langle s_1,\sigma'\rangle} \qquad \textsc{If1}\ \frac{\mathscr{E}(e,\sigma)\neq 0}{\langle \mathsf{if}\ e\ \mathsf{then}\ s_0\ \mathsf{else}\ s_1,\sigma\rangle \to \langle s_0,\sigma\rangle}$$

$$\textsc{If2}\ \frac{\mathscr{E}(e,\sigma) = 0}{\langle \mathsf{if}\ e\ \mathsf{then}\ s_0\ \mathsf{else}\ s_1,\sigma\rangle \to \langle s_1,\sigma\rangle} \qquad \textsc{Wh1}\ \frac{\mathscr{E}(e,\sigma)\neq 0}{\langle \mathsf{while}\ e\ \mathsf{do}\ s,\sigma\rangle \to \langle s; \mathsf{while}\ e\ \mathsf{do}\ s,\sigma\rangle}$$

$$\textsc{Wh2}\ \frac{\mathscr{E}(e,\sigma) = 0}{\langle \mathsf{while}\ e\ \mathsf{do}\ s,\sigma\rangle \to \sigma} \qquad \textsc{Var1}\ \frac{n=\mathscr{E}(e,\sigma) \quad \langle s,\sigma[\mathsf{x}\mapsto n]\rangle \to \langle s',\sigma'\rangle}{\langle \mathsf{var\ x}{:=}e\ \mathsf{in}\ s,\sigma\rangle \to \langle \mathsf{var\ x}{:=}\sigma'(\mathsf{x})\ \mathsf{in}\ s',\sigma'[\mathsf{x}\mapsto\sigma(\mathsf{x})]\rangle}$$

$$\textsc{Var2}\ \frac{n=\mathscr{E}(e,\sigma) \quad \langle s,\sigma[\mathsf{x}\mapsto n]\rangle \to \sigma'}{\langle \mathsf{var\ x}{:=}e\ \mathsf{in}\ s,\sigma\rangle \to \sigma'[\mathsf{x}\mapsto\sigma(\mathsf{x})]}$$

Figure 3 Small-step operational semantics for IMP

when run in σ). We write $[\![s]\!]\sigma$ for the final store σ' that can be derived from $\langle s,\sigma\rangle$ (if the derivation is finite), i.e. $\langle s,\sigma\rangle \to^* \sigma'$, otherwise if the derivation is infinite $[\![s]\!]\sigma$ is undefined (empty). In general, we define:

$$[\![s]\!] = \bigcup_{\sigma\in Store^{\mathsf{Init}}} [\![s]\!]\sigma$$

where $Store^{\mathsf{Init}}$ denotes the set of initial input stores on which s is executed.

3.2 IMPor

Syntax The language IMPor is obtained by extending IMP with a non-deterministic choice operator 'or' which can non-deterministically choose to evaluate either of its arguments.

$$e \quad ::= \quad \ldots \mid e_0\ \mathsf{or}\ e_1$$

Semantics. Since we have non-deterministic construct 'or', it is possible for an expression to evaluate to a set of different values in a given store. Therefore, now we have $\mathscr{E} : Exp \times Store \to \mathscr{P}(Val)$ defined as follows:

$$\mathscr{E}(n,\sigma) = \{n\}, \qquad \mathscr{E}(\mathsf{x},\sigma) = \{\sigma(\mathsf{x})\}, \qquad \mathscr{E}(e_0\ \mathsf{or}\ e_1,\sigma) = \mathscr{E}(e_0,\sigma) \cup \mathscr{E}(e_1,\sigma)$$
$$\mathscr{E}(e_0 \oplus e_1,\sigma) = \{v_0 \oplus v_1 \mid v_0 \in \mathscr{E}(e_0,\sigma), v_1 \in \mathscr{E}(e_1,\sigma)\}$$

The small-step operational semantics rules for IMPor are those of IMP given in Figure 3, but now they take into account the non-determinism of $\mathscr{E}(e,\sigma)$. For example, we have:

$$\textsc{Wh1}\ \frac{n \in \mathscr{E}(e,\sigma) \quad n \neq 0}{\langle \mathsf{while}\ e\ \mathsf{do}\ s,\sigma\rangle \to \langle s; \mathsf{while}\ e\ \mathsf{do}\ s,\sigma\rangle} \qquad \textsc{Wh2}\ \frac{0 \in \mathscr{E}(e,\sigma)}{\langle \mathsf{while}\ e\ \mathsf{do}\ s,\sigma\rangle \to \sigma}$$

For IMPor, we write $[\![s]\!]\sigma$ for the *set* of final stores σ' that can be derived from $\langle s,\sigma\rangle$, i.e. $\langle s,\sigma\rangle \to^* \sigma'$. Note that since IMPor is a non-deterministic language $[\![s]\!]\sigma$ may contain more than one final store. Finally, $[\![s]\!] = \bigcup_{\sigma\in Store^{\mathsf{Init}}} [\![s]\!]\sigma$.

3.3 $\overline{\textbf{IMP}}$

A finite set of Boolean variables $\mathbb{F} = \{A_1,\ldots,A_n\}$ describes the set of available *features* in the program family. Each feature may be *enabled* or *disabled* in a particular variant.

A *configuration* k is a truth assignment or a valuation which gives a truth value to each feature, i.e. k is a mapping from $\mathbb{F}$ to $\{\text{true}, \text{false}\}$. If a feature $A \in \mathbb{F}$ is enabled for the configuration k then $k(A) = \text{true}$, otherwise $k(A) = \text{false}$. Any configuration k can also be encoded as a conjunction of literals: $k(A_1) \cdot A_1 \wedge \cdots \wedge k(A_n) \cdot A_n$, where $\text{true} \cdot A = A$ and $\text{false} \cdot A = \neg A$. We write $\mathbb{K}$ for the set of all *valid* configurations defined over $\mathbb{F}$ for a family. The set of valid configurations is typically described by a feature model [23], but in this work we disregard syntactic representations of the set $\mathbb{K}$. Note that $|\mathbb{K}| \leq 2^{|\mathbb{F}|}$, since, in general, not every combination of features yields a *valid* configuration. We define *feature expressions*, denoted *FeatExp*, as the set of well-formed propositional logic formulas over $\mathbb{F}$ generated using the grammar: $\phi ::= \text{true} \,|\, A \in \mathbb{F} \,|\, \neg\phi \,|\, \phi_1 \wedge \phi_2 \,|\, \phi_1 \vee \phi_2$.

Syntax. The programming language $\overline{\text{IMP}}$ is our two-stage extension of IMP (thus, $\overline{\text{IMP}}$ does not contain the 'or' construct). Its abstract syntax includes the same expression and statement productions as IMP, but we add the new compile-time conditional statements for encoding multiple variants of a program. The new statements "#if (ϕ) s #endif" and "#if (ϕ) var x:=n in #endif s" contain a feature expression $\phi \in \mathit{FeatExp}$ as a presence condition, such that only if ϕ is satisfied by a configuration $k \in \mathbb{K}$ then the code between #if and #endif will be included in the variant for k.

s ::= ... | #if (ϕ) s #endif | #if (ϕ) var x:=n in #endif s

Note that only statements and local variable declarations can be compile-time conditionally defined in $\overline{\text{IMP}}$. However, in general "#if" constructs defined on arbitrary language elements could be translated into constructs that respect the appropriate syntactic structure of the language by code duplication [19]. Also note that the C preprocessor uses the following keywords: #if, #ifdef, and #ifndef to start a conditional construct; #elif and #else to create additional branches; and #endif to end a construct. Any of such preprocessor conditional constructs can be desugared and represented only by #if construct we use in this work, e.g. #ifdef (ϕ) s_0 #else s_1 #endif is translated into #if (ϕ) s_0 #endif ; #if ($\neg\phi$) s_1 #endif.

Semantics. The semantics of $\overline{\text{IMP}}$ has two stages: first, given a configuration $k \in \mathbb{K}$ compute an IMP single program without #if-s; second, evaluate the obtained variant using the standard IMP semantics. The first stage is a simple *preprocessor* specified by the projection function π_k mapping an $\overline{\text{IMP}}$ program family into an IMP single program corresponding to the configuration $k \in \mathbb{K}$. The projection π_k copies all basic statements of $\overline{\text{IMP}}$ that are also in IMP, and recursively pre-processes all sub-statements of compound statements. For example, $\pi_k(\text{skip}) = \text{skip}$ and $\pi_k(s_0;s_1) = \pi_k(s_0);\pi_k(s_1)$. The interesting case is "#if (ϕ) s #endif" (resp., #if (ϕ) var x:=n in #endif s) statement, where the statement s (resp., the local variable declaration var x:=n in) is included in the resulting variant iff $k \models \phi$;[I] otherwise it is removed. We have:

$$\pi_k(\texttt{\#if } (\phi)\ s \texttt{ \#endif}) = \begin{cases} \pi_k(s) & \text{if } k \models \phi \\ \texttt{skip} & \text{if } k \not\models \phi \end{cases}$$

$$\pi_k(\texttt{\#if } (\phi)\ \texttt{var x:=}n \texttt{ in } \texttt{ \#endif}\ s) = \begin{cases} \pi_k(\texttt{var x:=}n \texttt{ in } s) & \text{if } k \models \phi \\ \pi_k(s) & \text{if } k \not\models \phi \end{cases}$$

Note that since any configuration $k \in \mathbb{K}$ has only one satisfying truth assignment (values of all features are fixed in k), either $k \models \phi$ or $k \not\models \phi$ for any $\phi \in \mathit{FeatExp}$.

4 Variability-related Transformations

Our aim is to transform an input $\overline{\text{IMP}}$ program family $\bar{s}$ with sets of features $\mathbb{F}$ and configurations $\mathbb{K}$ into an output IMPor program $\overline{s'}$.

In a pre-transformation phase, we first convert each feature $A \in \mathbb{F}$ into the variable A, which is non-deterministically initialized to 0 or 1 (meaning to false or true). Let $\mathbb{F} = \{A_1, \ldots, A_n\}$ be the set of available features in the family $\bar{s}$, then we have the following initialization fragment in the resulting pre-transformed program pre-t($\bar{s}$):

$$\text{pre-t}(\bar{s}) = \texttt{var } A_1 := 0 \texttt{ or } 1 \texttt{ in } \ldots \texttt{var } A_n := 0 \texttt{ or } 1 \texttt{ in } \bar{s}$$

Note that in the initialization we consider all possible combination of values for features (in total $2^{|\mathbb{F}|}$). We will take into account the specific set of configurations $\mathbb{K}$ ($|\mathbb{K}| \leq 2^{|\mathbb{F}|}$) later on, in the transformation phase.

In the following, rewrite rules have the form:

$$\psi \ \vdash \ s \ \rightsquigarrow \ s'$$

meaning that: if the current program family being transformed matches any abstract syntax tree (AST) node of the shape s nested under #if-s with the resulting presence condition that implies $\psi \in \mathit{FeatExp}$ (i.e. in context ψ) then *replace s by s'*. Formally, if we apply the rule $\psi \ \vdash \ s \rightsquigarrow s'$ to a family:

$$\ldots\texttt{\#if } (\phi_1) \ldots \texttt{\#if } (\phi_n) \ldots; s; \ldots \texttt{ \#endif} \ldots \texttt{ \#endif} \ldots$$

where $\phi_1 \wedge \ldots \wedge \phi_n \implies \psi$, then we obtain the transformed program:

$$\ldots\texttt{\#if } (\phi_1) \ldots \texttt{\#if } (\phi_n) \ldots; s'; \ldots \texttt{ \#endif} \ldots \texttt{ \#endif} \ldots$$

We write $\mathit{Rewrite}(\bar{s}, \psi \ \vdash \ s \rightsquigarrow s')$ for the final transformed program $\overline{s'}$ obtained by repeatedly applying the rule $\psi \ \vdash \ s \rightsquigarrow s'$ on $\bar{s}$ and its transformed versions until we reach a point where this rule can not be applied (a fixed point of the rule). Note

[I] Here $\models$ denotes the standard satisfaction relation of propositional logic.

that rules of the form: true $\vdash s \rightsquigarrow s'$, are the most general and can be applied to any statement s no matter whether s is a top-level statement not nested within some #if or s is nested somewhere deep within #if-s. This is due to the fact that any "s" can be written as: "#if (true) s #endif" in the earlier case when s is a top-level statement, and $\phi \implies$ true for any $\phi \in FeatExp$ in the latter case when s is nested within #if-s with presence condition ϕ.

We start with three rules for eliminating configurable variable declarations. They involve duplicating code and variable renaming. The most straightforward way to handle renaming of variables in different contexts is by adding an *environment* δ as a parameter to the statements being transformed. We define an environment $\delta : Var \times FeatExp \to Var$ as a function mapping a given pair of a variable and a feature expression to a variable name. We write $\delta^{\mathsf{fe}}(\mathsf{x}) \subseteq FeatExp$ for the set of all feature expressions ϕ such that $\delta(\mathsf{x}, \phi)$ is defined, i.e. $\delta^{\mathsf{fe}}(\mathsf{x}) = \{\phi \in FeatExp \mid (\mathsf{x}, \phi) \in dom(\delta)\}$. We write (s, δ) to denote the result of simultaneously substituting $\delta(\mathsf{x}, \phi)$ for each occurrence of any variable x in s in the context (presence condition) that implies ϕ.

Conditional variable declaration. This rule transforms a local variable conditionally declared within a given context $\psi \in FeatExp$:

$$\psi \vdash (\texttt{\#if } (\phi) \texttt{ var x:=} n \texttt{ in \#endif} s, \delta) \rightsquigarrow \texttt{var x}_{new}\texttt{:=} n \texttt{ in } (s, \delta[(\mathsf{x}, \phi) \mapsto \mathsf{x}_{new}]) \qquad (1)$$

where x_{new} is a fresh variable name that does not occur as a free variable in s and $range(\delta)$.

Conditional variable use. The second rule handles the case when a local variable is used within a context $\psi \in FeatExp$. There are three cases to consider here.

$$\psi \vdash (\mathsf{y}\texttt{:=}e[\mathsf{x}], \delta) \rightsquigarrow (\mathsf{y}\texttt{:=}e[\delta(\mathsf{x}, \phi)], \delta) \qquad (2.1)$$

if there exists an unique $\phi \in \delta^{\mathsf{fe}}(\mathsf{x})$, such that $\psi \models \phi$. Here $e[\mathsf{x}]$ means that the variable x occurs free in the expression e. The second case is when there are several $\phi_1, \ldots \phi_n \in \delta^{\mathsf{fe}}(\mathsf{x})$, such that $\mathsf{sat}(\phi_1 \wedge \psi), \ldots, \mathsf{sat}(\phi_n \wedge \psi)$:

$$\psi \vdash (\mathsf{y}\texttt{:=}e[\mathsf{x}], \delta) \rightsquigarrow (\texttt{\#if } (\phi_1) \ \mathsf{y}\texttt{:=}e[\delta(\mathsf{x}, \phi_1)] \texttt{ \#endif}; \ldots \texttt{\#if } (\phi_n) \ \mathsf{y}\texttt{:=}e[\delta(\mathsf{x}, \phi_n)] \texttt{ \#endif}, \delta) \qquad (2.2)$$

Otherwise, meaning that for all $\phi \in \delta^{\mathsf{fe}}(\mathsf{x})$ it follows that $\mathsf{unsat}(\phi \wedge \psi)$, we have:

$$\psi \vdash (\mathsf{y}\texttt{:=}e[\mathsf{x}], \delta) \rightsquigarrow (\mathsf{y}\texttt{:=}e[\mathsf{x}], \delta) \qquad (2.3)$$

Conditional variable define. The third rule applies when a local variable is assigned to within a context $\psi \in FeatExp$. There are three cases to consider here as well.

$$\psi \vdash (\mathsf{x}\texttt{:=}e, \delta) \rightsquigarrow (\delta(\mathsf{x}, \phi)\texttt{:=}e), \delta \qquad (3.1)$$

when there exists an unique $\phi \in \delta^{\mathsf{fe}}(\mathsf{x})$, such that $\psi \models \phi$.

$$\psi \vdash (\mathsf{x}\texttt{:=}e, \delta) \rightsquigarrow (\texttt{\#if } (\phi_1) \ \delta(\mathsf{x}, \phi_1)\texttt{:=}e \texttt{ \#endif}; \ldots \texttt{\#if } (\phi_n) \ \delta(\mathsf{x}, \phi_n)\texttt{:=}e \texttt{ \#endif}), \delta \qquad (3.2)$$

when there are $\phi_1, \ldots \phi_n \in \delta^{\mathsf{fe}}(\mathsf{x})$, such that $\mathsf{sat}(\phi_1 \wedge \psi), \ldots, \mathsf{sat}(\phi_n \wedge \psi)$. Otherwise,

$$\psi \vdash (\mathsf{x}{:=}e, \delta) \rightsquigarrow (\mathsf{x}{:=}e, \delta) \tag{3.3}$$

After applying the above three rules, all local variable declarations that are conditionally defined ($\#\mathsf{if}\ (\phi)\ \mathsf{var}\ \mathsf{x}{:=}n\ \mathsf{in}\ \#\mathsf{endif} s$) are resolved. The transformed program contains only #if-s where statements are conditionally defined.

Conditional statement elimination. The set of valid configurations $\mathbb{K}$ can be equated to a propositional formula [4], say $\kappa \in \mathit{FeatExp}$, such that $\kappa = \vee_{k \in \mathbb{K}} k$. The last rule simply replaces #if-s with ordinary if-s whose guards are strengthen with the feature model κ, thus taking into account only valid configurations $\mathbb{K}$ of a family.

$$\psi \vdash \#\mathsf{if}\ (\phi)\ s\ \#\mathsf{endif} \rightsquigarrow \mathsf{if}\ (\phi \wedge \kappa)\ \mathsf{then}\ s\ \mathsf{else\ skip} \tag{4}$$

Note that we omit to write the environment δ in rules that do not use it explicitly (e.g. rules (4), (5)). Let $\delta_0 = [\,]$ be the empty environment. Let $\mathit{Rewrite}^{\mathsf{preserve}}(\mathsf{pre\text{-}t}(\overline{s}), \delta_0)$ be the final transformed program $\overline{s'}$ obtained from the pre-transformed program $\mathsf{pre\text{-}t}(\overline{s})$ by applying the rules (1)–(3), and then the rule (4). The following result shows that the set of final answers from terminating computations of $\overline{s'}$ coincides with the union of final answers from terminating computations of all variants from $\overline{s}$.

Theorem 1. *Let $\overline{s'} = \mathit{Rewrite}^{\mathsf{preserve}}(\mathsf{pre\text{-}t}(\overline{s}), \delta_0)$. We have: $[\![\overline{s'}]\!] = \bigcup_{k \in \mathbb{K}} [\![\pi_k(\overline{s})]\!]$.*

Proof. First, we show that $\mathit{Rewrite}^{\mathsf{preserve}}(\mathsf{pre\text{-}t}(\overline{s}), \delta_0)$ terminates. This is due to the fact the number of if-s in pre-t($\overline{s}$ is finite, and by iteratively applying rules (1)–(3) we eliminate all $\#\mathsf{if}\ (\phi)\ \mathsf{var}\ \mathsf{x}{:=}n\ \mathsf{in}\ \#\mathsf{endif} s$; whereas by applying rule (4) afterwards we eliminate all $\#\mathsf{if}\ (\phi)\ s\ \#\mathsf{endif}$. Subsequently, for each rule (1)–(3) and (4), the above result can be proved by structural induction. □

We now present an optimization rule, which is applied before the rules (1)–(4) for eliminating if-s. The correctness of our transformation does not depend on it, but we can use it for achieving faster convergence and smaller transformed programs. In our implementation, we use many such optimization rules.

Guard inlining. This rule collapses two sequentially composed #if-s with mutually exclusive presence conditions ϕ_0 and ϕ_1 (i.e. $\phi_0 \wedge \phi_1 \equiv \mathsf{false}$) that conditionally enable the same statement s into one #if that conditionally enables s:

$$\psi \vdash \#\mathsf{if}\ (\phi_0)\ s\ \#\mathsf{endif};\ \#\mathsf{if}\ (\phi_1)\ s\ \#\mathsf{endif} \rightsquigarrow \#\mathsf{if}\ (\phi_0 \vee \phi_1)\ s\ \#\mathsf{endif} \tag{5}$$

Example 2. *We present the transformation rules on a program family with $\mathbb{F} = \{A, B\}$ and $\mathbb{K} = \{A \wedge B, A \wedge \neg B, \neg A \wedge B, \neg A \wedge \neg B\}$.*

$$\begin{aligned}
&\big(\#\mathit{if}\ (A)\ \mathit{var}\ x{:=}2\ \mathit{in}\ \#\mathit{endif}\#\mathit{if}\ (\neg A)\ \mathit{var}\ x{:=}5\ \mathit{in}\ \#\mathit{endif}\#\mathit{if}\ (B)\ y{:=}x\ \#\mathit{endif}, \delta_0\big)\\
&\overset{(1)}{\rightsquigarrow}\ \mathit{var}\ x_1{:=}2\ \mathit{in}\ \big(\#\mathit{if}\ (\neg A)\ \mathit{var}\ x{:=}5\ \mathit{in}\ \#\mathit{endif}\#\mathit{if}\ (B)\ y{:=}x\ \#\mathit{endif}, [(x,A) \mapsto x_1]\big)\\
&\overset{(1)}{\rightsquigarrow}\ \mathit{var}\ x_1{:=}2\ \mathit{in}\ \mathit{var}\ x_2{:=}5\ \mathit{in}\ \big(\#\mathit{if}\ (B)\ y{:=}x\ \#\mathit{endif}, [(x,A) \mapsto x_1, (x,\neg A) \mapsto x_2]\big)\\
&\overset{(2.2)}{\rightsquigarrow}\ \mathit{var}\ x_1{:=}2\ \mathit{in}\ \mathit{var}\ x_2{:=}5\ \mathit{in}\ \#\mathit{if}\ (B)\ \#\mathit{if}\ (A)\ y{:=}x_1;\ \#\mathit{endif}\#\mathit{if}\ (\neg A)\ y{:=}x_2\ \#\mathit{endif}\ \#\mathit{endif}\\
&\overset{(4)}{\rightsquigarrow}\ \mathit{var}\ x_1{:=}2\ \mathit{in}\ \mathit{var}\ x_2{:=}5\ \mathit{in}\ \mathit{if}(B)\,\mathit{then}\,\mathit{if}\ (A)\ \mathit{then}\ y{:=}x_1\ \mathit{else\ skip};\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathit{if}\ (\neg A)\ \mathit{then}\ y{:=}x_2\ \mathit{else\ skip};\ \mathit{else}\,\mathit{skip}
\end{aligned}$$

5 Implementation

We have developed a tool, called C Reconfigurator, which implements variability-related transformations for the C language. All transformations are implemented using Xtend [2] . The C Reconfigurator tool is available from: https://github.com/models-team/c-reconfigurator. It calls variability-aware parser SuperC [20] to parse code with preprocessor annotations, which uses Binary Decision Diagrams (BDD's) for encoding feature expressions and for decisions during the parsing process. SuperC returns an AST with variability, in which variability is reflected with choice nodes over feature expressions. In particular, a choice node is a node with two children, such that the left child of the choice node is included in the result of those configurations for which the given feature expression is satisfied; otherwise the right child of the choice node is included in the parsing result when the feature expression is not satisfied. We apply our variability-related transformation rules as described in Section 4 on AST's with variability obtaining an ordinary AST, which is subsequently translated into a single C program. Since IMP is a subset of C, all rewritings described in Section 4 transfer directly to C. We now discuss how a selection of other interesting C constructs, which are not present in IMP, are handled by our tool.

Variables declared with optional types are very common in C. For example, we have x-bit integers on x-bit machines. We handle them in a similar way as configurable variable declarations in rules (1)–(3). First, we rename and duplicate the variable declaration, then at each point where the variable is used we transform the code such that the used variable refers to the correct configuration name. For example,

#if(A) int #else float #endif x=0;
x = x+1;

will be transformed into:

int $\mathsf{x}_1 = 0$; float $\mathsf{x}_2 = 0$;
#if(A) $\mathsf{x}_1 = \mathsf{x}_1$+1; #else $\mathsf{x}_2 = \mathsf{x}_2$+1; #endif

Note that if optional local variables are initialized by non-constant expressions, then we split their transformation into two parts: declaration which is performed by renaming and duplication, followed by initialization where all optional variables refer to the correct configuration.

Optional (configurable defined) functions are important since all statements in C are inside some function. If conditionally defined code occurs in the function body, then it will be transformed using the corresponding rules. For example,

int f(int x){return #if(A) x++ #else 0 #endif; }

will be transformed into:

int f(int x){return A? x++ : 0; }

[2] http://www.eclipse.org/xtend/.

If the function signature is configurable, then we use renaming plus duplication as in rules (1)–(3) for handling configurable variable declarations. For example, the code:

int f (#if (A) int #else float #endif x) $\{\ldots\}$
$\ldots f(5)\ldots$

will be transformed into:

int f_1(int x) $\{\ldots\}$
int f_2(float x) $\{\ldots\}$
$\ldots$#if (A) $f_1(5)$ #else $f_2(5)$ #endif$\ldots$

Arrays with optional size are also possible in real-world C programs. They usually emerge via constant macros with conditional definitions. For example, the code

int a[#if (A) 10 #else 15 #endif];
$\mathsf{a}[5] = 0$;

will be transformed into:

int $\mathsf{a}_1[10]$; int $\mathsf{a}_2[15]$;
#if (A) $\mathsf{a}_1[5] = 0$; #else $\mathsf{a}_2[5] = 0$; #endif

All other variability patterns that we met in our examples, such as configurable fields in struct-s and pointers, are also handled similarly: first by using renaming and duplication, then by modifying all references to the given pattern such that the use always refers to the correct definition. Consider the following code with pointers:

int $\mathsf{a} = 10$; int$*$ $\mathsf{p} = \&\mathsf{a}$; #if (A) $\mathsf{p} = \mathit{null}$; #endif $(*\mathsf{p})$++

will be transformed into:

int $\mathsf{a} = 10$; int$*$ $\mathsf{p} = \&\mathsf{a}$; if (A) $\mathsf{p} = \mathit{null}$; $(*\mathsf{p})$++

Hence, we obtain a variability bug whenever the feature A is enabled.

Remark. We can see that most of the variability patterns are handled using renaming plus duplication. In the worst case, this may cause exponential growth of the transformed program in the number of used features. However, in practice this does not happen often (see Table 3 for some data from real files). Namely, variability patterns usually depend on a few features, so only a few new definitions are used. Also we apply several optimization rules, which eliminate all definitions that do not correspond to a valid configuration. Finally, the evaluation results in Section 6 show that the analysis time for such transformed programs is comparable to single programs. This is due to the fact that transformed programs are not increased significantly and the analysis tools we use (Frama-C, Clang, LLBMC) are very optimized and mature.

6 Evaluation

We evaluate our reconfiguration technique based on variability transformations and single-program verification oracles on several real-world C case studies. The evaluation aims to show that we can use state-of-the-art single-program verification tools to verify realistic C program families using variability-related transformations. To do so, we ask the following research questions:

- How precise is our technique? **(RQ1)**
- How efficient is the verification oracle to identify variability bugs after transforming the code using our technique? **(RQ2)**

In particular, we want to reproduce the variability bugs reported in [1, 28] using various verification oracles applied on transformed programs, which are obtained using our tool. We use Frama-C [27], Clang [9] and LLBMC [30] as our verification oracles. Frama-C is a framework for modular static (dataflow) analysis of C programs. The Clang project includes the Clang compiler front-end and the Clang static analyzer for several programming languages, including C. LLBMC (the low-level bounded model checker) is a software model checking tool for finding bugs in C programs.

6.1 Subject Files and Experimental Setup

All transformations are applied using the C Reconfigurator tool as described in Section 5. We investigate precision and performance in finding real variability bugs extracted from three benchmarks: Linux, BusyBox and Libssh. In particular, we use simplified bugs from the VDBb [3] database that are found in the Linux kernel files [1] and in BusyBox. Abal et al. [1] created a simplified version of a program for each bug they found by capturing the same essential behavior (and the same problem) as in the original bug. Simplified bugs are independent of the kernel code and the corresponding programs were derived systematically from the error trace. In addition, we use real variability bugs from Libssh provided by Medeiros et al. [28].

Table 1 presents the characteristics of the subject files we analyzed in our empirical study. We list: the file id, bug type, number of features ($|\mathbb{F}|$), number of valid configurations ($|\mathbb{K}|$), lines of code, the size in KB of the files before (with #ifdef-s) and after (without #ifdef-s) our transformations, and commit hash (clickable) for each project. This collection consists of a diverse set of bug types such as null pointer dereferences, buffer overflow, and uninitialized variable. In total, we have 11 distinct kinds of bugs. The number of features per file varies from one to seven. In addition, the number of lines of code ranges from 12 to 165 for the simplified files (from VBDb), and from 1404 to 2959 for real files (from Libssh). After the transformation, the biggest increase in size of almost 8 times can be observed for file id 7. This is due to the fact that this file has seven different features and several variability patterns that depend on them. In most of the other cases the size increase is not very big.

[3] http://VBDb.itu.dk.

File ID	Bug type	$\lvert\mathbb{F}\rvert$	$\lvert\mathbb{K}\rvert$	LOC	Size KB		Hash
					before	after	
VBDb Linux files							
1	null pointer deref.	5	24	165	2.9	4.3	76baeeb
2	null pointer deref.	3	6	112	1.9	2.5	f7ab9b4
3	null pointer deref.	4	8	55	0.9	1.0	ee3f34e
4	null pointer deref.	3	6	34	0.5	0.6	6252547
5	buffer overflow	1	2	58	1.0	1.2	8c82962
6	buffer overflow	1	2	33	0.6	0.7	60e233a
7	read out of bounds	7	63	69	1.1	8.4	0f8f809
8	uninitialized var.	2	4	54	0.8	1.0	7acf6cd
9	uninitialized var.	1	2	54	1.0	1.1	bc8cec0
10	uninitialized var.	1	2	53	0.8	1.0	30e0532
11	uninitialized var.	2	4	38	0.9	1.2	1c17e4d
12	uninitialized var.	2	4	26	0.3	0.5	e39363a
13	undefined symbol	4	14	25	0.4	0.6	7c6048b
14	undefined symbol	2	4	20	0.3	0.5	2f02c15
15	undefined symbol	2	4	20	0.3	0.5	6515e48
16	undefined symbol	2	4	19	0.3	0.5	242f1a3
17	undeclared identifier	3	8	37	0.6	1.0	6651791
18	undeclared identifier	2	4	20	0.3	0.4	f48ec1d
19	wrong # of args	1	2	12	0.2	0.4	e67bc51
20	multiple funct. defs	2	4	21	0.3	0.8	e68bb91
21	dead code	1	2	19	0.2	0.3	809e660
22	incompatible type	2	4	27	0.4	0.7	d6c7e11
23	assertion violation	2	4	79	1.5	1.8	63878ac
24	assertion violation	2	4	75	1.1	1.2	657e964
25	assertion violation	2	4	41	0.6	0.7	0988c4c
VBDb BusyBox files							
26	null pointer deref.	1	2	28	0.4	0.7	199501f
27	null pointer deref.	2	4	24	0.4	0.6	1b487ea
28	uninitialized var.	2	4	28	0.4	0.7	b273d66
29	undefined symbol	1	2	42	0.8	0.9	cf1f2ac
30	undefined symbol	2	4	27	0.4	0.6	ebee301
31	undeclared identifier	1	2	35	0.5	0.8	5275b1e
32	undeclared identifier	1	2	19	0.3	0.4	b7ebc61
33	incompatible type	3	8	46	0.9	1.5	5cd6461
Real Libssh files							
34	null pointer deref.	6	48	1404	34.8	32.6	0a4ea19
35	null pointer deref.	4	4	1428	44.1	31.9	fadbe80
36	uninitialized var.	3	4	2959	72.4	77.6	2a10019

Table 1 Characteristics of the benchmark files.

ID	Frama-C				
	buggy variant		reconfigured		all
	y/n	time	y/n	time	time
VBDb Linux files					
1	✓	218	✓	235	5602
2	✓	220	✓	225	1394
3	✓	215	x	236	1918
4	✓	218	✓	224	1379
5	✓	218	✓	227	488
6	✓	213	✓	227	463
7	✓	218	✓	225	14381
8	✓	241	✓	250	918
9	✓	224	✓	230	462
10	✓	216	inc	224	460
11	✓	234	✓	224	917
12	✓	216	inc	227	914
13	✓	239	✓	248	3194
14	✓	237	✓	244	905
15	✓	224	✓	248	906
16	✓	213	✓	222	910
17	✓	216	✓	230	3823
18	✓	210	✓	224	901
19	✓	210	✓	224	452
20	✓	213	x	228	907
21	✓	239	x	240	458
VBDb BusyBox files					
26	✓	230	✓	234	484
27	✓	224	✓	234	959
28	✓	237	inc	237	957
29	✓	230	✓	236	481
30	✓	231	✓	228	968
31	✓	220	✓	228	486
32	✓	216	✓	224	477

(a) VBDb files using Frama-C.

ID	Clang/LLBMC				
	buggy variant		reconfigured		all
	yes/no	time	yes/no	time	time
VBDb Linux files					
22	✓	21	✓	23	91
23	✓	4	✓	10	10
24	✓	3	✓	7	11
25	✓	3	✓	5	8
VBDb BusyBox files					
33	✓	27	✓	31	222

(b) VBDb files using Clang (files 22 and 33) and LLBMC (files 23, 24, and 25).

ID	Clang/LLBMC				
	buggy variant		reconfigured		all
	yes/no	time	yes/no	time	time
34	✓	1526	✓	1702	17029
35	✓	1591	✓	1804	5917
36	✓	112	✓	144	448

(c) Libssh files using Clang (file 36) and LLBMC (files 34 and 35).

■ **Table 2** Verification results for the benchmark files. Times in milliseconds (ms).

All experiments were executed on a Kubuntu VM (64bit, 4 CPUs), Intel®Core^{TM} i7-3720QM CPU running at 2.6GHz with 12GB RAM memory. The performance numbers reported constitute the median runtime of fifty independent executions.

6.2 Results

We now present the results of our empirical study and discuss the implications. All experiment materials are available online at https://github.com/models-team/c-reconfigurator-test. Before we proceed, we stress that we only evaluate bugs that are detectable by the verification tools on the erroneous variant code.

Simplified files. Table 2a shows the results of verifying our benchmark files which contain known bugs by using Frama-C. The table has three main columns: buggy variant, reconfigured, and all that depict the tool results on the buggy variant code, on the reconfigured program family code, and on all valid variants from $\mathbb{K}$ analyzed one by one (in a brute force fashion), respectively. Each checkmark (✓) means that the same bug was found in both the buggy variant and reconfigured program by the verification tool. Otherwise, the result is either *x*—bug not found in the reconfigured program, or *inc*—inconclusive which means that Frama-C was able to detect a bug in the reconfigured program that is different from the bug in the product variant. In the case of brute force approach (all), we consider the analyses times of all valid variants regardless of whether they contain a bug or not.

In terms of precision, our C Reconfigurator tool transforms the family code by preserving the erroneous traces from the buggy variant in most cases. For instance, Frama-C could detect 22 (78%) bugs from the simplified benchmark files (28 in total) after reconfiguring the files using our tool. Besides that, the C Reconfigurator preserves a variety of bug types such as buffer overflow and uninitialized variable. Still, for different types of bugs the success rate depends on the tool which may or may not detect them. For example, our technique is able to transform a file containing a memory leak error, but Frama-C does not have any analysis to identify it.

In three specific cases (cf. file ids 10, 12 and 28), Frama-C did not report the original bug as an error, but it did detect that some variable might be uninitialized in some conditions. This happens because Frama-C performs a *may* value analysis for finding uninitialized variables. A *may* analysis describes information that may possibly be true along one path to the given program point and, thus in our case, computes a superset of all uninitialized variables in all variants. So the reported variable may not match with the one in the buggy variant. We marked these three cases as *inc*—inconclusive in the table. Still the verification oracle reports that there might be an error in the reconfigured code.

In addition, the verification tool could not identify the required bug in the reconfigured file in three cases (cf. file ids 3, 20 and 21). For example, file 21 contains dead code, which is a function (`do_sect_fault()`) that is never called when feature `ARM` is enabled (see the code snippet in Fig. 4, left column). The C Reconfigurator transforms the code by changing the `#ifdef` into ordinary `if` condition, making the function available for the transformed single program (i.e., the function is not dead any more), as shown in the code snippet in Fig. 4 (right column). The other two cases are similar to this one in the sense that the C Reconfigurator makes feature code explicit to the entire program family.

Generally speaking, if one variant does not use a variable/function, but another does, then the reconfigured code will use the variable/function and the error will be hidden (like in the example above). This happens due to the limitations of variability encoding, especially because we cannot preprocess the reconfigured code to filter out the irrelevant features for a particular variant. In a reconfigured code, all variants are encoded as a single program (see Section 6.4 for more discussion).

We now consider the remaining simplified files. We use Clang and LLBMC to analyze only the other types of bugs (incompatible type and assertion violation) that

```
int do_sect_fault(){
   return 0;
}
int main(){
   #ifndef ARM
      do_sect_fault();
   #endif
   return 0;
}
```

```
int do_sect_fault(){
 return 0;
}

int main(){
    if (! ARM)
       do_sect_fault();
   return 0;
}
```

Figure 4 File 21 - Before (left) and after (right) our transformations

Frama-C cannot handle. We treat Clang/LLBMC as one verification oracle, since we first need to compile and emit llvm code with Clang in order to analyze it using LLBMC. So, we do not make difference in reporting whether the bug was found by Clang during the compilation or afterwards by LLBMC.

Table 2b, similarly to Table 2a, shows the results of verifying both the buggy variant and the reconfigured code using Clang and LLBMC. We also report the analysis time of the brute force approach in the column all. As we can see, all bugs were found by Clang/LLBMC in the reconfigured version. We can thus confirm that our C Reconfigurator tool transforms the family code by preserving the erroneous traces from the buggy variant. We are now ready to answer RQ1 on the precision of our technique. Based on analyzing 33 simplified variability bugs from Linux and BusyBox, we find that:

> Answer RQ1 (precision). The C Reconfigurator enables single-program verification tools such as Frama-C, Clang, and LLBMC to **successfully** detect *most* of the simplified variability bugs on the reconfigured code, obtained from the Linux and BusyBox benchmark files.

We now turn to evidence regarding research question RQ2 (performance). We evaluate performance of the verification tools to identify the given variability bugs. Tables 2a and 2b show time needed for the verification tools to analyze the buggy variant code (buggy variant column) and the reconfigured program family code (reconfigured column). We can see that the analysis times in both cases are similar although reconfigured code is bigger in size. In fact, Frama-C takes less than half a second to analyze each file regardless whether it is a variant or a reconfigured file. For instance, Frama-C analyzes file 1 in 218 and 235 milliseconds on the variant code and on the reconfigured program family code, respectively. Recall that file 1 contains a null pointer dereference and has five features. If we apply the brute force approach (all column), which analyzes all variants individually one by one, to this file using Frama-C it takes 5,602 ms, since the number of configurations is 24. In this way, we obtain significant speed-up to verify the program family using our approach. We also obtain similar results in terms of performance using Clang/LLBMC (see Tables 2b and 2c). In general, the performance of analyzing a reconfigured code is similar to analyzing only one variant, which gives us a speed-up proportional to the number of valid variants of a program family. Overall, we answer the second research question (RQ2) by observing that:

> ANSWER RQ2 (PERFORMANCE). The C RECONFIGURATOR speeds-up the family-based analysis via single-program verification tools, so that we can **efficiently** detect simplified variability bugs on the reconfigured code, obtained from the VBDb benchmark.

Real files. We now consider real files to confirm our previous observations with respect to precision and performance. Table 2c presents the results of analyzing three real files from the Libssh project using CLANG and LLBMC.[4] These files contain two types of bugs: null pointer dereference and uninitialized variable. Each file has at least three distinct features.

We can see that our C RECONFIGURATOR transforms the family code by preserving the erroneous traces from the buggy variant even for complex and large files. In fact, the verification tool (CLANG/LLBMC) found the same bug (from the buggy variant code) on the reconfigured code in all three cases. From this preliminary evidence, we thus confirm that our technique enables single-program verification oracles to successfully detect variability bugs on the reconfigured code, obtained from complex and real files.

Regarding performance, we can still see the similarity in verifying a variant code and a reconfigured one. For example, CLANG/LLBMC took 1,5 sec to analyze file 34 in the single variant version, whereas in the reconfigured version, the tool analyzed it in 1,7 sec. We can also observe a speed-up of the family-based analysis using the C RECONFIGURATOR and single-program verification tools by a factor of the number of valid variants compared to the brute force approach. We conclude that:

> SUMMARY. All single-program verification tools (FRAMA-C, CLANG, LLBMC) detect **successfully** and **efficiently** most of the variability bugs on the reconfigured code as well as on the single variant code.

6.3 Threats to Validity

Internal validity. Verifying semantics preservation in a complex transformation is a very hard problem [22, 2]. We manually verified the correctness of the C RECONFIGURATOR on the simplified VBDb files by comparing the original and the reconfigured files side-by-side, which leaves space for human error. For the larger real files we were not able to determine if the C RECONFIGURATOR preserved semantics for all variants on the entire file due to the complex configuration space, but instead we focused on the functions involved in producing/reproducing the bug. We mitigate this threat by relying on the results of our evaluation which show the effectiveness of conventional single-program analysis tools to identify the same bugs in the reconfigured code version as in the buggy single varaints.

[4] We do not report results from FRAMA-C on the real files because FRAMA-C could not handle them.

External validity. From our preliminary evaluation, we show that our technique transforms the program family code by preserving the erroneous traces from the buggy variant. However, we acknowledge that our transformations were not tested under the entirety of the C language, but only on the subset used in the VBDb and Libssh files presented here. The C Reconfigurator though can be extended with extra rules to deal with other cases that we did not face in our benchmark files. Worst case exponential growth of transformed programs can happen, even though we have not observed it in our subject files.

6.4 Discussion

The main limitation of our transformation based approach is that we may not obtain conclusive results for all individual variants, thus losing some precision. This is due to the fact that our transformed program contains all possible paths that may occur in any variant. However, the precision loss depends on the particular analysis we use.

Consider the case of model checking. Since (single-system) model checkers stop once a single counter-example is found in the model, we can use our approach to find a variability bug which occurs in some subset of valid variants but we will not be able to report conclusive results (whether the given property is satisfied or not) for the rest of the valid variants. To overcome this issue, we may repeat our technique on the remaining variants for which no conclusive results were reported in the previous iteration.

Consider the case of *must* dataflow analysis (e.g., available expressions, very busy expressions). In this case, the result in a given program point contains only the common results found on all execution paths to that point. Thus, the analysis result for the transformed program will contain only the results that occur in all variants. For example, for available expressions analysis we may obtain less available expressions than there are in any single variant. The available expression analysis determines which expressions must have already been computed, and not later modified, on all paths to a program point [32]. This information can be used to avoid re-computation of an expression. Consider the program family:

$$\mathsf{x} := a + b; \mathsf{while}\ (\mathsf{y} > a + b)\ \mathsf{do}\ \{\texttt{\#ifdef}\ (A)\ \mathsf{y} := \mathsf{y} - 1\ \texttt{\#else}\ a := a + 1\ \texttt{\#endif}\}$$

The expression $a + b$ is available at the guard of the while loop for variants satisfying A, so it needs not be re-computed for them. However, in the transformed program we have paths from all variants, so the expression $a + b$ is modified by the assignment $a := a + 1$ in a path coming from variants $\neg A$. Therefore, the analyzer will not report this expression as available at the guard of the loop for the transformed program.

Consider the case of *may* dataflow analysis (e.g., reaching definitions, live variables, uninitialized variables). In this case, the result in a given program point contains the results found on at least one execution path to that point. Thus, the analysis result for the transformed program will contain all results that occur in at least one variant. For example, for live variables analysis, we may obtain more live variables than there are in any single variant. The live variables analysis determines which variables may be live at a program point, that is there is a path from the program point to a use of the variable that does not redefine it [32]. This information can be used as a basis for dead code elimination. If a variable is not live at the exit from an assignment to the variable, then that assignment can be eliminated. Consider the program family:

$$\mathtt{x} := 5; \mathtt{y} := 1; \texttt{\#ifdef}\ (A)\ \mathtt{x} := 1\ \texttt{\#else}\ \mathtt{x} := \mathtt{x} + 1\ \texttt{\#endif}$$

The variable x is not live at the exit from the first assignment x := 5 for variants satisfying *A*. Therefore, the assignment x := 5 is redundant for those variants. However, x is live for ¬*A* variants, so it will be live after the first assignment for the transformed program as well. Thus, we cannot eliminate this assignment in the transformed program. This is also the reason why Frama-C does not identify the variability bug for files 3, 20 and 21.

7 Related work

Recently, formal analysis and verification of program families have been a topic of considerable research. The challenge is to develop efficient techniques that work at the level of program families, rather than the level of single programs. There are two main approaches to address this issue: (1) to develop dedicated variability-aware (family-based) techniques and tools; (2) to use specific simulators and encodings which transform program families into single programs that can be analyzed by the standard single-program verification tools. The two approaches have different strengths and weaknesses. The advantage of (1) is that precise (conclusive) results are reported for every variant, but the disadvantage is that their implementation can be tedious and labor intensive. On the other hand, the approaches based on (2) re-use existing tools from single-program world, but some precision may be lost when interpreting the obtained results.

Specifically designed variability-aware techniques. Various lifted techniques have been proposed which lift existing single-program verification techniques to work on the level of program families. This includes lifted syntax checking [25, 20], lifted type checking [24, 8], lifted static analysis [7, 6, 31], lifted model checking [10, 14], etc. TypeChef [25] and SuperC [20] are variability-aware parsers, which can parse languages with preprocessor annotations. The results are ASTs with variability nodes. The difference between these two approaches is that feature expressions are represented as formulae in TypeChef, and as BDD's in SuperC. TypeChef has also implemented some variability-aware dataflow analyses. Several approaches have been proposed for type checking program families directly. In particular, lifted type checking for Featherweight Java was presented in [24], whereas variational lambda calculus was studied in [8]. Lifted model checking for verifying variability intensive systems has been introduced in [10]. SNIP, a specifically designed family-based model checker, is implemented for efficient verification of temporal properties of such systems. The input language to SNIP is fPromela, which represents a variability-aware extension of the known Promela language for the (single-system) SPIN model checker [21]. fPromela uses an #ifdef-like statement for encoding multiple variants, which represents a nondeterministic "if" statement guarded by features expressions that are used to specify what system parts are included (resp., excluded) for which variants. An approach for lifted software model checking using game semantics has been introduced in [14]. It verifies safety of #ifdef-based second-order program families containing undefined components, which are compactly represented using symbolic game semantics models [13, 12]. Brabrand et al. [7] and Midtgaard et al. [31] show how to lift any single-program dataflow analysis from the monotone framework to work on the level of program families. The obtained lifted dataflow analyses are much faster than ones based on the naive variant-by-variant approach that generates and analyzes all variants one by one. Another efficient implementation of lifted analysis formulated within the IFDS framework for inter-procedural distributive environments has been proposed in SPL$^{\text{LIFT}}$ [6]. In order to speed-up the lifted verification techniques, variability abstractions have been introduced in [17, 18, 15, 16]. They tame the exponential blowup caused by the large number of features and variants in a program family. In this way, variability abstractions enable deliberate trading of precision for speed in the context of lifted (monotone) data-flow analysis [17, 18] and lifted model checking [15, 16].

Lifting by simulation. Variability encoding [37] and configuration lifting [33] are based on generating a product-line *simulator* which simulates the behaviour of all products in the product line. Then, an existing off-the-shelf single-program analyzer is used to verify the generated product-line simulator, which represents a single program. The work in [37] defines variability encoding on the top of TypeChef parser for C and Java program families. They have applied the results of variability encoding to testing [26], model checking [3], and deductive verification [36]. Compared to [37], our approach has the following distinguished characteristics. C Reconfigurator is aimed at transforming C program families and uses SuperC as a back-end tool. We show transformation rules and their correctness with respect to a minimal C-like imperative (state-based) language, whereas in [37] the rules and their correctness is shown with respect to Featherweight Java. C is a language much wider used in industry for variability than (Featherweight)Java. Also, we do not have to rely on object-oriented encodings to make the variability-transformations work. We evaluate our approach with several state-of-the-art single-program verification tools for finding real variability bugs on real-world C programs (both on large and sanitized files). The academic examples (e-mail, elevator, mine-pump) considered by Apel et al. [3] are considerably smaller than those presented here; and they are focussed on verifying specific class of bugs: undesired feature interactions (using CPAchecker [5]), whereas we consider here various types of more severe bugs that occur in practice. In this way, the external validity of our experiments is considerably broader. Yet another difference is that the work in [3] considers product lines implemented using compositional approaches, where all features are modeled as separate and composable units. In contrast, we consider here annotative product lines based on #ifdef-s, which is a common way of implementing variability in industry.

8 Conclusion

We have proposed variability-related transformations to translate program families into single programs without variability. The transformed programs can then be effectively analyzed using various single-program analyzers. The evaluation confirms that some interesting variability bugs can be found in real-world C programs in this way. As a future work, we plan to extend our evaluation and consider more verification oracles as well as different practical case studies. We derive several observations from the attempt to verify, analyze, and find bugs in realistic C programs. We hope that our technique will be useful for future builders of analysis tools.

References

[1] Iago Abal, Claus Brabrand, and Andrzej Wasowski. 42 variability bugs in the linux kernel: a qualitative analysis. In *ACM/IEEE International Conference on Automated Software Engineering, ASE '14*, pages 421–432. ACM, 2014. URL: http://doi.acm.org/10.1145/2642937.2642990, doi:10.1145/2642937.2642990.

[2] Ahmad Salim Al-Sibahi, Aleksandar S. Dimovski, and Andrzej Wasowski. Symbolic execution of high-level transformations. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering, SLE 2016*, pages 207–220. ACM, 2016.

[3] Sven Apel, Alexander von Rhein, Philipp Wendler, Armin Größlinger, and Dirk Beyer. Strategies for product-line verification: case studies and experiments. In *35th Intern. Conference on Software Engineering, ICSE '13*, pages 482–491, 2013.

[4] Don Batory. Feature models, grammars, and propositional formulas. In *9th International Software Product Lines Conference, SPLC '05*, volume 3714 of *LNCS*, pages 7–20. Springer-Verlag, 2005.

[5] Dirk Beyer and M. Erkan Keremoglu. Cpachecker: A tool for configurable software verification. In *Computer Aided Verification - 23rd International Conference, CAV 2011. Proceedings*, volume 6806 of *LNCS*, pages 184–190, 2011. URL: http://dx.doi.org/10.1007/978-3-642-22110-1_16, doi:10.1007/978-3-642-22110-1_16.

[6] Eric Bodden, Társis Tolêdo, Márcio Ribeiro, Claus Brabrand, Paulo Borba, and Mira Mezini. Spl$^{\text{lift}}$: Statically analyzing software product lines in minutes instead of years. In *ACM SIGPLAN Conference on PLDI '13*, pages 355–364, 2013.

[7] Claus Brabrand, Márcio Ribeiro, Társis Tolêdo, Johnni Winther, and Paulo Borba. Intraprocedural dataflow analysis for software product lines. *Transactions on Aspect-Oriented Software Development*, 10:73–108, 2013.

[8] Sheng Chen, Martin Erwig, and Eric Walkingshaw. An error-tolerant type system for variational lambda calculus. In *ACM SIGPLAN International Conference on Functional Programming, ICFP'12*, pages 29–40, 2012. URL: http://doi.acm.org/10.1145/2364527.2364535, doi:10.1145/2364527.2364535.

[9] Clang. Clang static analyzer. Clang: a C language family frontend for LLVM. URL: http://clang-analyzer.llvm.org/.

[10] Andreas Classen, Maxime Cordy, Pierre-Yves Schobbens, Patrick Heymans, Axel Legay, and Jean-François Raskin. Featured transition systems: Foundations for verifying variability-intensive systems and their application to LTL model checking. *IEEE Trans. Software Eng.*, 39(8):1069–1089, 2013. URL: http://doi.ieeecomputersociety.org/10.1109/TSE.2012.86, doi:10.1109/TSE.2012.86.

[11] Paul Clements and Linda Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley, 2001.

[12] Aleksandar Dimovski and Ranko Lazic. Compositional software verification based on game semantics and process algebra. *STTT*, 9(1):37–51, 2007. URL: http://dx.doi.org/10.1007/s10009-006-0005-y, doi:10.1007/s10009-006-0005-y.

[13] Aleksandar S. Dimovski. Program verification using symbolic game semantics. *Theor. Comput. Sci.*, 560:364–379, 2014. URL: http://dx.doi.org/10.1016/j.tcs.2014.01.016, doi:10.1016/j.tcs.2014.01.016.

[14] Aleksandar S. Dimovski. Symbolic game semantics for model checking program families. In *Model Checking Software - 23nd International Symposium, SPIN 2016, Proceedings*, volume 9641 of *LNCS*, pages 19–37. Springer, 2016.

[15] Aleksandar S. Dimovski, Ahmad Salim Al-Sibahi, Claus Brabrand, and Andrzej Wasowski. Family-based model checking without a family-based model checker. In *22nd International SPIN Workshop on Model Checking of Software, SPIN '15*, volume 9232 of *LNCS*, pages 282–299. Springer, 2015.

[16] Aleksandar S. Dimovski, Ahmad Salim Al-Sibahi, Claus Brabrand, and Andrzej Wasowski. Efficient family-based model checking via variability abstractions. *STTT*, 2016. doi:10.1007/s10009-016-0425-2.

[17] Aleksandar S. Dimovski, Claus Brabrand, and Andrzej Wasowski. Variability abstractions: Trading precision for speed in family-based analyses. In *29th European Conference on Object-Oriented Programming, ECOOP '15*, volume 37 of *LIPIcs*, pages 247–270. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2015.

[18] Aleksandar S. Dimovski, Claus Brabrand, and Andrzej Wasowski. Finding suitable variability abstractions for family-based analysis. In *FM 2016: Formal Methods - 21st International Symposium, Proceedings*, volume 9995 of *LNCS*, pages 217–234, 2016. URL: http://dx.doi.org/10.1007/978-3-319-48989-6_14, doi:10.1007/978-3-319-48989-6_14.

[19] Alejandra Garrido and Ralph E. Johnson. Refactoring C with conditional compilation. In *18th IEEE International Conference on Automated Software Engineering (ASE 2003)*, pages 323–326. IEEE Computer Society, 2003. URL: http://doi.ieeecomputersociety.org/10.1109/ASE.2003.1240330, doi:10.1109/ASE.2003.1240330.

[20] Paul Gazzillo and Robert Grimm. Superc: parsing all of C by taming the preprocessor. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, 2012*, pages 323–334, 2012. URL: http://doi.acm.org/10.1145/2254064.2254103, doi:10.1145/2254064.2254103.

[21] Gerard J. Holzmann. *The SPIN Model Checker - primer and reference manual*. Addison-Wesley, 2004.

[22] Alexandru F. Iosif-Lazar, Ahmad Salim Al-Sibahi, Aleksandar S. Dimovski, Juha Erik Savolainen, Krzysztof Sierszecki, and Andrzej Wasowski. Experiences from designing and validating a software modernization transformation (E). In *30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015*, pages 597–607, 2015. URL: http://dx.doi.org/10.1109/ASE.2015.84, doi:10.1109/ASE.2015.84.

[23] Kyo C. Kang, Sholom G. Cohen, James A. Hess, William E. Novak, and A. Spencer Peterson. Feature-Oriented Domain Analysis (FODA) feasibility study. Technical report, Carnegie-Mellon University Software Engineering Institute, November 1990.

[24] Christian Kästner, Sven Apel, Thomas Thüm, and Gunter Saake. Type checking annotation-based product lines. *ACM Trans. Softw. Eng. Methodol.*, 21(3):14, 2012.

[25] Christian Kästner, Paolo G. Giarrusso, Tillmann Rendel, Sebastian Erdweg, Klaus Ostermann, and Thorsten Berger. Variability-aware parsing in the presence of lexical macros and conditional compilation. In *OOPSLA'11*, pages 805–824. ACM, 2011.

[26] Christian Kästner, Alexander von Rhein, Sebastian Erdweg, Jonas Pusch, Sven Apel, Tillmann Rendel, and Klaus Ostermann. Toward variability-aware testing. In *FOSD '12*, pages 1–8, 2012.

[27] Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. Frama-c: A software analysis perspective. *Formal Asp. Comput.*, 27(3):573–609, 2015. URL: http://dx.doi.org/10.1007/s00165-014-0326-7, doi:10.1007/s00165-014-0326-7.

[28] Flávio Medeiros, Márcio Ribeiro, Rohit Gheyi, Christian Kästner, and Sven Apel. An empirical study on configuration-related bugs. *Submitted for publication at IEEE TSE*, 2016.

[29] Jean Melo, Claus Brabrand, and Andrzej Wasowski. How does the degree of variability affect bug finding? In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 679–690, New York, NY, USA, 2016. ACM. URL: http://doi.acm.org/10.1145/2884781.2884831, doi:10.1145/2884781.2884831.

[30] Florian Merz, Stephan Falke, and Carsten Sinz. LLBMC: bounded model checking of C and C++ programs using a compiler IR. In *Verified Software: Theories, Tools, Experiments - 4th International Conference, VSTTE 2012, Proceedings*, volume 7152 of *LNCS*, pages 146–161. Springer, 2012. URL: http://dx.doi.org/10.1007/978-3-642-27705-4_12, doi:10.1007/978-3-642-27705-4_12.

[31] Jan Midtgaard, Aleksandar S. Dimovski, Claus Brabrand, and Andrzej Wasowski. Systematic derivation of correct variability-aware program analyses. *Sci. Comput. Program.*, 105:145–170, 2015. URL: http://dx.doi:10.1016/j.scico.2015.04.005, doi:10.1016/j.scico.2014.10.002.

[32] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer-Verlag, Secaucus, USA, 1999.

[33] H. Post and C. Sinz. Configuration lifting: Verification meets software configuration. In *ASE'08*, pages 347–350, LAquila, Italy, 2008. IEEE Computer Society.

[34] John C. Reynolds. *Theories of Programming Languages*. Cambridge University Press, 1998.

[35] Thomas Thüm, Sven Apel, Christian Kästner, Ina Schaefer, and Gunter Saake. A classification and survey of analysis strategies for software product lines. *ACM Comput. Surv.*, 47(1):6, 2014.

[36] Thomas Thüm, Ina Schaefer, Martin Hentschel, and Sven Apel. Family-based deductive verification of software product lines. In *Generative Programming and Component Engineering, GPCE'12*, pages 11–20. ACM, 2012. URL: http://doi.acm.org/10.1145/2371401.2371404, doi:10.1145/2371401.2371404.

[37] Alexander von Rhein, Thomas Thüm, Ina Schaefer, Jörg Liebig, and Sven Apel. Variability encoding: From compile-time to load-time variability. *J. Log. Algebr. Meth. Program.*, 85(1):125–145, 2016. URL: http://dx.doi.org/10.1016/j.jlamp.2015.06.007, doi:10.1016/j.jlamp.2015.06.007.