Journal of Software: Evolution and Process

JOURNAL OF SOFTWARE: EVOLUTION AND PROCESS
J. Softw. Evol. and Proc. (2013)
Published online in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/smr.1595

The Linux kernel: a case study of build system variability

Sarah Nadi*,† and Ric Holt

Software Architecture Group (SWAG), University of Waterloo, Ontario, Canada

SUMMARY

Although build systems control what code gets compiled into the final built product, they are often overlooked when studying software variability. The Linux kernel is one of the biggest open source software systems supporting variability and contains over 10,000 configurable features described in its KCONFIG files. To understand the role of the build system in variability implementation, we use Linux as a case study. We study its build system, KBUILD, and extract the variability constraints in its Makefiles. We first provide a quantitative analysis of the variability in KBUILD. We then study how the variability constraints in the build system affect variability anomalies detected in Linux. We concentrate on dead and undead artifacts, and by extending previous work, we show that considering build system variability constraints allows more anomalies to be detected. We provide examples of such anomalies on both the code block and source file level. Our work shows that KBUILD contains a large percentage of the variability information in Linux, so it should not be ignored during variability analysis. Nonetheless, the anomalies we find suggest that variability on the file level in KBUILD is consistent with KCONFIG, whereas the constraints on the code level are harder to keep consistent with both KBUILD and KCONFIG. Copyright © 2013 John Wiley & Sons, Ltd.

Received 6 July 2012; Revised 23 November 2012; Accepted 28 January 2013

KEY WORDS: software variability; variability anomalies; Linux; build systems; KBUILD

1. INTRODUCTION

Software variability allows users to configure different variants of a software system from the same code base by selecting their desired features. To achieve this, variability is supported by different parts of the software system such as source code, configuration scripts, and build scripts. Source code files and configuration files have been extensively studied in terms of their variability [1–6]. However, studying the variability of the build system (usually consisting of Makefiles to compile the source code) has fallen behind. Only a few papers, including previous work by us, have looked at the variability of the build system [7–11]. In this paper, we argue that because the build system is what really controls the composition of the final built product, its analysis is necessary in understanding the overall variability of the system. To study its role in variability implementation, we present a case study of the Linux kernel's build system, KBUILD.

The Linux kernel is one of the most important and widely used open source software systems. Linux's variability allows it to be used for various purposes and by different users. As a simple example, a user may choose to compile the kernel with Universal Serial Bus (USB) support, whereas another user may choose not to. Such variability supports Linux's portability where it can be used in different hardware devices ranging from embedded systems to large scale servers. However, providing such variability comes with the cost of a more complicated design, and thus, increased maintenance effort. This is especially the case in Linux that supports over 10,000 features and serves millions of users.

*Correspondence to: Sarah Nadi, Software Architecture Group (SWAG), University of Waterloo, Ontario, Canada.
†E-mail:snadi@uwaterloo.ca

Variability in Linux depends on three distinct artifacts shown in Figure 1: source code files (*code space*), KCONFIG files (*KCONFIG space*), and KBUILD Makefiles (*make space*) [1, 7]. Each of these spaces contains *constraints* governing the variability in the system. These constraints describe the KCONFIG features that must be selected for a particular functionality to be present in the compiled kernel, as well as which features are allowed to be simultaneously selected. The constraints in all variability spaces must be consistent. For example, if KCONFIG feature `F1` depends on the absence of feature `F2`, written as `!F2`, and code block `B1` depends on both `F1` *and* `F2`, then `B1` will never compile because both features cannot be selected at the same time. We refer to such cases as *variability anomalies* because they contain some unexpected behavior related to the variability implementation of the system. Such anomalies may lead to decreased reliability and increased maintenance effort.

To study variability in KBUILD, we implement a tool that extracts the constraints enforced in the Makefiles (*make constraints*). These are the conditions that must be satisfied for a source file to compile. We provide quantitative analysis of the variability in KBUILD by studying how KCONFIG features are used in these make constraints. Through examining KBUILD's Makefiles, we show that a major part of Linux's variability implementation occurs in KBUILD, and that most of Linux's code files are conditionally compiled under the control of the user's feature selection. We then study the effect of the extracted make constraints on variability anomalies in Linux. We do so by extending previous work by Tartler *et al*. [1] that detected code anomalies by analyzing the constraints in the code space and KCONFIG space using the tool, UNDERTAKER [12]. However, they do not include the make constraints as part of their analysis. We extend their work to consider the make constraints in the anomaly detection process. Our results show that more variability anomalies are discovered when the make constraints are considered. We examine two types of variability anomalies (dead and undead artifacts) at two levels of analysis (code blocks and code files).

This paper is an extended version of our earlier work [11], which was the first work to examine the effect of the make space constraints on variability anomalies in details and contributed the following:

- A technique to extract the variability constraints from Linux Makefiles.
- Boolean formulas needed to use the make constraints to detect anomalies as an extension to the work by Tartler *et al*. [1].
- Demonstration that the make constraints allow us to detect more variability anomalies.

We extend that work to provide a more in-depth study of variability in KBUILD, by providing the following:

- Quantitative analysis of the use of features in KBUILD to control source file compilation.
- A set of metrics partly adapted from previous work [6] to quantify variability in KBUILD.
- An improved extractor to derive more accurate make constraints from KBUILD.
- An updated analysis of the effect of make constraints on variability anomalies using the modified extractor.

The rest of this paper is organized as follows. Section 2 provides background information about variability in the Linux kernel. Section 3 describes the tools we use in our analysis including our make space constraint extractor and UNDERTAKER. Section 4 presents our quantitative analysis of the variability in KBUILD. Section 5 then reports the variability anomalies detected at both the code block and code file level after considering the make constraints. Section 6 presents the threats to the validity of this study. Section 7 presents related research, and Section 8 concludes this paper.

2. BACKGROUND: VARIABILITY IN THE LINUX KERNEL

Figure 1 illustrates the process of building the Linux kernel. The three variability spaces are shown in dashed boxes, and each box consists of the Linux kernel artifacts that are part of the space. Linux is configured through tools that read the KCONFIG files and display them to the user in menu format. This produces two files containing the user's feature selection: (i) `.config` used by KBUILD to control which files become compiled, and (ii) `autoconf.h` included by the `gcc` compiler in all

DOI: 10.1002/smr

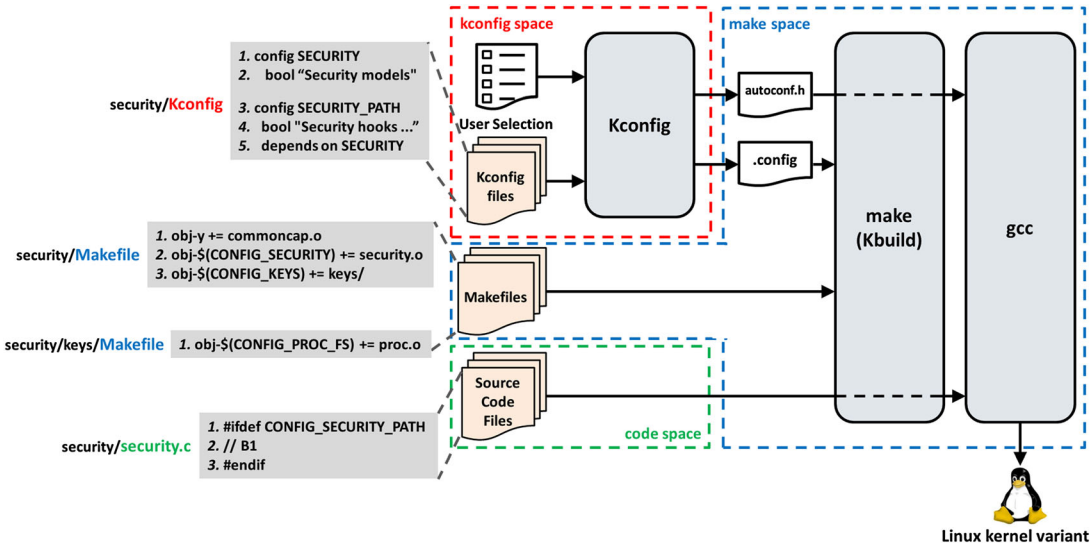

Figure 1. Linux build process.

source files to control which parts of the code become compiled. Thus, there are two levels of variability control in Linux: the file level (i.e., conditionally compiling whole source code files) and the code block level (i.e., conditionally compiling code blocks within source files). We now explain how each of these artifacts supports variability in Linux.

2.1. KCONFIG space (*K*)

The KCONFIG space *K* consists of KCONFIG files that describe the various features of the Linux kernel and their interdependencies. Each directory usually contains a KCONFIG file to describe the features related to the functionalities of this directory. For example, Figure 1 shows a snippet from the KCONFIG file in the `security` subsystem folder.[‡] In lines 1 and 3, two features, `SECURITY` and `SECURITY_PATH`, are defined. Both features are of type bool, which means that they are either present or not. The `SECURITY_PATH` feature depends on the feature `SECURITY` (line 5), which means that the user will not be able to select `SECURITY_PATH` unless `SECURITY` is also selected. Such relationships are what we refer to as the KCONFIG *constraints*. Further details on the format of KCONFIG files and the types of features can be found in other work [2, 5].

2.2. *Code space (C)*

The code space *C* consists of *C* source and header files, as well as some scripts that implement the functionality of the kernel. Certain parts of the code are conditionally compiled. That is, they are only compiled if certain configuration features are selected. This is performed by using *C* preprocessor (CPP) directives such as `#ifdef` and `#ifndef`. Entries in `autoconf.h` contain the user's selection in a format understandable by CPP. For example, when `SECURITY` is selected, the following entry will be generated in `autoconf.h`: `#define CONFIG_SECURITY 1`.[§] In Figure 1, the code space snippet from `security.c` contains a code block, B1, that is guarded by the CPP directive `CONFIG_SECURITY_PATH`. This means that B1 will only be compiled if the user selects this feature (i.e., the feature has a corresponding definition in `autoconf.h`). We refer to such constraints as the *code constraints*.

[‡]Code snippets have been slightly modified for simplicity and better illustration.
[§]Note that there is a convention that a CONFIG_ prefix is attached to the name of the features in these generated files.

DOI: 10.1002/smr

2.3. Make space (M)

The make space, which is the focus of this paper, is responsible for compiling source files into the final executable product through KBUILD. KBUILD consists of Makefiles that use a special notation to conditionally compile whole source files and are read by the tool `make`. KBUILD uses the file `.config` to read the user's selection. For example, if the user selects feature `CONFIG_SECURITY`, an entry `CONFIG_SECCURITY=y` is generated in `.config`.

Each source directory usually contains a Makefile that controls the compilation of files in that directory. A list called `obj-y` is used to collect the files that will be compiled. Figure 1 shows snippets from two separate Makefiles in the `security` directory. In the first file, `security/Makefile`, line 1 indicates that the source file `commoncap.c` will be compiled into `commoncap.o` and unconditionally added to the `obj-y` list. Line 2 indicates that `security.c` will be compiled into `security.o` only if `CONFIG_SECURITY` is selected by the user. This means that `CONFIG_SECURITY` has the value 'y' in `.config`, and thus, the expression will evaluate to `obj-y += security.o` that adds the compiled file `security.o` to the list of files in the final compiled kernel image. We refer to such constraints as the *make constraints*.

In some cases, a whole directory can also be conditionally compiled. For example, line 3 of the top Makefile snippet in Figure 1 indicates that the directory `keys` (i.e., the files within it) will only be compiled if `CONFIG_KEYS` is selected. This indicates to MAKE that it should descend into the `keys` directory and read the Makefile there.

3. TOOLS USED TO ANALYZE KBUILD VARIABILITY

Our work is divided into two main parts. The first explores and quantifies variability in KBUILD and the second determines the effect of this variability on anomalies in Linux. To accomplish this, we need two sets of tools or analyses. The first for extracting variability constraints in KBUILD and the second for detecting anomalies. For the first, we develop a *Makefile constraint extractor*, MAKEX, and for the second, we use the results of MAKEX and extend UNDERTAKER to detect anomalies. In this section, we explain how MAKEX and UNDERTAKER work. The performance reported in both cases is based on using a machine with two quad-core Intel Xeon 2.67 GHz CPUs and 16 GB RAM.

3.1. Extracting make constraints

3.1.1. MAKEX. A *presence condition* of a source file determines when this file is compiled. To extract these presence conditions, we implement a prototype constraint extractor MAKEX[¶] that recursively reads all the Makefiles in the source code directories and generates the corresponding constraints. Listing 1 shows an example of such constraints extracted from the Makefiles in Figure 1.

Because some source files are only compiled on certain CPU architectures, we extract a different set of make constraints for each CPU architecture supported in Linux. MAKEX is implemented in Java and uses text-based pattern matching to extract the constraints from the Makefiles. MAKEX searches for the `obj-y` occurrences and the files added to them. For example, the first line in the top Makefile snippet in Figure 1 indicates that `commoncap.c` is compiled unconditionally. Therefore, MAKEX generates the entry in line 1 in Listing 1 that means that this file has no constraints.[**] The second line in the Makefile snippet in Figure 1 means that `security.c` is only compiled if `CONFIG_SECURITY` is chosen. In this case, MAKEX generates the constraint in line 2 of Listing 1.

Line 3 of Listing 1 indicates that `proc.c` is compiled only if both `CONFIGCONFIG_PROC_FS` and `CONFIG_KEYS` are selected. This is because any file in the keys directory is only compiled if `CONFIG_KEYS` is selected (line 3 of the first Makefile in Figure 1), whereas file `proc.c` itself is compiled only if `CONFIG_PROC_FS` is selected (line 1 of the second Makefile), which means that we have to combine both conditions when generating the corresponding make constraints.

[¶]MAKEX is available online at http://swag.uwaterloo.ca/~snadi/KbuildVariability.html
[**]Note that when writing out the constraints, we use the .c extension of the file name.

DOI: 10.1002/smr

Listing 1 Example make constraints. These constraints determine the KCONFIG features each code file depends on.

```
1. security/commoncap.c
2. security/security.c <-> CONFIG_SECURITY
3. security/keys/proc.c <-> CONFIG_KEYS && CONFIG_PROC_FS
```

3.1.2. Limitations and evaluation. Linux's Makefiles are difficult to parse statically [13] because they use specialized and complicated syntax to represent special cases and are not consistently structured. The previous section has illustrated the most common entries and patterns responsible for source file compilation in KBUILD. There are other more specialized patterns scattered throughout the Makefiles in KBUILD. Some of these are discussed in our previous work [7, 11]. For each pattern we recognize, we need to add a new pattern matching function in MAKEX's implementation. In our original work [11], we ignored some special cases such as conditional blocks in Makefiles. In this paper, we improved MAKEX to handle conditional blocks in Makefiles, as well as some special cases such as the use of `machine-y` to specify machine directory names in the `ARM` architecture. For example, the entry `machine-$(CONFIG_ARCH_AT91) :=at91` actually indicates visiting the directory `mach-at91/` if feature `ARCH_AT91` is selected.

Although MAKEX still has some limitations because of static parsing challenges, it achieves a reasonable coverage rate in terms of covering the variability points in KBUILD. There are three coverage metrics we use for evaluation: (i) percentage of Makefiles MAKEX analyzes, (ii) percentage of .c files MAKEX finds presence conditions for, and (iii) percentage of KCONFIG features used in KBUILD for which MAKEX found presence conditions using them. For the first metric, we achieve a 75% Makefile coverage rate, which means that we analyzed 75% of the Makefiles present in the kernel. For the second metric, we achieve an 85% source file coverage rate, which means that we were able to find presence conditions for 85% of the source files present in Linux. In the initial version of MAKEX used in our previous work [11], we only achieved a 74% source file coverage rate. Thus, we have improved our source file coverage rate by 11% when we considered more of the special cases used. For the third metric, the feature coverage rate of MAKEX is 93% (i.e., MAKEX was able to see the effect of 93% of the KCONFIG features appearing in Makefiles).

3.1.3. Performance and scalability. MAKEX runs in a single thread starting from the Makefile in the root of the kernel's source code directory and recursively reads nested Makefiles as needed. Analyzing all architectures in a single release of the Linux kernel runs in approximately 51 s. Although it is quite fast, its limitations lie in having to customize the patterns detected according to the system being analyzed. However, for the purposes of exploring KBUILD, and not providing a comprehensive tool, such limitation does not affect our results.

3.2. *UNDERTAKER and anomaly detection*

3.2.1. Overview. Because variability information is stored in three different places (KCONFIG files, source code files, and Makefiles), inconsistencies are likely to arise. The UNDERTAKER tool [1] extracts the constraints for each CPP guarded code block as well as the constraints in the KCONFIG files. These constraints are then combined into a Boolean formula that is fed into a satisfiability (SAT) solver that tries to satisfy the formula. If there are conflicts between the constraints, then the formula cannot be satisfied and the SAT solver reports that. An unsatisfiable formula suggests an anomaly. Anomalies are manifested as *dead* and *undead* code blocks. A *dead* block is a CPP-guarded block that can never be compiled on any valid configuration, and an *undead* one is a CPP-guarded block that is always compiled when its parent[††] is compiled. In this work, we use the latest version of UNDERTAKER (version 1.3).

To study the effect of the variability constraints in KBUILD, we use the make constraints extracted from KBUILD (Section 3.1) and include them in UNDERTAKER's analysis of dead and undead code blocks. We also modify the analysis to work at the file level to detect dead and undead source files. We combine the constraints from all three spaces (KCONFIG space (*K*), code space (*C*), and make

[††]A CPP guarded block has a parent if it is nested within another CPP guarded block.

DOI: 10.1002/smr

space (M)) in Boolean formulas that if not satisfied, detect dead and undead artifacts as shown in the succeeding text. B_N or $Block_N$ denote a code block, and F_N or $File_N$ denote a whole source file. The formulas essentially ensure that a code block or code file is a variable such that there are configurations where it will be compiled and others where it will not be compiled.

$$Dead_{B_N} = \neg sat(Block_N \wedge C \wedge M \wedge K) \tag{1}$$

$$Undead_{B_N} = \neg sat(\neg Block_N \wedge parent(Block_N) \wedge C \wedge M \wedge K) \tag{2}$$

$$Dead_{F_N} = \neg sat(File_N \wedge M \wedge K) \tag{3}$$

$$Undead_{F_N} = \neg sat(\neg File_N \wedge M \wedge K) \tag{4}$$

For code block formulas, constraints from all three spaces are combined. For file formulas, only constraints from the make and KCONFIG spaces are combined because we do not look at the block level. More details about how the Boolean formulas are derived can be found in our previous work [11]. As an example of using these formulas, consider the code block B1 shown in the code snippet in Figure 1. From the code space, B1 depends on `CONFIG_SECURITY_PATH`. From the make space, the whole source file `security.c` depends on `CONFIG_SECURITY`, and from the KCONFIG constraints, `CONFIG_SECURITY_PATH` depends on `CONFIG_SECURITY`. If we want B1 to compile, all these constraints must be satisfied. Listing 2 illustrates the combination of the three sets of Boolean constraints according to Formula 1 to determine if `B1` will compile. If this formula cannot be satisfied, then `B1` will never compile, and thus, will be *dead*. In Listing 2, the formula can be satisfied, so no anomaly is detected.

Listing 2 Example of a boolean formula to detect a dead block with the combination of code constraints, make constraints, and KCONFIG constraints

```
B1 && (B1 -> CONFIG_SECURITY_PATH)
&& (CONFIG_SECURITY) && (CONFIG_SECURITY_PATH -> CONFIG_SECURITY)
```

3.2.2. Performance and scalability. The analysis in UNDERTAKER is extremely parallelized such that several code blocks can be simultaneously evaluated to improve performance. When detecting anomalies using UNDERTAKER, an analysis of a single Linux release takes approximately 45 min to run using four parallel threads. Given the size of the whole kernel, such performance is acceptable.

4. MEASURING VARIABILITY IN KBUILD

In this section, we investigate the extent of the role played by KBUILD in the variability implementation in Linux. To do so, we need to measure the variability in KBUILD and compare it to the rest of the system, where applicable. In this section, we first explain the metrics we use. We then describe the setup and results of our analysis, and then provide our interpretation of these results.

4.1. Metrics

To the best of our knowledge, there are no standard metrics to measure variability and its complexity. Some metrics were introduced by Liebig *et al*. [6] to measure CPP variability in code. We adapt some of these metrics (NOF, SD, TD, and GRAN) for measuring variability in KBUILD and also introduce some of our own (POF, POCCF, POCCD) as follows:

Number of features (NOF) and percentage of features (POF). The variability in Linux arises from its configuration features. All features (K) are defined in the KCONFIG files and control the final compiled kernel in one of two ways: in the code space (C) through `cpp` directives or in the make space (M) to control source file compilation. Some features may be used in neither or in both spaces. This yields four categories of uses of KCONFIG features: (i) $K - C - M$, features that are not used in neither the

DOI: 10.1002/smr

code nor make spaces but are rather used internally within KCONFIG to support dependency constraints between other features, (ii) $C - M$, features that are only used in the code space, (iii) $C \cap M$, features that are used in both the code and make spaces, and (iv) $M - C$, features that are only used in the make space. We use the NOF and POF metrics to describe the number and POF in each of the four categories as follows:

1. $\mathbf{NOF}_{K-C-M}$ (or $\mathbf{POF}_{K-C-M}$): number (percentage) of features that are defined in KCONFIG and *only* used there.
2. $\mathbf{NOF}_{C-M}$ (or $\mathbf{POF}_{C-M}$): number (percentage) of KCONFIG features *only* used in code space.
3. $\mathbf{NOF}_{C \cap M}$ (or $\mathbf{POF}_{C \cap M}$): number (percentage) of KCONFIG features used in *both* code and make spaces.
4. $\mathbf{NOF}_{M-C}$ (or $\mathbf{POF}_{M-C}$): number (percentage) of KCONFIG features *only* used in make space.

Percentage of conditionally compiled files (POCCF) and percentage of conditionally compiled directories (POCCD). The POCCF and POCCD metrics measure the percentage of files and directories, respectively, in KBUILD that are conditionally compiled according to some feature selection. Such a metric illustrates how much variability is present at the source file level. That is, whether most files are conditionally compiled or are compiled by default in every variant.

Scattering degree (SD) and tangling degree (TD): We use these metrics to quantify feature usage in KBUILD. The SD of a feature is its number of occurrences in different file presence conditions in KBUILD. Conversely, the TD is the number of different features that occur in a file presence condition. For each release, we state a single SD or TD that calculates the average for that release.

Granularity (GRAN): KCONFIG features used in KBUILD control the compilation of specific source code files as well as whole directories. We consider two levels of GRAN of control. At a high level of GRAN, a feature controls a directory that generally contains several source files implementing some related functionality, for example, sound or USB support. We define GRAN_{dir} as the POF used in KBUILD that control directories. At a low level of GRAN, a feature only controls the compilation of source code files that generally implement a specific part of this functionality. We use GRAN_{file} to measure the POF used in KBUILD that *only* control source code files. Note that high level GRAN features still appear in the presence conditions of source code files. This is because source code files in a directory will not be compiled unless their containing directory is also compiled (Section 2.3).

4.2. Analysis and results

We study 10 recent versions of Linux spanning a period of around 2 years. Examining several releases ensures that conclusions we draw are not just specific to one release. It also provides an evolutionary view of KBUILD variability. All numbers reported (unless otherwise specified) are the median of the metric being measured over all releases examined. We divide our results into four questions.

1. *How many KCONFIG features does each space use?* We use NOF and POF to measure the variability in each space in terms of the number and POF used. Figure 2 presents our findings in terms of how KCONFIG features are used according to the four categories explained earlier. Figure 2 shows that the total NOF defined in KCONFIG, NOF_K (column height), is growing in each release. When we compare percentages to see which part of the system uses most of these features, we find that the POF used in the make space, POF_M,[‡‡] is 63% versus 35% used in the code space (POF_C[§§]) that suggests that more variability takes place in KBUILD. Over the 10 releases examined, POF_{M-C} is 48%, whereas POF_{C-M} is 17%. This means that a higher POF are *only* used in KBUILD to control whole source file compilation rather than being used in the code to control code block compilation. Additionally, if we look at the percentages shown in each category over the releases, we can see that POF_{M-C} is growing, whereas POF_{C-M} and $\text{POF}_{C \cap M}$ represent about the same percentage in all examined releases.

[‡‡]POF_M is obtained by adding POF_{M-C} and $\text{POF}_{C \cap M}$ in Figure 2.
[§§]POF_C is obtained by adding POF_{C-M} and $\text{POF}_{C \cap M}$ in Figure 2.

DOI: 10.1002/smr

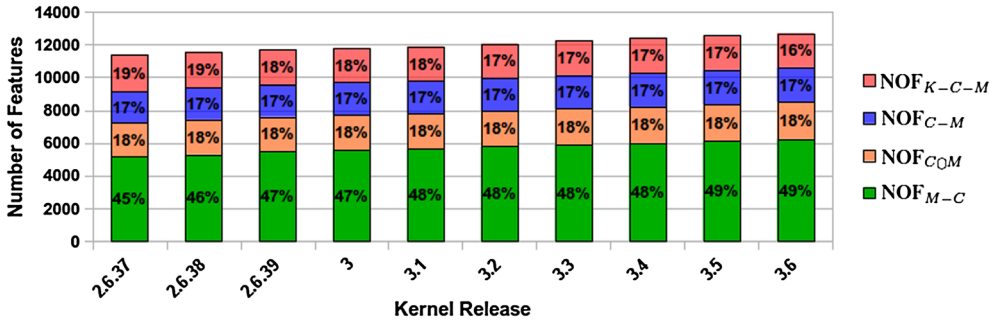

Figure 2. Feature usage across Linux releases. NOF is number of features used in KCONFIG space only (*K-C-M*), code space only (*C-M*), make space and code space ($M \cap C$), and make space only (*M-C*).

Finding 1: The majority of KCONFIG features are used in the make space. A total of 46% of KCONFIG features are only used in the make space, and this percentage is growing over time.

2. *How many files and directories are conditionally compiled in KBUILD?* Our analysis shows that throughout these 10 kernel releases, the POCCF is 92%. This means that 92% of the source code files' compilation depends on one or more KCONFIG features. Similarly, the POCCD is 84%. These numbers indicate that most of the source files within Linux are controlled by the user's selection of configuration features and are not compiled by default.

Finding 2: A total of 92% of source files and 84% of source directories are conditionally compiled.

3. *What GRAN do features mostly control in KBUILD?* For all 10 releases, we find a GRAN$_{file}$ of 88%, which means that 88% of the features used in KBUILD control the compilation of source files only (i.e., low GRAN), whereas the remainder 12% control both files and directories (i.e., high GRAN). We use v3.3 in Figure 3 to show the number of directories and source files controlled by each configuration feature we found. Each dot on the graph corresponds to a particular configuration feature that appears in KBUILD as found by MAKEX (total of 7543 features) and shows the number of directories and files it controls. The distribution of feature usage is skewed towards the bottom left that indicates that most of the variability in KBUILD is at a low level of GRAN. We find that on average, a feature controls 0.2 directories and three source files, and that around 78% of these features control exactly one source file.
   To illustrate how this control works, consider the `SCSI` feature (circled on the graph). Its corresponding flag `CONFIG_SCSI` controls 30 directories and 303 files that support the `SCSI` driver. Directories controlled are an example of high level GRAN. Now, we will consider the specific bus types within the `SCSI` driver. These are at a lower level of GRAN and represent more specific functionalities governed by additional features besides `SCSI`. For example, file `in2000.c` in `SCSI`'s directory implements an `ISA SCSI` host adapter that is only compiled if both `SCSI` and `SCSI_IN2000` flags are turned on. `SCSI_IN2000` is an example of a low GRAN feature.Figure 3 also shows several outliers in the right half of the graph. Two main outliers shown on the top right are features `STAGING` and `SND`. `STAGING` controls the `drivers/staging` directory that contains code that is still under development and has not been finalized for full integration into the kernel yet. The fact that feature `STAGING` is an outlier comes at no surprise because there are 62 directories directly under the `/drivers/staging/` directory apart from the subdirectories under each of those. The `staging` directory itself would not be visited unless the `STAGING` feature is selected. The need for the `STAGING` feature to be selected would then be propagated to all subdirectories and files underneath it that results in `STAGING` controlling 84 directories and 531 files. The same applies to `SND` that controls the `SOUND` directory that has 21 main directories.

DOI: 10.1002/smr

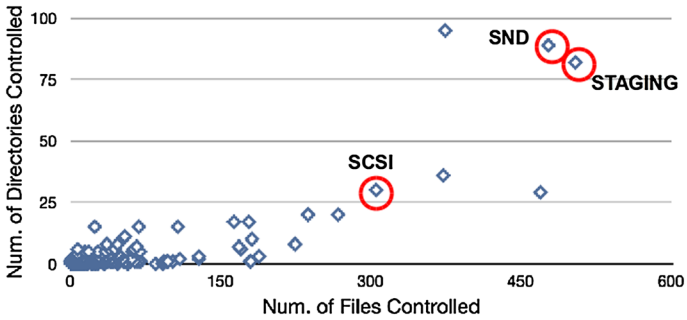

Figure 3. Granularity of control of features in KBUILD v3.3. Each point represents a KCONFIG feature used in the Makefiles.

Finding 3: A total of 88% of the features used in KBUILD have low level GRAN control whereas 78% control exactly one source file.

4. *How complex are the make constraints?* We next examine particular aspects of the presence conditions of the source files in Linux to determine how many features usually control the compilation of a source file. We find that although more than half of the KCONFIG features are used in KBUILD, the presence conditions of files are not complex. We find that the TD of features in the presence conditions is 2. This means that conditionally compiled files have an average of only two configuration features in their presence conditions. These two features are usually the feature controlling the directory, and then the feature controlling the specific lower level functionality (e.g., line 3 in Listing 1). If we only consider the features that directly control a file (i.e., apart from the feature(s) that control the file's directory), we find that 76% of source files have only one feature in their presence condition. We also found that the SD of a feature used in KBUILD is 2, which means that on average, a feature appears in two different presence conditions.

Finding 4: The Presence conditions of files in KBUILD are not complex. The make space constraints have a TD of 2 features, and features used in KBUILD also have a SD of 2. Additionally, 76% of source files have only one feature in their presence condition (apart from the directory control feature).

4.3. Interpretation of findings

Finding 1 implies that variability in the make space is growing in terms of its usage of KCONFIG features with respect to the rest of the system. We explain this phenomenon as follows. Each time a new source file is added to the Linux kernel source code, an entry must be created in KBUILD so that the file compiles. The majority of new kernel code is drivers implementations, and drivers are usually conditionally compiled because they differ from one platform or machine to another. This means that each time a new driver is added, a feature is added for it in KCONFIG so that the user can select it, and this feature will control the compilation of the implementation source file in KBUILD. Given Finding 2 that 88% of source code files are conditionally compiled, we can safely say, that in most cases, whenever a new file is added, a new configuration feature will be used to control it in KBUILD. However, the same does not apply for #ifdef variability. A new file may be added with an entry in KBUILD, but this file may not contain any #ifdef blocks. Thus, most files will have a conditional compilation entry in KBUILD but not necessarily conditionally compiled blocks.

If we look at Findings 3 and 4 together, we can deduce that there is commonly a one to one mapping between a feature and the source file it controls. This means that most of the time, a file depends on a single feature, and this feature only controls this file. This is an interesting characteristic of the Linux kernel because it indicates that the user's selection directly controls the compilation of whole source files.

DOI: 10.1002/smr

5. EFFECT OF KBUILD ON DETECTING VARIABILITY ANOMALIES

5.1. Overview

The findings of the previous section show that more KCONFIG features are used in KBUILD to control source file variability as opposed to being used in the code space to control code block variability. This serves as a motivation for further studying the effect of this variability on the quality of the kernel. We do this by analyzing the effect of the constraints in KBUILD on variability anomalies in the Linux kernel. In this section, we discuss the results of this analysis that detects dead and undead code blocks and code files in the Linux kernel. In both levels of analysis, we investigate the effect of adding the make constraints on discovering variability anomalies. That is, on a block level, we compare using only the constraints from the code space and KCONFIG space versus also adding the make constraints. On the file level, we investigate if we can find any dead or undead files due to conflicts between the make constraints and the KCONFIG constraints.

We use the same 10 kernel releases from the previous section. For each release, we run the analysis on the code block level and on the file level. In this work (as opposed to our previous work [11]), we choose to ignore the `drivers/staging` directory in our analysis. The `staging` directory by definition contains code that has not been completely finalized, and so is expected to still contain some anomalies. To avoid skewing our results, and to focus on anomalies in the actual kernel code, we do not perform our block or file level analysis on any of the files in the `staging` directory.

Variability anomalies mainly arise from conflicts between constraints in one or more of the three spaces supporting variability in Linux. On the block level, these conflicts include the following:

1. *Code*: conflict within code constraints themselves, for example, dead block: `B1 -> F1 & !F1`.
2. *Code*-KCONFIG: conflict between code constraints and KCONFIG constraints, for example, dead block: `(B1 -> F1 && F2) && (F1 -> !F2)`.
3. *Code-make*: conflict between code constraints and make constraints, for example, dead block: `(B1 -> !F1) && F1` where `F1` is the presence condition of the code file containing `B1`.
4. *Code-make*-KCONFIG: conflicts between combination of code, make, and KCONFIG constraints, for example, undead block: `(B1 -> F1) && F2 && (F2 -> F3 && F1)`.
5. *Code*-KCONFIG *missing*: conflicts between code and KCONFIG constraints because of missing feature definitions, for example, dead block: `(B1 -> F2) && (F2 -> F3)`, but `F3` is not defined in KCONFIG.
6. *Code-make*-KCONFIG *missing*: conflicts between all three spaces because of missing feature definitions, for example, dead block: `(B1 -> F1) && F2 && (F2 -> F3)`, but F3 is not defined in KCONFIG).

On the file level, the code space no longer plays a role, so anomalies may arise because of the following:

1. *Make*-KCONFIG: conflicts between make constraints and KCONFIG constraints, for example, dead file: `(File1 -> F2 && F3) && (F3 -> !F2)`.
2. *Make*-KCONFIG *missing*: conflict between make constraints and KCONFIG constraints due to missing feature definitions, for example, dead file: `(File1 -> F2) && (F2 -> F3)`, but `F2` is not defined in KCONFIG.

The next section presents the results of our analyses and discusses illustrating anomaly examples in various categories.

5.2. Results

Block level. When we enhance the block level analysis with make constraints, we detect an average of 20% additional anomalies when compared with just using the code and KCONFIG constraints. Figure 4 shows the percentage of additional code block anomalies (both dead and undead) detected in each of the three categories involving make constraints. Throughout the 10 releases shown in Figure 4, we can see that the anomalies caused by conflicts between all three spaces (code-make-KCONFIG) constitute most of the additional anomalies detected. Because detecting this category of anomalies

DOI: 10.1002/smr

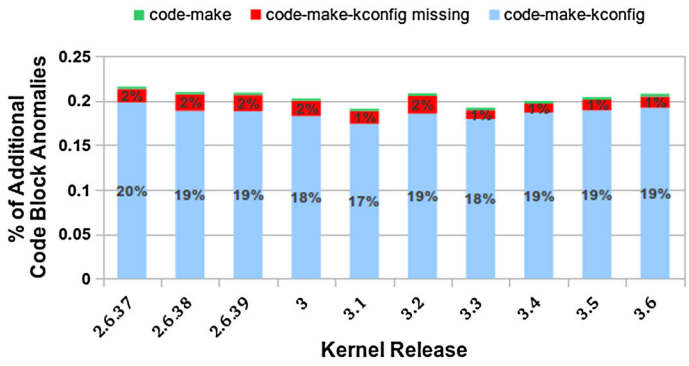

Figure 4. Percentage of additional code block anomalies due to adding the make constraints to the analysis. The code-make represents the smallest percentage of anomalies detected.

requires solving a complex SAT formula, it suggests that these anomalies are hard to find manually by the developer and that having automated tools to detect them is important.

Finding 5: Considering the make constraints when detecting variability anomalies allows more anomalies to be detected (20% increase).

Finding 6: Code block anomalies caused by a conflict between all spaces (code-make-KCONFIG) are more common and are also harder to manually detect by developers.

File level. On the code file level, we detected an average of 56 dead files in each release because of missing KCONFIG definitions. This means that the file either directly or indirectly depends on a feature that is not defined in KCONFIG and therefore never becomes compiled. Over the 10 releases examined, there was a total of 52 unique dead files (an anomaly can persist over several releases). We did not detect any undead files.

Finding 7: Dead files detected are caused by missing features. No dead files due to direct conflicts between make and kconfig spaces found.

We now provide examples in each of the anomaly categories involving make constraints on both levels of analysis.

5.2.1. Code-make block anomalies. Code-make anomalies are a result of a direct conflict between the code constraints and the constraints in the Makefiles. Over the 10 releases examined, we found 11 distinct code-make anomalies. We found that in some cases, these dead blocks are intentional by the developers so that they mark invalid feature configurations in the code. For example, two dead blocks contained code such as `#error invalid SiByte UART configuation` and `#error unknown platform`, which means that the developers are aware that the feature combinations enabling these blocks should never happen. The remaining dead and undead code-make block anomalies involved actual code. We investigated two of these anomalies that occurred in file `arch/sparc/kernel/jump_label.c`. According to the make constraints, this file is only compiled if `SPARC64` is selected. Within the actual file, there is a code block that does one thing if `SPARC64` is selected and another if it is not. However, because the file will not be compiled without this feature in the first place, then the block depending on it is always selected (i.e., undead), whereas the other block is never selected (i.e., dead). We submitted a patch to remove this unnecessary check, but the developer replied that the check is there so that 32-bit support is easy to add in the future if someone wants to do that.

DOI: 10.1002/smr

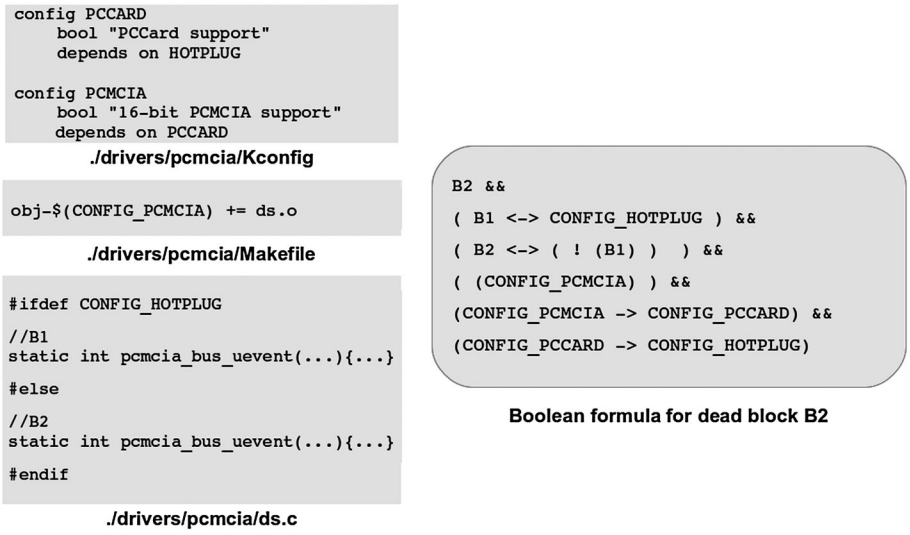

Figure 5. Example of a code-make-KCONFIG dead block anomaly.

This response indicates that developers might intentionally leave dead/undead code behind for future anticipated maintenance. On the other hand, we submitted another patch for a different dead code-make block in file `arch/m68k/sun3/prom/init.c`, and the developer accepted the patch stating that this dead code has been copied from elsewhere, but it is actually not relevant to the functionality here and shall be removed.

5.2.2. Code-make-KCONFIG block anomalies. Anomalies in the code-make-KCONFIG category are caused by a conflict involving all three spaces. This category differs from the previous one in that it is not caused by conflicts of direct dependencies in the code and make spaces but conflicts caused by indirect dependencies that are exhibited in the KCONFIG constraints. Figure 5 provides an example of a code-make-KCONFIG anomaly.[¶¶] As shown in the example, the code file depends on `PCMCIA` as indicated in the Makefile. Code block `B1` depends on `HOTPLUG`, whereas code block `B2` depends on `!HOTPLUG` as shown in the code. In the KCONFIG file, we can see that `PCMCIA` depends on `PCCARD` which in turn depends on `HOTPLUG`. This means that given the file is compiled, block `B2` can never be selected because `HOTPLUG` will always be enabled for the file to compile. This is shown in the Boolean formula illustrated in Figure 5.

5.2.3. Make-KCONFIG file anomalies. In our previous work [11], we had reported a few dead files in this category, but after some investigation, we discovered that this was due to an error in the KCONFIG parsing carried out by UNDERTAKER, which we had reported. After correcting this error in our current work, we did not detect any dead files in this category.

5.2.4. Missing features. Missing features are those that appear in the presence condition of a code block or file but have no definition in KCONFIG [1]. These can appear on the block or file level of the analysis and result in the *code-make-KCONFIG missing* and *make-KCONFIG missing* anomaly categories. Features that are missing on the code file level causing the file to be dead will also be missing on the block level for the code blocks within that file. However, on the code file level, only one anomaly would be reported for the whole file, whereas on the block level, several anomalies would be reported for each dead/undead code block in the file.

For example, four different dead code-make-KCONFIG missing block anomalies were detected in the file `drivers/spi/spi-stmp.c` directory. The make constraints indicate that `SPI_STMP3XXX` needs to be defined for the source file to compile. However, `SPI_STMP3XXX` depends on another feature, `ARCH_STMP3XXX`, which has no KCONFIG definition. Thus, all the conditional code

[¶¶]Snippets have been slightly modified for simplicity and better illustration. Original files contain concepts not discussed in this paper such as menuconfig and tristate [5] in KCONFIG files and composite variables in Makefiles [11].

DOI: 10.1002/smr

blocks in the file are reported as code-make-KCONFIG missing. On the file level, the same missing feature caused the file to be dead but only one anomaly report is generated. This anomaly has been introduced in v3.1 by creating the code file, and its entry in the Makefile, and has stayed in the kernel until the last release we examined, v3.6.

5.3. *Interpretation of findings*

Because we did not find any dead files due to direct conflicts between the make space and KCONFIG constraints (Finding 7), this suggests that there is usually no direct conflict between the presence condition of a file and the KCONFIG constraints related to that presence condition. However, there are a number of conflicts between the presence condition of a file, and that of the code blocks within the file along with the KCONFIG constraints related to them (Finding 6). This suggests that although a large percentage of the KCONFIG features are used in the Makefiles, developers have no problem choosing the right feature to control a file without breaking the KCONFIG constraints. This may be the case because there is a one-to-one mapping between features and the files they control, and a source file usually has a single entry in KBUILD that makes it easier to manage. On the other hand, keeping the code block presence conditions in sync with those of the whole file as well as the KCONFIG constraints seems more of a challenge. The inconsistencies we detect suggest that more automatic support for consistency checking at the block level is needed. Integrating tools such as UNDERTAKER into the development process where developers can check if their changes caused any inconsistencies may be helpful.

Our analysis shows that some of the anomalies caused by missing features on the block level may also be caught on the file level but with fewer anomalies presented to the developer (e.g., see the missing feature `ARCH_STMP3XXX` example in Section 5.2). However, many of the anomalies dealing with conflicts of constraints are only unique to the block level. Thus, it may be less time consuming for developers to start the analysis at the file level, and solve the issues there, which in turn will remove many of the block level anomalies, and then move down to the block level analysis to detect any remaining issues.

6. THREATS TO VALIDITY

6.1. *Internal validity*

In this work, we emphasize the importance of considering make constraints in variability analysis. Because we are extending the UNDERTAKER tool to discover variability anomalies, any shortcomings in their analysis will be reflected in our analysis. The UNDERTAKER tool is an ongoing work, and its authors are constantly updating it. Therefore, running the analysis with a different version could possibly yield a different number of anomalies.

Any problems with our extraction of the make constraints will also affect the results. There are certain parts of the Makefiles that are hard to parse using textual pattern detection. In our work so far, there are some of these aspects that we ignore such as `#define`'s used with the Makefiles to define additional variables that are later used. However, the frequency of such cases in the parts of the Makefiles that deal with KCONFIG features is low.

In terms of the anomalies discovered, some of these do not necessarily reflect errors. We choose to use the term *anomaly* precisely for this reason. A dead artifact may exist because of bad maintenance, and an undead artifact may be used as a form of checking that certain conditions actually hold. In both cases, we believe that developers should still be aware of such anomalies because they are potential sources of errors and undesired behavior.

6.2. *External validity*

We only examine one software system, Linux. However, Linux is the largest open source software system available. Our results, which conclude that most of the variability is implemented in the build system and that the constraints in the build system cause some variability anomalies, do not necessarily apply to other systems. Linux's build system, KBUILD, is complex and unique in terms

DOI: 10.1002/smr

of the customized notation it uses. However, there are many other configurable systems that use a similar structure for their build systems (e.g., BUSYBOX and BUILDROOT). Although we do not attempt to generalize our results beyond Linux, we believe that this work provides interesting findings that can be used to guide the study of variability in other build systems. To generalize and categorize variability anomalies caused by build systems beyond Linux, we plan to apply our analysis to other systems in order to improve external validity. This will also allow us to determine how variability is generally implemented in build systems, where other systems differ from KBUILD, and whether the technique used affects the quality of the software system.

7. RELATED WORK

Our work relates to two research areas: software variability and build systems. Software variability research studies how configurable software systems support variability and how they evolve over time. We divide build system research into three areas: studying the complexity of build systems, studying variability in build systems, and extracting variability constraints from build systems.

7.1. Software variability

The Linux kernel has been one of the main subjects of variability research due to its large size and the large number of supported features. The work on the Linux kernel (as well as other systems) has mainly focused on studying configuration and source code files (e.g., [1–3, 5, 14–17]). These papers focus on extracting the variability constraints from the source files and configuration files, and studying these constraints in terms of understanding things such as the feature model size, the evolution of constraints, as well as the evolution of both artifacts together.

Propositional logic has often been used to study the constraints in feature models. For example, Zengler [4] has encoded the constraints in KCONFIG as a single propositional logic formula that can be verified to ensure that the combination of features is valid. However, such a formula is very large, and would be very difficult to analyze. This is one reason why we opted for using the KCONFIG propositional encoding by Tartler *et al*. [1] as they provide a slicing algorithm that only chooses the constraints related to the artifacts in question.

7.2. Build systems

7.2.1. Complexity of build systems. Adams *et al*. [13] developed a tool, MAKAO, for visualizing and manipulating build systems and have applied it to KBUILD. They mainly focused on the targets that appear in Makefiles and their dependencies but did not study the configuration features that appear in Makefiles and how they contribute to Linux's configurability. Other work also by Adams *et al*. [18] studied the evolution of the Linux KBUILD files and how these files co-evolve with the source code. Their findings showed that the build system's complexity grows over time in terms of its size, and the number of targets it supports. McIntosh *et al*. [19] found similar findings for Java build systems. Both sets of work suggested that studying build systems is important and that they consume a fair amount of the maintenance effort for any system.

7.2.2. Build system variability. It is our understanding that Berger *et al*. [8] were the first to discuss variability in the Linux build system. They showed that the extraction of presence conditions of source code files from Makefiles is feasible and extracted them for the x86 architecture in Linux and for all of FreeBSD. Our quantitative analysis of KBUILD is based on all Linux CPU architectures over a longitudinal study and not solely on the x86 architecture. We also show the effect of these constraints on the variability of the final compiled kernel image through the anomalies we detect.

Dietrich *et al*. [9] found that KBUILD v3.1 alone uses almost 50% of the KCONFIG features in Linux. Our work is different in that we perform an evolutionary study with several releases of KBUILD. We also adapt previous metrics used to measure CPP variability to customize them for KBUILD. This allows for standardization of future quantitative analysis of variability. Additionally, we also analyze the

DOI: 10.1002/smr

complexity of constraints and GRAN of control within KBUILD. This provides a better overall picture of the variability in KBUILD and how it contributes to the configurability of the whole Linux kernel.

7.2.3. Extracting and using KBUILD variability constraints. To the best of our knowledge, our previous work [11], which this paper is an extended version of, was the first to analyze the effect of variability in KBUILD on anomalies. Recently, the UNDERTAKER team have developed their own make constraint extractor, GOLEM [10]. Their variability extraction approach is based on running an actual Linux build using different configurations and probing it to see which files become built. The advantage of their approach versus a static parsing approach such as ours and that of Berger *et al*. [8] is that they avoid explicitly analyzing the complicated syntax and special cases that occur in KBUILD. Currently, GOLEM takes about 90 min to extract the constraints of a single CPU architecture in Linux versus about 51 s for all architectures by MAKEX. Arguably, performance is not everything, but such a high running time may affect the practicality of the approach. It is also not clear yet if such a probing-based approach would catch all the complex constraints in KBUILD. For example, it would not be able to correctly identify constraints containing negations (i.e., that a feature should not be present) because they rely on incremental addition of features to the feature set and then probing the build system on what files will be built. It is also difficult to correctly identify disjunctions in some cases depending on the order the features become probed by. Thus, it seems that both static and probing/dynamic-based approaches have their limitations. However, the goal of our work is not to determine the most accurate parsing for Makefiles, but rather to clarify the role of build systems in variability support so that it is recognized in future variability research.

8. CONCLUSIONS AND FUTURE WORK

The goal of this paper is to explore the role of build systems in variability implementation. To do so, we have presented a case study of variability in Linux's build system, KBUILD. We extracted the presence conditions of source files from KBUILD (make constraints) using our developed extractor, MAKEX, and used this information to provide two sets of results: a quantitative analysis of KBUILD variability and the effect of such constraints on detecting variability anomalies in Linux. We performed both analyses on the latest 10 stable releases of Linux.

We have found that KBUILD plays a key role in Linux's variability implementation: 63% of configuration features in Linux are used by KBUILD, and 92% of source files are conditionally compiled in KBUILD. We have also found that 76% of source files have only one feature in their presence condition, and that 78% of features control exactly one file. This suggests that in most cases, there is a one to one mapping between a user-selected configuration feature and the source file it controls.

To examine the effect of the extracted make constraints on variability anomalies, we extended UNDERTAKER [1] to use our extracted constraints to detect variability anomalies in Linux. By including the make constraints in the analysis, we detected additional anomalies at the code block level and contrasted them with those detected by the original UNDERTAKER tool. We also performed the analysis at the file level and discovered several dead files due to missing feature definitions.

The anomalies we detected suggest a need for automatic anomaly detection tools (especially at the block level) that are easy to use by the developers and can be integrated in their regular development cycle. Additionally, because #IFDEF usage is often discouraged [20] and our findings have suggested that developers have less trouble managing variability at the file level in KBUILD, it might be recommended to limit variability at the file level. Although such a limitation may not be feasible to implement now in a long established system such as Linux, it might be useful for future systems supporting variability to keep in mind. Such a design will also enforce more modularity in the system that makes a system more maintainable.

We hope that our investigation of KBUILD opens the door for more research in the variability of build systems. Our case study of Linux was a step in that direction. Studying software systems besides Linux will help in clarifying the general role build systems play in implementing variability.

DOI: 10.1002/smr

ACKNOWLEDGMENTS

Thanks to Daniel Lohmann for his feedback on the conference version of this work [11]. Reinhard Tartler and Christian Dietrich have answered many of our questions about using UNDERTAKER. Many thanks to our anonymous reviewers who provided us with helpful suggestions to improve this paper.

REFERENCES

1. Tartler R, Lohmann D, Sincero J, Schröder-Preikschat W. Feature consistency in compile-time configurable system software: facing the linux 10,000 feature problem. *EuroSys '11: Proceedings of the 6th Conference on Computer Systems*, of ACM SIGOPS EC (ed.). ACM: New York, NY, USA, 2011. DOI:10.1145/1966445.1966451.
2. She S, Lotufo R, Berger T, Wąsowski A, Czarnecki K. The Variability Model of the Linux Kernel. *VaMoS 2010: Proceedings of the Fourth International Workshop on Variability Modeling of Software-intensive Systems*, 2010.
3. Sincero J, Tartler R, Lohmann D, Schröder-Preikschat W. Efficient extraction and analysis of preprocessor-based variability. *GPCE '10: Proceedings of the Ninth International Conference on Generative Programming and Component Engineering*. ACM: New York, NY, USA, 2010; 33–42. DOI:10.1145/1868294.1868300.
4. Zengler C, Küchlin W. Encoding the Linux kernel configuration in propositional logic. *ECAI '10: Proceedings of 19th European Conference on Artificial Intelligence (Workshop on Configuration)*, 2010; 51–56.
5. Lotufo R, She S, Berger T, Czarnecki K, Wąsowski A. Evolution of the Linux kernel variability model. *SPLC'10: Proceedings of the 14th International Conference on Software Product Lines: Going Beyond*, Springer-Verlag: Berlin, Heidelberg, 2010; 136–150.
6. Liebig J, Apel S, Lengauer C, Kästner C, Schulze M. An analysis of the variability in forty preprocessor-based software product lines. *ICSE '10: Proceedings of the 32nd International Conference on Software Engineering*, vol. 1, 2010; 105–114. DOI:10.1145/1806799.1806819.
7. Nadi S, Holt R. *Make it or break it: Mining anomalies in Linux Kbuild*. *WCRE '11: Proceedings of the 18th Working Conference on Reverse Engineering*, 2011. DOI:10.1109/WCRE.2011.46.
8. Berger T, She S, Lotufo R, Czarnecki K, Wąsowski A. Feature-to-code mapping in two large product lines. *Software Product Lines: Going Beyond* 2010; **6287**:498–499.
9. Dietrich C, Tartler R, Schröder-Preikschat W, Lohmann D. Understanding Linux feature distribution. *AOSD-MISS '12: Proceedings of the 2nd AOSD Workshop on Modularity in Systems Software*, 2012. DOI:10.1145/2162024.2162030.
10. Dietrich C, Tartler R, Schröder-Preikschat W, Lohmann D. A robust approach for variability extraction from the linux build system. *SPLC '12: Proceedings of the 16th International Software Product Line Conference (to appear)*, 2012.
11. Nadi S, Holt R. Mining Kbuild to detect variability anomalies in Linux. *CSMR '12: Proc. of the 16th European Conference on Maintenance and Reengineering*, 2012.
12. Undertaker. Available at: http://vamos.informatik.uni-erlangen.de/trac/undertaker [Accessed 1 July 2012].
13. Adams B, Tromp H, De Schutter K, De Meuter W. Design recovery and maintenance of build systems. *ICSM '07: Proceedings of the 23rd IEEE International Conference on Software Maintenance*, 2007; 114–123.
14. Kästner C, Giarrusso P, Rendel T, Erdweg S, Ostermann K, Berger T. Variability-aware parsing in the presence of lexical macros and conditional compilation. *OOPSLA '11: Proceedings of the 2011 ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, 2011.
15. Sincero J, Schröder-Preikschat W. The Linux kernel configurator as a feature modeling tool. *SPLC'08: Proceedings of the 12th International Conference on Software Product Lines: Going Beyond*, 2008; 257–260.
16. Berger T, She S, Lotufo R, Wąsowski A, Czarnecki K. Variability modeling in the real: a perspective from the operating systems domain. *ASE '10: Proceedings of the IEEE/ACM international conference on Automated software engineering*. ACM: New York, NY, USA, 2010; 73–82. DOI:10.1145/1858996.1859010.
17. She S, Lotufo R, Berger T, Wąsowski A, Czarnecki K. Reverse engineering feature models. *ICSE '11: Proceedings of the 33rd International Conference on Software Engineering*. ACM: New York, NY, USA, 2011; 461–470.
18. Adams B, De Schutter K, Tromp H, De Meuter W. The evolution of the Linux build system. *EVOL '07: Proceedings of the Third International ERCIM Symposium on Software Evolution*, vol. 8, 2007.
19. McIntosh S, Adams B, Hassan A. The evolution of Java build systems. *Empirical Software Engineering* 2011; **17**:1–31.
20. Spencer H, Collyer G. #ifdef considered harmful, or portability experience with C news. *Proceedings of the Summer 1992 USENIX Conference*, Adams R (ed.), USENIX Association: Berkeley, CA, 1992; 185–198.

DOI: 10.1002/smr

AUTHORS' BIOGRAPHIES:

Sarah Nadi is a PhD candidate at the University of Waterloo working with Prof. Ric Holt. Her research interests include using mining software repositories techniques to provide decision support for difference stages in the software development cycle. She currently focuses on studying software variability by analyzing the different information provided by various variability artifacts and finding methods to ensure their consistency. Her work includes studying variability implementation in build systems, detecting variability anomalies, and mining historic artifacts to find the causes and fixes of these anomalies.

Ric Holt is a professor at the University of Waterloo, where his research interests include software architecture and mining software repositories (MSR). His architectural visualizations have included Linux, Mozilla (Netscape), IBM's TOBEY code generator, and Apache. He is the developer of the Grok relational language. His previous research includes foundational work on deadlock, development of a number of compilers and compilation techniques, development of one of the first Unix clones, and authoring a dozen books on programming and operating systems. He is co-designer of the Turing programming language.

DOI: 10.1002/smr