Mining Kbuild to Detect Variability Anomalies in Linux

Sarah Nadi and Ric Holt
David R. Cheriton School of Computer Science
University of Waterloo
Ontario, Canada
Email: snadi, holt@uwaterloo.ca

***Abstract*—The Linux kernel is extensively specialized or configured so that it can be used for many purposes. This variability is implemented by means of three distinct artifacts: source code files, Kconfig (configuration) files, and Makefiles. Any inconsistencies between these three can lead to undesirable anomalies which can lead to increased maintenance efforts or decreased reliability. This paper extends published work that had found anomalies (dead and undead code blocks) by concentrating largely on code and Kconfig files. We detect further anomalies in the Linux kernel when we also consider the Makefiles. At the level of code blocks, our work exposes many additional anomalies — more than we could study manually. We found that when we lift the level from code blocks to code files, the detected anomalies became easier to study and understand and thus more useful to the developer. By means of examples, we illustrate how the anomalies we detect can lead to undesired behavior. We show how, over time, developers tend to find and delete such anomalies. We suggest that automatic detection of such anomalies has the potential to decrease maintenance efforts and increase reliability.**

I. Introduction

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. Variability (or configurability) means that the software allows users to customize it for their needs by choosing different sets of features. For example, one Linux user may choose to compile a kernel with USB support while another user may choose not to. 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 which supports over 10,000 features, and serves millions of users.

Variability in Linux depends on three distinct artifacts: source code files (*code space*), Kconfig files (*kconfig space*), and Kbuild Makefiles (*make space*) [12], [17]. Having related information divided over three separate artifacts raises the problem of inconsistencies in the variability of the Linux kernel. This can cause anomalies which may lead to decreased reliability and increased maintenance effort.

Previous work by Tartler et al. [17] discovered anomalies in the Linux kernel by studying the constraints in the source code files (code space), and Kconfig files (kconfig space) through their UNDERTAKER[1] tool. Their work focuses on finding anomalies at the level of the code blocks using a SAT (satisfiability) solver. However, they do not include the information in the Makefiles (make space) as part of their analysis. Previous work [12] has shown the importance of the make space and its role in implementing variability in the Linux kernel. In the work reported in this paper, we extend the UNDERTAKER tool to consider the constraints in the make space during the detection of anomalies. We focus on two types of anomalies: *dead* and *undead* artifacts. A dead artifact is one that is never present in any variant of the Linux kernel. An undead artifact is one which is always present in every variant of the Linux kernel.

We start by working at the code block level which is the same level of analysis as the UNDERTAKER tool. By considering the additional constraints imposed by the make space, we discover many additional dead and undead code blocks that were not found in the original analysis. However, the number of additional anomalies is too large to be of help to a developer. Additionally, this large number along with the complicated nature of boolean formulas makes it hard for us to analyze the results. This leads us to suspect that the code block level might not be the appropriate artifact level to use when studying Kbuild. The fact that Makefiles deal with code files and not with code blocks further suggests to us that the right level may be files, not blocks. Our findings indeed show that when we lift the level from code blocks to code files, the detected anomalies become easier to study and understand, and thus are more useful to the developer.

Figure 1 shows the overview of our anomaly detection process at both levels of artifacts. The constraints from all three spaces are first extracted from the raw artifacts and represented as propositional logic (*Extracting Constraints* stage). Source code and Kconfig parsing are already present in the UNDERTAKER tool. We develop the Makefile parser to translate the Makefiles in the Linux kernel into propositional logic constraints. The constraints from the three spaces are then combined into one of the four formulas shown in the *Building Formulas* stage in Figure 1. Depending on whether we are working on the code block or code file level, the appropriate formula will be used. The details of these formulas are explained in Section IV. In the final stage, *Detecting Anomalies*, this formula is presented to a SAT solver engine, and a report is produced if anomalies are detected.

[1] http://vamos.informatik.uni-erlangen.de/trac/undertaker

1534-5351/12 $26.00 © 2012 IEEE
DOI 10.1109/CSMR.2012.21

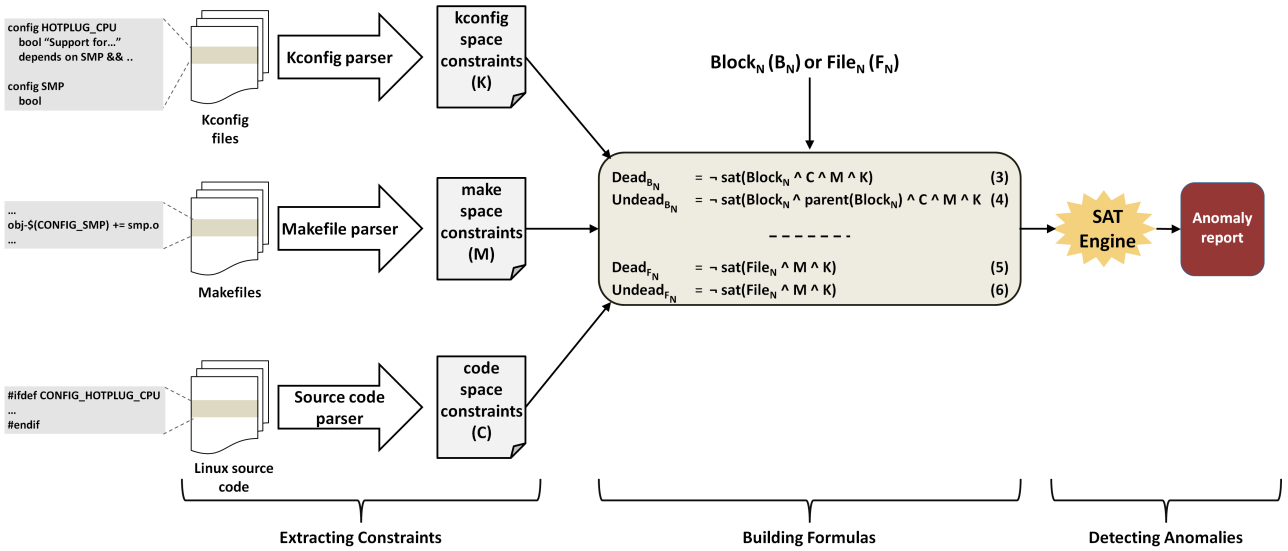


Fig. 1. Anomaly detection process

In this paper, we present both levels of analyses we performed (code block level and file level) over 11 releases of the Linux kernel. Through examples, we illustrate how the detected anomalies can lead to undesired behavior. We use developers' commit comments to analyze how developers fix such anomalies. The majority of anomalies detected in this work are hard to find manually which suggests the need for automatic detection. We suggest that automatic detection of such anomalies has the potential to decrease maintenance and increase reliability. Key contributions of this paper include:

1) Extracting the make space constraints from all Makefiles in the Linux kernel and encoding them as propositional logic formulas.
2) Using make space constraints to detect anomalies at both the file and block level in the Linux kernel.

The rest of this paper is organized as follows. Section II provides background information about variability in the Linux kernel. Section III describes how we extract the make space constraints from Kbuild to encode them as propositional logic. Section IV explains the propositional formulas we use to detect dead and undead code blocks and files. Sections V and VI describe the detection of anomalies at the block level and at the file level respectively. Section VII discusses possible threats to the validity of our work. Section VIII presents related work, and Section IX concludes this paper.

II. Background: Variability in the Linux Kernel

Figure 2 illustrates the process of building the Linux kernel. The three artifacts which affect variability, and accordingly control what gets compiled, are shown as the files on the left.

The Linux kernel is configured through tools that read the Kconfig files and display them to the user in a menu format. This produces two files containing the user's feature selection: the `.config` file used by Kbuild, and the `autoconf.h` file used by the `gcc` compiler. Entries in the `.config` file define environment variables that are then used in the Makefiles to control which files get compiled by `gcc` into the final kernel image, `vmlinux` (shown at the bottom right of the figure). The Makefiles force the header file `autconf.h` to be included in all source code compilations. Accordingly, the features defined in `autoconf.h` determine which parts of the code are actually compiled based on the C Preprocessor directives (`#ifdef`) present in the source code.

Details of how variability is implemented in each of the three spaces involved in the build process are provided in the following sections.

A. Configuration Space (Kconfig)

Kconfig files describe the various features of the Linux kernel. They specify configuration options and their interdependencies. Each feature has a `config` entry in a Kconfig file.

Listing 1 shows examples of Kconfig entries. The first entry defines a feature called `USB_DEVICEFS` of type `bool` (i.e., it can have values `y` for yes or `n` for no), and which depends on the `USB` feature (not shown). This means that it cannot be selected unless `USB` is also selected. The second entry shows another feature, `USB_SERIAL_CYPRESS_M8`, of type `tristate`. This means that besides the values `y` and `n`, this feature can take on the value `m` which means it is compiled as a loadable module.

Listing 1 Kconfig example

```
config USB_DEVICEFS
    bool
    depends on USB

config USB_SERIAL_CYPRESS_M8
    tristate
```

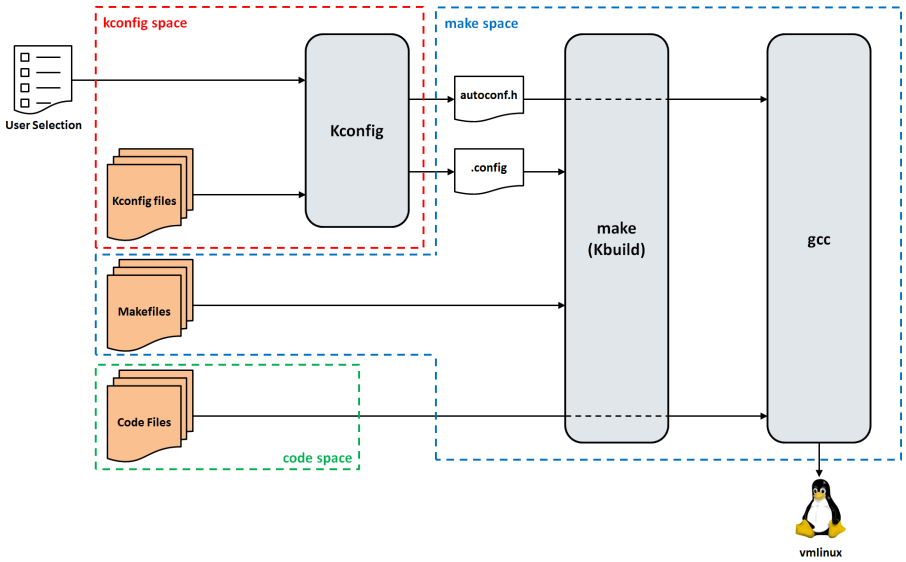


Fig. 2. Linux kernel build process

Note that whenever Kconfig features are referenced in the code (through preprocessor directives) or in the Makefiles, they have a `CONFIG_` prefix attached to their name. For example, if the `USB` feature is to be used in a Makefile, it is referred to as `CONFIG_USB`. Additional details about Kconfig files can be found in other work [8], [9], [14].

B. Code Space

In order to have certain functionalities present only when their respective features are chosen, corresponding blocks in the source code are conditionally compiled. This is done through C Preprocessor (CPP) directives such as `#ifdef`, `#ifndef`, `#elif`, and `#else`. Listing 2 shows an example of a conditionally compiled code block.

Listing 2 Variability in the Source Code

```
#ifdef CONFIG_FOO
//Block 1
#else
//Block 2
#endif
```

In this example, Block 1 will be compiled only if feature `FOO` is selected (i.e., it is defined in the generated `autoconf.h` header file). In this example, we can tell that this is a Kconfig feature from the `CONFIG_` prefix. On the other hand, Block 2 will only be compiled if feature `FOO` is not selected (i.e., it is not defined in the generated `autoconf.h` header file). Details about CPP directives and how they affect variability can be found in the work by Sincero et al. [16].

C. Make Space (Kbuild)

Most source code directories of the Linux kernel tree contain a Makefile responsible for compiling the files in that directory [7]. All Makefiles in the system have access to the features defined in the `.config` file which comes from the Linux kernel configuration process. The build process starts with the Makefile in the root directory of the Linux kernel tree and then recursively descends into the Makefiles of the subdirectories according to what is specified in each Makefile.

Listing 3 Example Makefile in a directory dir1

```
1.  obj-y += bar.o
2.  obj-y += dir2/
3.  obj-$(CONFIG_FOOBAR) += foobar.o
4.  obj-$(CONFIG_FOO)    += foo.o
5.  foo-y := foo1.o foo2.o
```

Listing 3 shows a sample Makefile. This Makefile shows entries that contribute to the variability of the kernel. For each directory, there is an `obj-y` variable which contains a list of files that are to be compiled and linked. The various entries in the Makefile append more files to this list. In each directory, the files added to the `obj-y` list are compiled and linked into a `built-in.o` file. All `built-in.o` files are then linked into the final built kernel image, `vmlinux`.

Listing 3 show that there are three different types of entries that can be added to the `obj-y` list: object files (e.g., `bar.o`), directories (e.g., `dir2`), and composite objects (e.g., `foo.o`). An object file is created by compiling a source file with the corresponding name (e.g., `bar.c` compiles into `bar.o` in Line 1). A directory entry indicates that the Makefile in that directory will be visited and that it will specify which files in that directory will be compiled (Line 2). A composite object combines two or more files together to be compiled into one object file and then this combined object file can be added to the `obj-y` list (Lines 4 and 5 in Listing 3).

Each of these types of entries can also be conditional. That

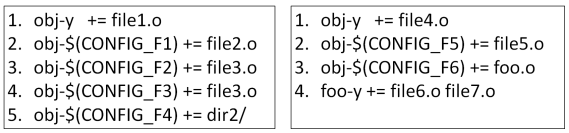


Fig. 3. Part a) shows a Makefile in a directory called dir1. Part b) shows a Makefile in a subdirectory of dir1 called dir2. Adding dir2/ to the obj-y list in Line 5 causes the Makefile in dir2 to be visited.

Listing 4 Make space constraints for example shown in Figure 3. These constraints show the config features each code file depends on.

```
1.  dir1/file1.c
2.  dir1/file2.c <-> CONFIG_F1
3.  dir1/file3.c <-> CONFIG_F2 || CONFIG_F3
4.  dir1/dir2/file4.c <-> CONFIG_F4
5.  dir1/dir2/file5.c <-> CONFIG_F4 && CONFIG_F5
6.  dir1/dir2/file6.c <-> CONFIG_F4 && CONFIG_F6
7.  dir1/dir2/file7.c <-> CONFIG_F4 && CONFIG_F6
```

is, each will only be added to the `obj-y` list if the relevant feature is selected. For example, Line 3 in Listing 3 indicates that `foobar.o` will be added to the `obj-y` list only if `CONFIG_FOOBAR`'s value is `y` (i.e., it is selected by the user). Note that there is also an `obj-m` list which contains the files that will be compiled as loadable modules. Conditional entries added to that list must have their respective feature value be `m`. More details about Kbuild's notation can be found in previous work [12].

III. Parsing Makefiles to Extract Make Space Constraints (M)

Figure 1 shows that the first step in the anomaly detection process is extracting the constraints from the three spaces. The UNDERTAKER tool already parses the code [16], [17] and kconfig [17], [18] files to extract their constraints. Accordingly, we implement a Makefile parser in Java to extract these make space constraints for each architecture in the Linux kernel. In this section, we explain how this parsing occurs. We use an example of two Makefiles shown in Figure 3, and explain how we translate them into the corresponding make space constraints shown in Listing 4.

Our goal while parsing the Makefiles is to determine which files are conditionally added to the `obj-y` list according to the selection of certain features. We start from the Makefile in `dir1` shown in Figure 3a. Line 1 in this Makefile means that `file1.c` will always be compiled into `file1.o`. Accordingly, we will put an entry in the make space constraints showing that `file1.c` does not depend on anything. This is shown in Line 1 of Listing 4. Note that when writing out the constraints, we use the `.c` extension of the file name.

Line 2 of the Makefile in Figure 3a will result in the entry on Line 2 of Listing 4 which indicates that `file2.c` depends on `CONFIG_F1`. The notation "`<->`" indicates that `file2.c` will be compiled *if and only if* `CONFIG_F1` is selected. Lines 3 and 4 in the same Makefile both deal with `file3.o`. These lines mean that `file3.c` will be compiled either if `CONFIG_F2` *or* `CONFIG_F3` is defined. Accordingly, this translates to the constraints in Line 3 of Listing 4. Line 5 in Figure 3a means that `dir2` should be visited only if `CONFIG_F4` is defined. As mentioned in Section II-C, the Makefile in the subdirectory is the one responsible for which files get compiled there. We now move to the Makefile of `dir2` shown in Figure 3b.

Line 1 of the second Makefile simply adds `file4.o` to the `obj-y` list without constraints. However, we must remember that visiting `dir2` in the first place was dependent on `CONFIG_F4` being selected. This results in the constraints shown on Line 4 of Listing 4. Line 2 of the same Makefile adds `file5.o` to `obj-y` given that `CONFIG_F5` is selected. Again, we must consider the dependencies of visiting the whole directory in the first place. This means that `file5.c` actually depends on *both* `CONFIG_F4` and `CONFIG_F5` as shown in Line 5 of Listing 4. Lines 3 and 4 of that Makefile show a composite object consisting of files `file6.o` and `file7.o` and depending on `CONFIG_F6`. Since there are two files, this results in the entries of Lines 6 and 7 in Listing 4 with both files having the same constraints.

IV. Propositional Formulas for Detecting Dead and Undead Artifacts

Given that the constraints of the three spaces have been extracted, we now present the propositional formulas that use these constraints to detect dead and undead artifacts. These are the four formulas shown in Figure 1. We now explain how these formulas were derived, and the different cases in which they detect anomalies. We first explain the formulas used in the original UNDERTAKER analysis, and then the modified versions we use in our analysis.

A. Code Block Level Anomalies in Original UNDERTAKER

The UNDERTAKER tool extracts preprocessor based variability in the Linux source code in order to determine the *presence condition* [16] of each code block, and encodes it as propositional logic. The presence condition of a code block is the set of constraints (in terms of feature selections) that must be satisfied in order for this code block to get compiled. Additionally, it identifies the dependencies in the Kconfig files and also represents them in propositional logic [15], [18]. To find anomalies, it checks the satisfiability of the combination of these constraints for each code block. Currently, the UNDERTAKER tool only considers the code space (denoted by *C*) and the kconfig space (denoted by *K*), and uses Formulas 1 and 2 to check for dead and undead code blocks respectively.

Tartler et al. [17] use Formula 1 to define a code block, $Block_N$ (B_N), as dead if there is never a case where it can be selected. In terms of propositional logic, this means that we can never satisfy the combination of constraints imposed by the code and kconfig spaces and have $Block_N$ present at the same time [17].

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

Similarly, Formula 2 defines an undead code block, $Block_N$ (B_N), as one which is always present whenever its parent block is present [17]. If this is the case, then this code block's presence is not really variable since it always gets compiled if its parent block is compiled. This is shown in Formula 2 where a block is undead if it can never be deselected in the presence of its parent while satisfying the combination of constraints in the code and kconfig spaces.

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

B. Code Block Level Anomalies with Make Space

The analysis at the code block level in UNDERTAKER ignores the constraints enforced in the Makefiles (the make space constraints). In our analysis, we add these constraints, and modify Formulas 1 and 2 to those shown in Formulas 3 and 4. We denote the make space constraints as M. By adding the make space constraints, we use Formula 3 to define a code block as dead if it can never be present while satisfying the code and kconfig constraints along with the make constraints.

$$Dead_{B_N} = \neg sat(Block_N \wedge C \wedge M \wedge K) \quad (3)$$

Similarly, we define a block, $Block_N$ (B_N), as undead in Formula 4 if we can never find a case where $Block_N$ is not present, but its parent is present while still satisfying the constraints in all three spaces.

$$Undead_{B_N} = \neg sat(\neg Block_N \wedge parent(Block_N) \wedge C \wedge M \wedge K) \quad (4)$$

Given Formulas 3 and 4, it follows that there are six different reasons or causes for the formulas to detect anomalies. These six causes are summarized in Table I. Code block anomalies may arise from conflicts or inconsistencies between the code space and one or more of the other two spaces. This results in four possible causes of conflicts: *code*, *code-make*, *code-kconfig*, and *code-make-kconfig*. Additionally, when considering the kconfig space, another category of anomalies arises, that of *missing* definitions. That is, one or more of the features used in the formula may not have a definition in the kconfig constraints. This adds two more reasons for conflicts: *code-kconfig missing* and *code-make-kconfig missing*. All six cases with their descriptions are shown in Table I.

Cause	Description
code	Conflicting code constraints.
code-make	Code constraints are not consistent with constraints in Makefiles.
code-kconfig	Code constraints are not consistent with constraints in Kconfig.
code-make-kconfig	The combination of constraints in the three spaces are conflicting.
code-kconfig missing	Code constraints are not consistent with Kconfig constraints because certain features used in the code are not defined in the Kconfig files and are, therefore, always false.
code-make-kconfig missing	The combination of constraints in the three spaces are conflicting because certain features used in the compilation constraints are not defined in the Kconfig files, and are therefore always false.

TABLE I
CAUSES OF CODE BLOCK VARIABILITY ANOMALIES.

C. Code File Level Anomalies

We now move from the level of code blocks to the level of code files. To work at the file level, we again modify the original UNDERTAKER formulas (Formulas 1 and 2) by replacing the block with the code file, removing the code constraints (since we are no longer working on the block level), and adding the make constraints. This is shown in Formulas 5 and 6.

In Formula 5, we define a file as dead if it can never be present (i.e., will never get compiled) while satisfying the combination of constraints in the make space and the kconfig space.

$$Dead_{F_N} = \neg sat(File_N \wedge M \wedge K) \quad (5)$$

Similarly, Formula 6 defines a file as undead if we cannot find a case where it does not get compiled while satisfying the make space and kconfig space constraints.

$$Undead_{F_N} = \neg sat(\neg File_N \wedge M \wedge K) \quad (6)$$

Since there are only two spaces involved in these formulas, anomalies can either arise because of conflicts between the two spaces (i.e., make-kconfig) or because of missing feature definitions (i.e., make-kconfig missing).

V. DETECTING ANOMALIES AT THE CODE BLOCK LEVEL

A. Overview

We now discuss the results of our analysis in terms of detecting dead and undead code blocks in the Linux kernel. To determine the effect of adding the make constraints to the analysis of dead and undead code blocks, we begin by running the UNDERTAKER tool version 1.1 (i.e., using $C \wedge K$, but not M, as shown in Formulas 1 and 2) over 11 kernel releases, 2.6.30 to 3.0. Then, we run our modified version of the tool (i.e., using $C \wedge K \wedge M$ as shown in Formulas 3 and 4) over the same 11 releases. We then compare the anomalies detected in each case, and collect statistics about them. We provide these statistics below as well as some illustrative examples of the anomalies we detected. For some of the examples, we also show patches that we found in the Linux git repository which fixed these anomalies.

B. Results for Code Block Level

Figure 4 shows the total number of code blocks having anomalies detected with and without considering the make space (M). We can see that with all three spaces, $C \wedge M \wedge K$, we detect more anomalies than with just $C \wedge K$. Although the total number of anomalies shown in Figure 4 seem to

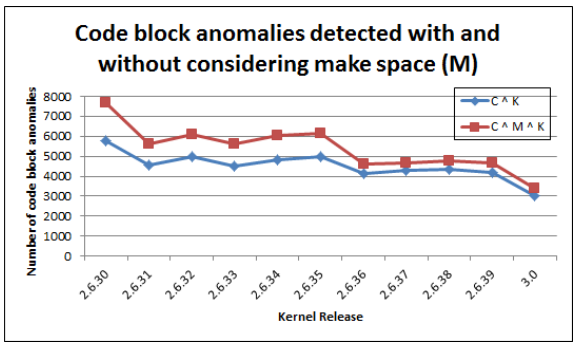


Fig. 4. Total code block anomalies detected with and without the make space (M)

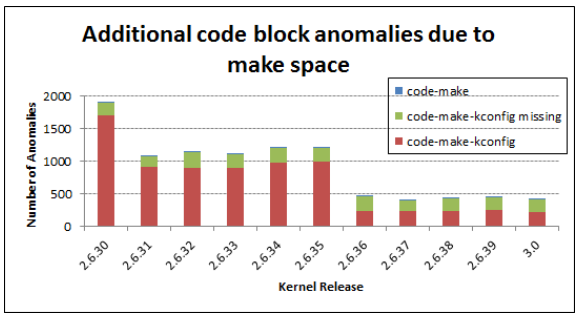


Fig. 5. Additional anomalies caused by adding the make space constraints to the analysis. The code-make-kconfig represents the highest percentage of additional anomalies detected, while the code-make represents the smallest percentage of anomalies detected.

have decreased over time in both cases, there are still over 3000 anomalies present in release 3.0 which still represents a considerable amount of anomalies.

We are interested in examining the additional anomalies that are detected when the make space constraints were added. Figure 5 focuses on these additional anomalies and divides them into the three anomaly causes involving the make space constraints discussed in Section IV (i.e., code-make, code-make-kconfig, and code-make-kconfig missing). The figure shows that considering the make constraints allows us to detect a considerable number of additional anomalies in each release, ranging from 400 additional anomalies up to 1900.

In Figure 5, we can see that the anomalies caused by conflicts between all three spaces (code-make-kconfig) represent the highest percentage of additional anomalies detected. Since detecting this category of anomalies requires solving a complex satisfiability formula, it suggests that these anomalies are hard to find manually by the developer, and that having automated tools to detect them is important. In order to understand the nature of these additional anomalies, we provide illustrating examples from each cause.

Analysis of code-make Anomalies

Code-make anomalies are those caused by a conflict between the constraints in the code space and those in the make space. In the original UNDERTAKER, such anomalies are not detected since the Makefiles are not considered in the analysis. In our modified analysis that considers Makefiles, we find eight distinct code-make anomalies over the eleven releases we examine. Two of them were introduced in release 2.6.37 while the remaining anomalies already existed in release 2.6.30 (the first release in our analysis). We now discuss two examples of these anomalies.

The first is an anomaly that got fixed in release 2.6.31. This anomaly is a dead code block in file, `./arch/x86/kernel/acpi/boot.c`. The block is dead because of conflicts in its constraints. The code space constraints indicate that `CONFIG_ACPI` should not be selected while the make space constraints indicate that `CONFIG_ACPI` needs to be selected for the file to compile. This anomaly gets fixed in release 2.6.31 with the following comment by the developer "Testing `CONFIG_ACPI` inside boot.c is a waste of text, since boot.c is built only when `CONFIG_ACPI=y`." [4].

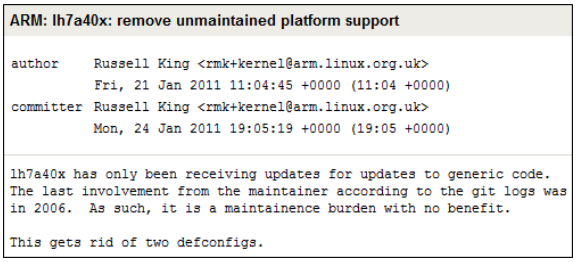
The other example is of two anomalies (one dead block and one undead block) introduced in release 2.6.37 in file `./arch/sparc/kernel/jump_label.c`. These are also caused by the redundant check of `CONFIG_SPARC64` on which the file itself depends on in the Makefiles. However, these two anomalies are still present in the last release we examine (release 3.0).

Analysis of code-make-kconfig 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.

For example, `./drivers/serial/bfin_5xx.c` (which later got moved to `./drivers/tty/serial/bfin_5xx.c`) has a dead block since release 2.6.32. From the code constraints, we find that this code block depends on `CONFIG_SERIAL_BFIN_MODULE` while the make space constraints indicate that the code file itself depends on `CONFIG_SERIAL_BFIN`. However, the kconfig space constraints show that `CONFIG_SERIAL_BFIN` depends on `!CONFIG_SERIAL_BFIN_MODULE`. Thus, we have a conflict here since the same feature (`CONFIG_SERIAL_BFIN_MODULE`) can never be selected and not selected at the same time. This means this code block is dead. This anomaly still exists in the last release we examine.

Some anomalies get fixed by simply removing the code block since it is not used. For example, the file `./drivers/usb/gadget/lh7a40x_udc.c` has 11 dead code-make-kconfig anomalies reported. These anomalies no longer exist after the file was removed. Figure 6 shows the `git` commit removing the file. This commit is interesting since the developer explicitly states that leaving unused code in the kernel is a maintenance burden. This suggests that



```
ARM: lh7a40x: remove unmaintained platform support

author    Russell King <rmk+kernel@arm.linux.org.uk>
          Fri, 21 Jan 2011 11:04:45 +0000 (11:04 +0000)
committer Russell King <rmk+kernel@arm.linux.org.uk>
          Mon, 24 Jan 2011 19:05:19 +0000 (19:05 +0000)

lh7a40x has only been receiving updates for updates to generic code.
The last involvement from the maintainer according to the git logs was
in 2006.  As such, it is a maintainence burden with no benefit.

This gets rid of two defconfigs.
```

Fig. 6. Commit that fixes a dead code-make-kconfig anomaly

automatically detecting these dead code anomalies is useful. This is especially true in cases where manual inspection of the code will not easily identify a code block as dead.

Analysis of code-make-kconfig missing Anomalies

Anomalies in this category are caused by undefined Kconfig features that appear in the boolean formula. We can see in Figure 5 that many of the anomalies detected fall in this category. As an example, 19 different dead code-make-kconfig missing anomalies were detected in several files in the `./drivers/net/stmmac` directory. The make space constraints indicate that `CONFIG_STMMAC_ETH` needs to be defined for the files to compile. However, `CONFIG_STMMAC_ETH` depends on another feature, `CONFIG_CPU_SUBTYPE_ST40`, which has no Kconfig definition. Thus, all these code blocks are reported as code-make-kconfig missing. This is fixed in release 2.6.37 by removing the dependency on the undefined feature, `CONFIG_CPU_SUBTYPE_ST40` as shown in the commit comment in Figure 7. This commit fixes all 19 anomalies.

VI. Detecting Anomalies at the File Level

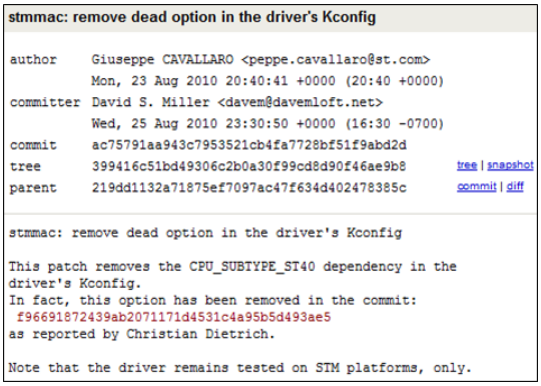
A. Overview

After carrying out analysis at the code block level, we realize that presenting thousands of anomalies to a developer is impractical and overwhelming. Since Makefiles deal directly with the code files, we change our analysis to identify dead and undead code files instead of code blocks using Formulas 5 and 6 (see Section IV). We run the modified analysis on the same 11 releases, 2.6.30 to the latest release 3.0.

B. Results for Code File Level

From this analysis, we detect several dead code files, but do not find any undead files. We observe that the dead files are either caused by undefined (missing) configuration features or by conflicts in the kconfig and make constraints. Table II summarizes the number of make-kconfig and make-kconfig missing anomalies detected in each release. We can see that the number of anomalies detected at the file level in each release is much smaller when compared to the number of code block anomalies in the previous section, and thus more manageable.

We analyze the anomalies in Table II over the 11 releases, and find that there is a total of 203 unique anomalies. This indicates that some anomalies exist through several releases.



```
stmmac: remove dead option in the driver's Kconfig

author    Giuseppe CAVALLARO <peppe.cavallaro@st.com>
          Mon, 23 Aug 2010 20:40:41 +0000 (20:40 +0000)
committer David S. Miller <davem@davemloft.net>
          Wed, 25 Aug 2010 23:30:50 +0000 (16:30 -0700)
commit    ac75791aa943c7953521cb4fa7728bf51f9abd2d
tree      399416c51bd49306c2b0a30f99cd8d90f46ae9b8   tree | snapshot
parent    219dd1132a71875ef7097ac47f634d402478385c   commit | diff

stmmac: remove dead option in the driver's Kconfig

This patch removes the CPU_SUBTYPE_ST40 dependency in the
driver's Kconfig.
In fact, this option has been removed in the commit:
 f96691872439ab2071171d4531c4a95b5d493ae5
as reported by Christian Dietrich.

Note that the driver remains tested on STM platforms, only.
```

Fig. 7. This commit removes the dependency on missing feature CONFIG_CPU_SUBTYPE_ST40 which resolves 19 code-make-kconfig missing anomalies.

Release	make-kconfig missing	make-kconfig	Total
2.6.30	78	13	91
2.6.31	65	13	78
2.6.32	102	14	116
2.6.33	94	12	112
2.6.34	91	15	107
2.6.35	97	15	112
2.6.36	96	15	111
2.6.37	71	21	92
2.6.38	82	23	105
2.6.39	85	23	108
3.0	84	21	111

TABLE II
DEAD FILES FOUND BY EXAMINING THE MAKE AND KCONFIG CONSTRAINTS. ONLY TWO CATEGORIES OF ANOMALIES ARISE: MAKE-KCONFIG AND MAKE-KCONFIG MISSING.

We analyze their evolution and the rate at which anomalies get introduced and fixed in each release. Figure 8 shows this evolution by plotting the number of make-kconfig and make-kconfig missing anomalies introduced and fixed in each release.

The first column in Table II and the darker columns in Figure 8 show that most of the dead files are caused by missing definitions of Kconfig features. Since anomalies still get introduced over time, as shown in Figure 8, this suggests a need for automatic anomaly detection. The fixed anomalies indicate that developers invest time in removing them. Additionally, these fixes provide us with insight about the nature of the anomalies, and how they get addressed. We now discuss examples for make-kconfig and make-kconfig missing anomalies which we detected, and how some of them get fixed.

Analysis of make-kconfig Anomalies

For a given code file, any conflict between the constraints in the make space (i.e., its dependencies stated in the Makefiles) and the kconfig space (i.e., related constraints imposed by Kconfig) will result in make-kconfig anomalies.

We inspect the make-kconfig anomalies listed in Table II

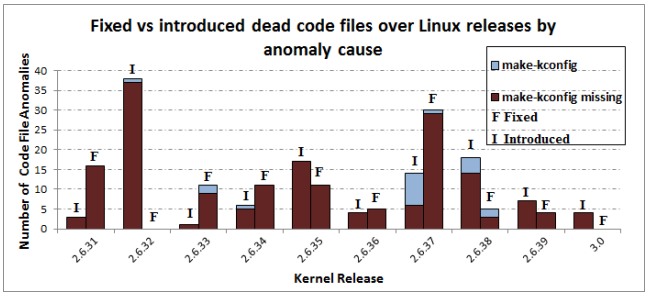


Fig. 8. Evolution of dead code files over time divided by cause. This evolution plot starts at release 2.6.31 to be able to correctly identify introduced and fixed anomalies in each release by comparing them to the previous release.

over the 11 kernel releases, and find that only 27 of them are unique anomalies. When we analyze these anomalies, we find that 25 of them are in the `arch/x86` directory. These 25 anomalies are caused by an indirect dependency on feature `CONFIG_X86_EXTENDED_PLATFORM` which has conflicts in its own dependencies. This causes the boolean formula to be unsatisfiable. Listing 5 shows the boolean formula for one of these 25 cases.

Our detailed analysis of the formula (which corresponds to Formula 5 in Section IV) is as follows. Line 1 of Listing 5 shows the name of the file we are examining. Line 3 shows the make space constraints, and Lines 4 - 16 show the kconfig constraints. Line 16 shows that as part of the kconfig constraints, `CONFIG_X86_EXTENDED_PLATFORM` is needed, and that it depends on `CONFIG_X86_64` and `CONFIG_X86_32`. From Line 10, we can see that `CONFIG_X86_64` depends on `CONFIG_64BIT` while Line 5 shows that `CONFIG_X86_32` depends on the *absence* of `CONFIG_64BIT`. This means that the constraints for `CONFIG_X86_EXTENDED_PLATFORM` can never be satisfied which in turn makes the formula unsatisfiable, and thus, the code file is dead.

We are surprised that the analysis detected anomalies in important code files in the x86 architecture, which is one of the most commonly used and rigorously maintained architectures in Linux, and that most of these anomalies remain in the system till release 3.0. We observe that a couple of the anomalies cease to exist, but this was mainly because the file itself was renamed or removed. Accordingly, we further investigate why these anomalies occur. As a result, we discover an error in the Kconfig parsing provided by the UNDERTAKER tool. The parsing error stems from how `if` clauses within Kconfig are handled in UNDERTAKER. If the same Kconfig feature is defined within two separate `if` clauses (e.g., `if X86_32` and `if X86_64`), the Kconfig parser in UNDERTAKER translates that to mean that this feature depends on *both* the conditions in the `if` clauses (i.e., `X86_32 and X86_64`). However, we believe this should instead translate in to the feature depending on *either* of these conditions (i.e., `X86_32 or X86_64`).

Analysis of make-kconfig missing Anomalies

We now present some of the dead code file anomalies that are caused by missing Kconfig definitions. These make-kconfig missing anomalies are caused by one or more features appearing in the propositional formula, but for which there is no definition in the Kconfig files. We discuss some of the cases we found below.

Listing 5 Example of a make-kconfig dead anomaly in file ./arch/x86/kernel/apic/es7000_32.c

```
1.  ./arch/x86/kernel/apic/es7000_32.c
2.  &&
3.  ( ./arch/x86/kernel/apic/es7000_32.c <-> ((CONFIG_X86_ES7000) ))
4.  &&
5.  (CONFIG_X86_32 -> ((!CONFIG_64BIT)))
6.  &&
7.  (CONFIG_X86_32_NON_STANDARD -> ((CONFIG_X86_32 && CONFIG_SMP
    && CONFIG_X86_EXTENDED_PLATFORM)))
9.  &&
10. (CONFIG_X86_64 -> ((CONFIG_64BIT)))
11. &&
12. (CONFIG_X86_BIGSMP -> ((CONFIG_X86_32 && CONFIG_SMP)))
13. &&
14. (CONFIG_X86_ES7000 -> ((CONFIG_X86_32_NON_STANDARD && CONFIG_X86_BIGSMP)))
15. &&
16. (CONFIG_X86_EXTENDED_PLATFORM -> ((CONFIG_X86_64) && (CONFIG_X86_32)))
```

Listing 6 Example of a make-kconfig missing dead anomaly in file ./drivers/macintosh/via-pmu-event.c. The last lines shows the four missing features.

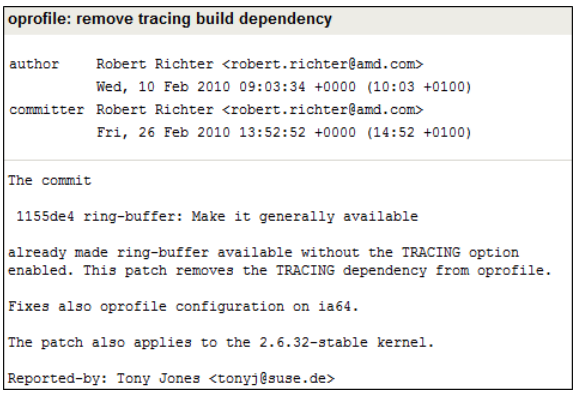
```
./drivers/macintosh/via-pmu-event.c
&&
( ./drivers/macintosh/via-pmu-event.c <-> (((CONFIG_MAC && CONFIG_ADB_PMU)) ))
&&
(CONFIG_ADB_PMU -> ((CONFIG_MACINTOSH_DRIVERS && CONFIG_PPC_PMAC)))
&&
(CONFIG_MACINTOSH_DRIVERS -> ((CONFIG_PPC || CONFIG_MAC || CONFIG_X86)))
&&
( ! ( CONFIG_MAC || CONFIG_PPC || CONFIG_PPC_PMAC || CONFIG_X86 ) )
```

Listing 6 shows the propositional formula for file `./drivers/macintosh/via-pmu-event.c`, which is dead because of missing features. The last line lists the features that are missing (four features in this case). Since these features are not defined, and can thus never be selected, the missing features are defaulted to false in the boolean formula to see if it can still be satisfied even though they are not defined. This also allows the developer to know the missing features from the anomaly reports.

Figure 9 shows an example of a commit which addresses a different missing feature definition. In this particular case, there are several files which indirectly depend on `CONFIG_TRACING`, while `CONFIG_TRACING` itself depends on several undefined features such as `CONFIG_TRACE_IRQFLAGS_SUPPORT`. Thus, removing the dependency on `CONFIG_TRACING` removes the missing features from the propositional formula. This fix alone eliminated 6 dead files in release 2.6.34. Removing a dead file anomaly means that this file will now get compiled in some variants of the kernel depending on the user's feature selection.

C. Discussion

Although working on the block level is more detailed, and is a natural extension to previous work [6], [18], working at a higher level, namely the file level, has provided us several advantages. A first advantage is that it yields more manageable results which is beneficial to both the developer and to us when analyzing the results. Developers do not want to be overwhelmed by thousands of anomalies if they can reach the same conclusion with less information. For



```
oprofile: remove tracing build dependency

author    Robert Richter <robert.richter@amd.com>
          Wed, 10 Feb 2010 09:03:34 +0000 (10:03 +0100)
committer Robert Richter <robert.richter@amd.com>
          Fri, 26 Feb 2010 13:52:52 +0000 (14:52 +0100)

The commit

 1155de4 ring-buffer: Make it generally available

already made ring-buffer available without the TRACING option
enabled. This patch removes the TRACING dependency from oprofile.

Fixes also oprofile configuration on ia64.

The patch also applies to the 2.6.32-stable kernel.

Reported-by: Tony Jones <tonyj@suse.de>
```

Fig. 9. This commit removes a dependency on TRACING. This resolves many make-kconfig missing anomalies, because TRACING depended on several undefined (missing) features. Thus, removing the dependency removes these undefined features from the propositional formula.

example, in terms of missing anomalies, the same missing feature `CONFIG_CPU_SUBTYPE_ST40` shown in Figure 7 was also discovered during our analysis at the file level. At the file level, only eight dead files are caused by this problem while at the block level, 19 code blocks are reported as dead because of the same problem. This suggests that working on the file level may end up solving the same problems, but with less information provided to the developer. We believe developers may find it easier to deal with the file level since the number of reported anomalies for the same problem are more manageable. However, there is of course the risk that some anomalies that are only unique to the blocks might be missed at the file level. Thus, it might be beneficial 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 they may move onto block level analysis to solve any remaining issues.

A second advantage is that by giving us a more manageable data set to work with, we were able to discover incorrect interpretation of Kconfig parsing as explained in the previous section.

A third advantage is that we are able to track the evolution of anomalies. This is difficult to do when working at the code block level because code blocks are identified by their line numbers in a code file. Unfortunately, these line numbers may change from one release to the other as more patches and changes are being applied to the kernel. In many situations, we thought that certain code block anomalies were fixed in one release, but only later discover that the anomaly still exists, but this piece of code has been moved to somewhere else in the file which changes the block number. Thus, tracking code block anomalies by the block number, anomaly description or line numbers can lead to inaccurate analysis.

VII. Threats to Validity

Internal Validity. In this work, we emphasize the importance of considering the make space constraints in variability analysis. However, statically parsing Makefiles is very tricky [2] especially in a system as large as Linux. There are certain parts of Makefiles that are not very regularly structured especially with the use of variables. In our work so far, there are some of these aspects which we ignore such as conditional blocks in Makefiles (e.g., `ifeq ($(CONFIG_SND),y)`) and `#define`'s. These blocks may indeed contain variability information; however, ignoring such blocks only ignores additional constraints. Thus, it may result in missed anomalies (i.e., anomalies which our analysis does not catch), but it will not result in any false positives. We believe that our current work shows the potential of the constraints in the Makefiles, and we plan to address the missing aspects of Makefiles in our future extension to this work.

Since we are extending the UNDERTAKER tool, any shortcomings in the original analysis will be reflected in our analysis. We discussed such an example in Section VI-B. We have reported this possible flaw in parsing to the UNDERTAKER team and they are currently investigating it. However, this should not greatly impact our results since this case does not occur frequently (less than 10 features having multiple conditional definitions). This can also be fixed by investigating other Kconfig parser tools such as the Kconfig parser developed by She [13], [14] as part of the Variability Analyses Tools.

Some of the anomalies we discover may not necessarily reflect errors. We chose to use the term *anomaly* precisely for this reason. A dead artifact may exist due to bad maintenance, and an undead artifact may be used as a form of double checking that certain conditions actually hold. In both cases, we believe that developers should still be aware of such anomalies since they are potential sources of errors and undesired behavior. However, in order to address the fact that developers may intentionally leave dead or undead artifacts behind, a whitelist approach can be adopted to allow developers to remove certain files from the analysis. The UNDERTAKER tool already has such a whitelist approach implemented.

External Validity. Since we are only using the Linux kernel as a subject of study in this work, we do not generalize our results to other systems. In order to generalize and categorize variability anomalies caused by build systems, we plan to apply our analysis to other systems in order to improve external validity. This will also allow us to discover how variability is generally implemented in build systems, and whether the technique used affects the quality of the software system.

VIII. Related Work

The variability in the implementation of the Linux kernel has recently been attracting considerable work. Zengler [19] has encoded the constraints in Kconfig as a single propositional logic formula which can be verified to ensure that the combination of features is valid. Such a formula will likely be huge, and would be very difficult to analyze. This is why we opted for using the propositional encoding by Tartler et al. [17] as they provide a slicing algorithm that only chooses the constraints related to the artifacts in question. Kästner et al. [6] also study variability at the level of the code blocks by

implementing variability parsers for programs written in Java and GNU C.

Berger et al. [3] show that the extraction of presence conditions of source code files from Makefiles are feasible. They extract them for the x86 architecture in Linux and for FreeBSD. In our work, we use our own implementation for extracting the constraints in the Makefiles, and we do so for all the CPU architectures present in Linux over several releases. Additionally, we show the effect of these constraints on the variability of the final compiled kernel image through the anomalies we detect.

Adams et al. [1] study the evolution of the Linux build system. Their work focuses on how Kbuild evolves over time in comparison to the source code. Similarly, McIntosh et al. [10] study the evolution of ANT and Maven. Such work suggests that studying build systems is important and that they consume a fair amount of the maintenance effort for any system. Other work [5], [11] has focused on how to improve the `make` tool itself, and to make it more efficient in terms of parsing Makefiles.

IX. Conclusions and Future Work

In this work, we show the importance of considering build systems (Makefiles in this case) in variability analysis. We study the Linux kernel which implements variability through three distinct artifacts: source code (code space), Kconfig files (kconfig space), and Makefiles (make space). Previous work [17] has developed the UNDERTAKER tool which only analyzes the code space and the kconfig space to detect code block anomalies. In this paper, we extract the make space constraints from the Makefiles in the Linux kernel, and extend the UNDERTAKER tool to include these constraints in the anomaly detection process.

By including the make space constraints in the analysis, we detect many additional anomalies at the code block level. We contrast the anomalies detected by the original UNDERTAKER tool, and our modified version of it over 11 kernel releases. We observe that the number of anomalies reported is overwhelming, and would not be beneficial when presented to a developer. Since Kbuild directly deals with the code files and not the code blocks, we decide to raise the analysis level to code files. This proves to be more useful and illustrative of the causes of these anomalies, and allows us to track the evolution of these anomalies over time. For both levels of analysis, we provide examples to illustrate the types of anomalies detected and how developers handle them.

We plan to extend this work to other software systems to be able to study more types of variability anomalies. We hope that this will lead to a means for developing a practical anomaly analysis framework for large configurable software systems which can also include visual support for presenting the anomalous code blocks and files to allow the developer to easily fix them. Such a framework can be achieved by using our current work with various previous work in the field (e.g., [3], [12], [17], [19].

Acknowledgments

We would like to thank our anonymous reviewers for their insightful comments, and helpful suggestions. We would also like to thank the UNDERTAKER team for their feedback, and for clarifying different aspects related to the tool.

References

[1] B. Adams, K. De Schutter, H. Tromp, and W. De Meuter. The evolution of the Linux build system. In *EVOL '07: Proceedings of the 3rd International ERCIM Symposium on Software Evolution*, volume 8, 2007.

[2] B. Adams, H. Tromp, K. De Schutter, and W. De Meuter. Design recovery and maintenance of build systems. In *ICSM '07: Proceedings of the 23rd International Conference on Software Maintenance*, pages 114 –123, 2007.

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

[4] L. Brown, 2009. http://git.kernel.org/?p=linux/kernel/git/torvalds/linux-2.6.git;a=commitdiff;h=c636f753b5b943f08fb3490e7f1a9b47aa5cc7d3.

[5] N. Jørgensen. Safeness of make-based incremental recompilation. In *FME '02: Proceedings of the International Europe Symposium on Formal Methods - Getting IT Right*, volume 2391 of *Lecture Notes in Computer Science*, pages 126–145. Springer Berlin / Heidelberg, 2002.

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

[7] G. Kumfert and S. Ravnborg. Kernel configuration and building in Linux 2.5. *Linux Symposium*, pages 197–212, 2003.

[8] R. Lotufo. On the complexity of maintaining the linux kernel configuration. Technical report, Electrical and Computer Engineering, University of Waterloo, 2009.

[9] R. Lotufo, S. She, T. Berger, K. Czarnecki, and A. Wasowski. Evolution of the Linux kernel variability model. In *SPLC'10: Proceedings of the 14th International Conference on Software Product Lines: Going Beyond*, pages 136–150. Springer-Verlag, 2010.

[10] S. McIntosh, B. Adams, and A. Hassan. The evolution of java build systems. *Empirical Software Engineering*, pages 1–31, 2011.

[11] P. Miller. Recursive make considered harmful. In *AUUGN 1998: Australian Unix User Group Newsletter*, page 19(1):14?25. 1998.

[12] S. Nadi and R. Holt. Make it or break it: Mining anomalies in linux kbuild. In *WCRE '11: Proceedings of the 18th Working Conference on Reverse Engineering*, 2011.

[13] S. She. http://code.google.com/p/linux-variability-analysis-tools/.

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

[15] J. Sincero, R. Tartler, C. Egger, W. Schröder-Preikschat, and D. Lohmann. Facing the Linux 8000 Feature Nightmare. In *EUROSYS '10: Proceedings of the 5th European conference on Computer systems*.

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

[17] R. Tartler, D. Lohmann, J. Sincero, and W. Schröder-Preikschat. Feature Consistency in Compile-Time Configurable System Software. In G. Heiser and C. Kirsch, editors, *Proceedings of the EuroSys 2011 Conference (EuroSys '11)*, pages 47–60, New York, NY, USA, 2011.

[18] R. Tartler, J. Sincero, W. Schröder-Preikschat, and D. Lohmann. Dead or alive: finding zombie features in the linux kernel. In *FOSD '09: Proceedings of the 1st International Workshop on Feature-Oriented Software Development*, pages 81–86, 2009.

[19] C. Zengler and K. Wolfgang. Encoding the Linux Kernel Configuration in Propositional Logic. In *ECAI '10: Proceedings of 19th European Conference on Artificial Intelligence (Workshop on Configuration)*, pages 51–56, 2010.